

Performance Controlled Power Optimization for Virtualized Internet Datacenters

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Yefu Wang

August 2011

© by Yefu Wang, 2011
All Rights Reserved.

This dissertation is dedicated to my parents and wife.

Acknowledgements

I would like to thank all the individuals who have inspired, encouraged, and advised me in the preparation of this dissertation. First and foremost, I would like to thank my advisor, Dr. Xiaorui Wang. His willingness to support my work and his guidance throughout my studies has allowed me to develop my skills as a researcher within a supportive team environment. I would further like to thank the other members of my committee: Dr. Gregory D. Peterson, Dr. Seddik M. Djouadi, and Dr. Dongjun Lee. I greatly appreciate their time and input to this dissertation.

I would also like to thank my friends and colleagues at University of Tennessee for helping me in my research: Kai Ma, Yanwei Zhang, Xue Li, Robert Deaver, Ming Chen, Xing Fu, Xiaodong Wang and, Shara K Galloway.

I also wish to thank the scientists who collaborated with me in my research, who shared their experiences and codes to help me with my research: Dr. Yuan Chen at HP, Dr. Ripal Nathuji at Microsoft, Dr. Faraz Ahmad at Purdue University.

Abstract

Modern data centers must provide performance assurance for complex system software such as web applications. In addition, the power consumption of data centers needs to be minimized to reduce operating costs and avoid system overheating. In recent years, more and more data centers start to adopt server virtualization strategies for resource sharing to reduce hardware and operating costs by consolidating applications previously running on multiple physical servers onto a single physical server. In this dissertation, several power efficient algorithms are proposed to effectively reduce server power consumption while achieving the required application-level performance for virtualized servers.

First, at the server level this dissertation proposes two control solutions based on dynamic voltage and frequency scaling (DVFS) technology and request batching technology. The two solutions share a performance balancing technique that maintains performance balancing among all virtual machines so that they can have approximately the same performance level relative to their allowed peak values. Then, when the workload intensity is light, we adopt the request batching technology by using a controller to determine the time length for periodically batching incoming requests and putting the processor into sleep mode. When the workload intensity changes from light to moderate, request batching is automatically switched to DVFS to increase the processor frequency for performance guarantees.

Second, at the datacenter level, this dissertation proposes a performance-controlled power optimization solution for virtualized server clusters with multi-tier

applications. The solution utilizes both DVFS and server consolidation strategies for maximized power savings by integrating feedback control with optimization strategies. At the application level, a multi-input-multi-output controller is designed to achieve the desired performance for applications spanning multiple VMs, on a short time scale, by reallocating the CPU resources and DVFS. At the cluster level, a power optimizer is proposed to incrementally consolidate VMs onto the most power-efficient servers on a longer time scale.

Finally, this dissertation proposes a VM scheduling algorithm that exploits core performance heterogeneity to optimize the overall system energy efficiency.

The four algorithms at the three different levels are demonstrated with empirical results on hardware testbeds and trace-driven simulations and compared against state-of-the-art baselines.

Contents

List of Tables	xi
List of Figures	xii
1 Introduction	1
2 Related Work	10
2.1 Power and Performance Management for Non-virtualized Servers . . .	10
2.2 Performance Management for Virtualized Servers	12
2.3 Power and Performance Management at Cluster Level or Datacenter Level	13
2.4 VM Scheduling	15
3 A Server-Level Algorithm Based on DVFS for Medium Workloads	17
3.1 Control Architecture	18
3.2 Load Balancing Controller	20
3.2.1 Problem Formulation	20
3.2.2 System Modeling	21
3.2.3 Controller Design	23
3.2.4 Control Analysis	25
3.3 Response Time Controller	26
3.4 System Implementation	28
3.4.1 Testbed	28

3.4.2	Control Components	29
3.5	Empirical Results	31
3.5.1	Load Balancing	31
3.5.2	Power Efficiency	33
3.5.3	Comparison to SIMPLE	37
3.6	Summary	39
4	Virtual Batching: A Server-Level Algorithm Based on Request Batching for Light Workloads	40
4.1	System Architecture	41
4.1.1	Performance Balancing	41
4.1.2	Integration of Request Batching and DVFS	43
4.1.3	Request Batching Policy	44
4.2	Performance Balancing Controller	46
4.2.1	Problem Formulation	46
4.2.2	System Modeling and Controller Design	47
4.2.3	Controller Analysis	48
4.3	Request Batching Controller	52
4.3.1	Problem Formulation	52
4.3.2	System Modeling and Controller Design	53
4.4	Integration of Virtual Batching and Server Consolidation	55
4.4.1	Motivation Studies	55
4.4.2	Integration of Virtual Batching and pMapper	56
4.5	System Implementation	59
4.5.1	Testbed	59
4.5.2	Simulation Environment	62
4.6	Evaluation	63
4.6.1	Baselines	63
4.6.2	Request Batching	64

4.6.3	Integration of Request Batching and DVFS	65
4.6.4	Comparison with Baselines for Bursty Workloads	68
4.6.5	Integration with Server Consolidation	71
4.7	Summary	74
5	A Datacenter-Level Algorithm Based on DVFS and Server Consol- idation	75
5.1	System Architecture	76
5.2	MIMO Response Time Controller	78
5.2.1	Problem Formulation	78
5.2.2	Response Time Controller Design	79
5.2.3	CPU Resource Arbitrator	82
5.3	Server Consolidation for Power Optimization	84
5.3.1	Problem Formulation	84
5.3.2	Power Model	85
5.3.3	Optimization Algorithm	86
5.4	System Implementation	89
5.4.1	Testbed	89
5.4.2	Control Components	91
5.4.3	Simulator	93
5.5	Experimentation	93
5.5.1	Response Time Control	94
5.5.2	Server Consolidation for Power Optimization	97
5.5.3	Simulation Results for Large Data Centers	100
5.6	Summary	102
6	Energy-Efficient Virtual Machine Scheduling in Performance-Asymmetric Multi-Core Architectures	104
6.1	Background and Motivations	105
6.1.1	Energy Efficiency	105

6.1.2	CPU Scheduling in Xen	107
6.1.3	Motivation Example	108
6.1.4	Performance and Power Models	110
6.2	Energy-Efficiency Factor	111
6.2.1	Definition of Energy-Efficiency Factor	112
6.2.2	Online Estimation of energy-efficiency factor	114
6.3	Energy-Efficient VM Scheduling	117
6.3.1	Algorithm	119
6.3.2	Discussions	120
6.3.3	Integration with an Application-Level Performance Manage- ment Algorithm	121
6.4	System Implementation	122
6.4.1	Testbed and Workloads	122
6.4.2	Implementation in Xen Hypervisor	122
6.5	Evaluation	124
6.5.1	Improvement in Energy Efficiency	124
6.5.2	Integration with an Application-level Performance Manage- ment Algorithm	127
6.5.3	Homogeneous Workloads	128
6.6	Summary	129
7	Conclusion	131
	Bibliography	133
	Vita	145

List of Tables

3.1	Model orders and RMSE.	22
4.1	Models and parameters of the six real servers used in the testbed. . .	60
4.2	Server models and parameters used in large-scale simulations.	62
5.1	Server models and parameters used in large-scale simulations.	93
6.1	Power and performance behaviors of 2 VCPUs on fast and slow cores in an example.	109
6.2	Performance events measured	115

List of Figures

3.1	Two-layer control architecture.	18
3.2	Model validation.	22
3.3	Performance comparison between load balancing controller and OPEN, under a workload increase to VM2 from time 300s to time 600s. . . .	32
3.4	Performance of the load balancing controller under system model changes.	34
3.5	Performance and power efficiency comparison between the two-layer control solution and OPEN, under a workload increase to VM2 from time 300s to time 600s.	35
3.6	Performance and power consumption with different workloads.	37
3.7	Performance comparison between two-layer control solution and SIM- PLE.	38
4.1	Request batching system architecture.	41
4.2	Euclidean norm of the output vector Y of the closed-loop system (4.7) and (4.8) under different levels of uniform model variations.	52
4.3	Average server CPU utilization under different VM memory require- ments.	56
4.4	A typical run of Virtual Batching. The controllers are activated at time 750s.	65
4.5	Integration of request batching and DVFS.	66
4.6	Request batching switching to DVFS at time 300s during a typical run. .	68

4.7	Comparison with baselines for bursty workloads based on a real 24-hour trace file.	69
4.8	Power and performance comparison between the integrated solution and pMapper on the testbed.	72
4.9	Comparison between the integrated solution and pMapper in a simulated large-scale data center with 5,415 VMs from a real trace file. . .	73
5.1	System architecture, including a response time controller for every multi-tier application and a cluster-level power optimizer.	77
5.2	Our testbed consisting of 4 servers.	91
5.3	Response time of all 8 applications in the cluster.	95
5.4	Typical runs of the response time controller and the baseline pMapper under a workload increase from time 600s to 1200s on App5.	96
5.5	Response time of App5 under different workloads.	97
5.6	Response time of App5 under different set points.	97
5.7	Power consumption of the cluster. Power optimizer is activated at time 600s.	98
5.8	Response time of an application during the VM running its database service migrates to another server.	99
5.9	Average power savings of the cluster with different number of applications.	99
5.10	Response time of the applications in the cluster with different number of applications.	99
5.11	Energy consumption per VM in 7 days under different numbers of VMs.	101
5.12	Percentage of energy savings (over pMapper) as results of DVFS and better consolidation of IPAC.	101
5.13	Number of migrations of IPAC and pMapper in 7 days under different numbers of VMs.	102

6.1	The energy efficiency of the system can be improved by selecting an energy-efficient VCPU-core mapping.	109
6.2	Proposed framework for estimating the energy-efficiency factor. The offline part derives an energy-efficiency model which is used online. .	114
6.3	Energy-efficient load balancing. VCPUs with high energy-efficiency factors are dynamically moved to fast cores.	119
6.4	Relative energy efficiency of our energy-efficient algorithm and the default Xen Credit Scheduler in 4 sets of core frequency settings. . . .	125
6.5	Relative energy efficiency of proposed energy-efficient algorithm and the default Credit Scheduler running a mix of 24 different benchmarks.	126
6.6	Relative energy efficiency of proposed energy-efficient algorithm (relative to that of the default Credit Scheduler) running web workloads (RUBBoS and SPECjbb).	126
6.7	The performance of the proposed energy-efficient algorithm and the default Credit Scheduler running web workloads (RUBBoS and SPECjbb).	127
6.8	Throughput (average and standard deviation) and energy consumption for 1 million request per application.	128
6.9	Relative energy efficiency of the energy-efficient algorithm and the default Credit Scheduler running homogeneous workloads.	129

Chapter 1

Introduction

Recent years have seen a rapid growth of enterprise data centers that host thousands of high-density servers and provide outsourced business-critical IT services. There are two key challenges for effectively operating a modern data center. First, service owners need to be assured by meeting their required Service-Level Objectives (SLOs) such as response time and throughput. Second, power consumption has to be minimized in order to reduce operating costs and avoid failures caused by power capacity overload or system overheating due to increasing high server density.

In recent years, more and more data centers start to adopt server virtualization strategies for resource sharing to reduce hardware and operating costs. Virtualization technologies such as VMware, Xen [19] and Microsoft Virtual Servers can consolidate applications previously running on multiple physical servers onto a single physical server, and so effectively reduce the energy consumption of a data center by shutting down unused servers.

In this dissertation, several power efficient algorithms are proposed to effectively reduce server power consumption while achieving the required application-level performance for virtualized enterprise servers.

A two-layer control architecture based on processor dynamic frequency and voltage scaling (DVFS)

A well-known approach to reducing power consumption of enterprise servers is to transition the hardware components from high-power states to low-power states whenever possible. Most components in a modern computer server such as processors, main memory and disks have adjustable power states. Components are fully operational, but consume more power in high-power states while having degraded functionality in low-power states. An energy-efficient server design is to have run-time measurement and control of desired application performance by adjusting the power states of the processor, *i.e.*, processor dynamic frequency and voltage scaling (DVFS). As a result, we can have the desired server performance while reducing power consumption to the maximum degree.

However, this approach cannot be directly applied to today’s popular virtualized server environments. As a result, virtual machines on the same physical server are correlated, because the state transition of any hardware component of the physical server will affect the performance of all the virtual machines. As some virtual machines may have light workloads while others may have heavy workloads at the same time, reducing server power consumption based on the performance level of a single virtual machine may cause others to violate their performance specifications. Therefore, the performance of all virtual machines has to be controlled simultaneously for intelligent power saving.

This dissertation proposes a novel two-layer control architecture to provide response time guarantees for virtualized enterprise servers. The primary control loop adopts a MIMO control strategy to maintain load balancing among all virtual machines so that they can have roughly the same response time relative to their maximum allowed response times. The secondary loop then controls the relative response times of all the virtual machines to a desired level by adapting CPU frequency for power efficiency.

Request Batching for Energy Conservation in Virtualized Servers

Data centers generally provision based on a worst-case scenario which leads to the average server utilization in modern data centers being lower than 10% [60]. These

under-utilized servers spend a large portion of their time in an idle state. Several recent studies have shown that a server uses approximately 60% of its required peak power when it is idle [8, 48]. This over-provisioning leads to large amounts of energy waste in data centers. On the other hand, average US electricity costs increased 146% between 2000 and 2008 [17]. Therefore, reducing energy waste, while guaranteeing SLA agreements, can lead to significantly reduced operating costs in data centers.

The DVFS-based approach effectively reduces the power consumption of the computer systems when the server has a medium intensity workload (we define workload intensity in Section 4.1). However, the capability of this approach to reduce power consumption is limited when the server has a low-intensity workload due to two reasons. First, when the utilization of the processor is very low, the leakage power, which cannot be significantly reduced by DVFS, contributes a major portion of the power consumption. Second, many high-performance processors only allow a small range of DVFS levels and even the lowest level provides a higher speed than is required for some light workloads. For example, in a case study on our testbed, the power consumption of an idle server with an Intel Xeon 5360 processor can only be reduced from 163W to 158W when the processor is transitioned from the highest DVFS level to the lowest one.

To further reduce energy consumption, processors need to be put into sleep states such as Deep Sleep. In Deep Sleep, the processor is paused and consumes significantly less power. For example, the power consumption of an Intel processor may be reduced to 23% of its peak value when it is switched to the Deep Sleep state [35]. When the processor is in Deep Sleep, the server can be configured to use Direct Memory Access (DMA) to place incoming packets into memory buffers for processing when the processor is returned to the active state, thus avoiding harming the functionality of the hosted server applications. Therefore, to save more power (than what DVFS can do) for servers with light workloads, we can perform *request batching* to put the processor into the Deep Sleep state when there are no incoming requests. During the sleep time, we delay and batch the requests when they arrive and wake the processor

up when the earliest request in the batch has been kept pending for a certain batching timeout.

However, it is challenging to perform request batching directly on a virtualized server. Virtualization technologies such as Xen, VMware, and Microsoft Hyper-V allow provisioning multiple virtual machines onto a single physical server.

To address these problems and further save power consumption for light workloads, this dissertation proposes Virtual Batching, a novel request batching solution for virtualized enterprise servers with primarily light workloads. Our solution dynamically allocates the CPU resource such that all the virtual machines can have approximately the same performance level relative to their allowed peak values. Based on the uniform level, our solution then determines the time length for periodically batching incoming requests and putting the processor into sleep. When the workload intensity changes from light to medium, request batching is automatically switched to DVFS to increase processor frequency for performance guarantees.

A datacenter-level algorithm based on server consolidation

Though both DVFS and Virtual Batching algorithms effectively reduce the dynamic power of the system, they cannot minimize the system leakage power for maximized power savings.

Virtualization technologies can consolidate applications previously running on multiple physical servers onto a smaller number of physical servers, effectively reducing the power consumption of a server cluster by shutting down unused servers. More importantly, live migration [16] allows the movement of a virtual machine (VM) from one physical host to another with a reasonably short downtime [31]. This function makes it possible to use server consolidation as an online management approach, *i.e.*, having run-time estimation of resource requirements of every VM and dynamically re-mapping VMs to physical servers using live migration. When a more power-efficient VM-server mapping is found, unused servers can be put into the sleep mode for reduced power consumption. A recent study [57] shows that server consolidation can achieve a power saving of 34%, which is significantly more than

that achieved by DVFS (17%). However, the down side of consolidation is that it migrates VMs from one host to another, which may incur a high overhead in terms of both time and server resources.

It is important to integrate application-level performance control and cluster-level server consolidation for maximized power savings due to two reasons. First, most existing work of VM placement assumes that the resource requirements of the VMs are known a priori or can be estimated through measuring the resource utilization. However, resource utilization commonly differs from resource requirements, especially when the server is overloaded. Thus, it is preferable to have an application-level performance controller that can dynamically allocate system resource in response to application requirement variations. Second, a performance controller may fail to guarantee the application performance when the server is overloaded due to possible workload increase. Therefore, it is preferable to have a cluster-level consolidation algorithm to dynamically re-map VMs to physical servers to resolve the overload problem.

This dissertation proposes an integrated management solution to minimize power consumption for virtualized server clusters while providing application-level performance assurance. Each application-level performance controller adopts a MIMO control strategy to maintain the desired performance and reduce power consumption through DVFS and dynamic CPU resource reallocation. The cluster-level power optimizer then consolidates the VMs onto the most power-efficient servers and places unused servers into the sleep mode for power savings on a much longer time scale, to amortize the migration overhead.

Energy-Efficient Virtual Machine Scheduling in Performance-Asymmetric Multi-Core Architectures

The design of multi-core processors with identical cores faces an inevitable dilemma: using only slow cores may compromise the performance of single-threaded applications, while using only fast cores may unnecessarily lead to low energy efficiency. As a result, multi-core architectures with asymmetric core performance

(*i.e.*, a mix of fast and slow cores) have recently shown great promise, because they can offer applications the flexibility to run on a fast core for high single-threaded performance, or on a group of slow cores for high parallelism with better energy efficiency [43, 71, 45, 69, 28]. Multi-core systems with performance heterogeneity can be beneficial to today’s high-performance computer servers that commonly host virtual machines (VMs) on multi-core architectures, because VMs hosting different applications often have different resource needs. Therefore, they have different preferences on the type of cores they run from an energy-efficiency perspective. Unfortunately, existing VM scheduling algorithms, such as Credit Scheduler in Xen, map VMs to the cores without considering the heterogeneities of the cores [15, 88, 42, 27, 58] which can lead to degraded overall energy efficiency of the system.

There are two challenges in developing an energy-efficient solution for scheduling VMs on heterogeneous cores with performance asymmetry, such as cores with different CPU frequencies. First, it’s difficult to characterize the application needs and resource usage. In many virtualized platforms, such as Amazon EC2, the platform administrators have no knowledge of the applications running inside the VMs because the users of the VMs may run any application at any time. Thus, technologies requiring accurate knowledge of the applications (*e.g.*, [71]) are impractical from a VM scheduling perspective. Furthermore, an application’s needs and performance/power characteristics can vary during its execution and need to be dynamically evaluated. Second, in order to achieve better energy efficiency, both performance and power need to be considered. Mapping VMs to cores solely based on the performance metrics (*e.g.*, [69, 71, 43]) may lead to unnecessarily poor overall system energy efficiency.

In general, heterogeneous multi-core systems fall into two categories: multi-core systems containing heterogeneous cores with different special purpose functions (function asymmetry) and systems containing cores with homogeneous functions but heterogeneous performance and power characteristics (performance asymmetry) [43]. In this dissertation, we focus on performance asymmetry and particularly multiple cores with heterogeneous CPU frequencies, because they are technically mature and

widely used. For example, many current designs, such as Intel’s Single-chip Cloud Computer (SCC)[36], allow cores in a group to have a frequency different from those in other groups. Some designs, such as the AMD Independent Dynamic Core Technology [6] even allow per-core frequency scalings. To exploit these group-level or core-level frequency scaling technologies, many recently proposed power management solutions [78, 64, 86, 54, 11] dynamically run cores at different frequencies, making the systems behave like multi-core systems with performance asymmetry.

In this dissertation , we present a practical and effective solution that schedules VMs on heterogeneous cores with different frequencies to maximize the overall energy efficiency of the system. We first introduce a metric termed *energy-efficiency factor* to characterize the power and performance behaviors of a virtual CPU (VCPU) of a VM on different cores. We then develop a light-weight method for dynamically estimating the energy-efficiency factors of VCPUs using hardware performance counters which are widely available in most multi-core processors. We then propose a scheduling algorithm that maps the VCPUs to different types of cores based on their estimated energy-efficiency factors, in such a way that the overall energy efficiency (*i.e.*, performance per watt) is maximized.

Contributions

Specifically, this dissertation has the following contributions.

For the DVFS-based algorithm:

- We model the response times of the applications running in virtual machines using system identification, and validate the models with data measured in real experiments.
- We design a two-layer control architecture based on control theory and analyze the control performance.
- We present the system architecture of our control loops and the implementation details of each component.

- We present empirical results to demonstrate that our control solution can effectively reduce server power consumption while achieving desired response time for virtualized enterprise servers.

For Virtual Batching:

- We propose a novel request batching technique in virtualized environments to achieve further energy conservation when the server workload becomes light.
- We integrate request batching and DVFS to provide energy conservation for virtualized servers when the workload varies at runtime. Our solution allows energy to be saved across a wide range of workload intensities.
- We design a two-layer control architecture that relies on feedback control theory as a theoretical foundation to achieve analytical assurance of control accuracy and system stability.
- We conduct experiments on a hardware testbed with real trace files and present empirical results to demonstrate the effectiveness of our control solution to conserve energy for virtualized enterprise servers.

For the server consolidation algorithm:

- We design a power optimizer that consolidates VMs onto a smaller number of physical servers in the cluster based on the resource requirements determined by the application-level performance controllers to achieve minimized power consumption.
- We introduce the system architecture of our integrated power management solution, and the implementation details of each component.
- We present both empirical results on a hardware testbed and simulation results to demonstrate that our solution can effectively reduce cluster power consumption while achieving the desired response time for multi-tier applications hosted

in the server clusters. In addition, we show that our solution outperforms a state-of-the-art baseline, pMapper [79].

For scheduling algorithm:

- We introduce energy-efficiency factor, a metric to characterize the power and performance behaviors of a VCPU on different cores and propose a statistical method to estimate the energy-efficiency factors of VCPUs online.
- We propose an VM scheduling algorithm that maps VMs to heterogeneous cores with different CPU frequencies to maximize a system’s overall energy efficiency. Our algorithm can be applied directly to existing multi-core hardware and is completely transparent to the users.
- We present a method that integrates our VM scheduling algorithm, which requires no application-level knowledge of the VMs, with an application-level performance management solution to improve the application-level energy efficiency.
- We implement our scheduling algorithm in the Xen VM hypervisor on top of an existing VM scheduling algorithm and evaluate our algorithm on a real testbed using various standard benchmarks, such as SPEC CPU2006, SPECjbb2005, and RUBBoS. The results demonstrate the our algorithm effectively improves the energy efficiency over the default scheduler of Xen.

The rest of this dissertation is organized as follows. Chapter 2 discusses the related work. Chapter 3 proposes a server-level algorithm based on DVFS for medium workloads. Chapter 4 proposes Virtual Batching, a server-level algorithm based on request batching for light workloads. Chapter 5 proposes a datacenter-level algorithm based on DVFS and server consolidation. Chapter 6 proposes an energy-efficient virtual machine scheduling in performance-asymmetric multi-core architectures. Chapter 7 concludes the dissertation.

Chapter 2

Related Work

Power and performance are both important concerns for internet datacenters. Research work related to the topic in this dissertation falls into three categories.

2.1 Power and Performance Management for Non-virtualized Servers

Control theory has been applied to manage application-level performance metrics for virtualized computer servers. Padala et al. [60] propose a two-layered architecture to control resource utilization and response time. Xu et al. [90] use predictive control for dynamic resource allocation in enterprise data centers. Liu et al. [50] control the mean response time of individual applications by adjusting the percentage of allocated CPU entitlements. Zhang et al. [91] adjust the resource demands of virtual machines for fairness and efficiency. Wang et al. [87] present an adaptive controller for more robust performance. However, all the aforementioned work is not designed to reduce power consumption for virtualized servers.

Some prior work has been proposed to use power as a tool for guaranteeing required application-level SLOs in a variety of heuristic-based ways, such as using approximation algorithms [12]. More closely related to the algorithm proposed in

Chapter 3, Horvath et al. [32] use dynamic voltage scaling (DVS) to control end-to-end delay in multi-tier web servers. Zhu et al. [93] develop a feedback control scheduling algorithm using DVS. Sharma et al. [70] effectively apply control theory to control application-level quality of service requirements. Chen et al. [13] also present a feedback controller to manage the response time in a server cluster. Although they all use control theory to manage system performance while reducing power consumption, they cannot be directly applied to virtualized computing environments because multiple application servers are correlated due to shared hardware resource.

Several research projects [55, 72, 62] have successfully applied control theory to explicitly control power or cooling of computing servers. Lefurgy et al. [48] have shown that control-theoretic solution outperforms a commonly used heuristic-based solution by having more accurate power control and less overhead. Ranganathan et al. [65] propose a negotiation-based algorithm to allocate power budget to different servers in a blade enclosure for better performance. Wang et al. [83] develop a MIMO control algorithm for cluster-level power control. Those projects are different from our work because they control power consumption only and thus cannot provide guarantees for application-level performance. They also are not designed for virtualized enterprise servers.

Elnozahy et al. [22] proposed a request batching technique for non-virtualized web servers by putting the processor into the sleep mode when the server is idle, significantly reducing the energy consumption of web servers when the workload is light. In contrast, the solution proposed in Chapter 4 provides additional energy savings by allowing the processor to sleep even when the server is not completely idle. Further, the solution in Chapter 4 provides a performance balancing controller to dynamically adjust the CPU resource allocation to the VMs such that request batching can be utilized in virtualized environments.

2.2 Performance Management for Virtualized Servers

Virtualization technology has provided a promising way to manage application performance by dynamically reallocating resources to VMs. Several management algorithms have been proposed to control application performance for virtualized applications [60, 89, 14, 59]. For example, Padala et al. [59] used MIMO control theory to control throughputs for virtualized servers. In contrast, the four algorithms proposed in this dissertation control response time, a user-perceived performance metric. In addition, the solution proposed in Chapter 4 provides energy savings through DVFS and request batching.

A recent work [57] discusses a heuristic solution to power control in virtualized environments. In contrast to their work, the work proposed in Chapter 3 rely on rigorous control theory for systematically developing control strategies with analytic assurance of control accuracy and system stability.

VM migration is an important tool for resource and power management in virtualized computing environments. Several recent studies propose to dynamically consolidate VMs to a smaller number of servers and putting the unused servers into the sleep mode for energy savings [62, 41, 79, 80]. In contrast to these algorithms, the solution proposed in Chapter 4 has several advantages. First, the request batching technique used in our solution has a much less overhead than VM migration, thus can be used on a much smaller time scale. Second, due to resource limitations, such as the memory and bandwidth, sometimes it is infeasible to further consolidate VMs to a smaller number of servers even if the utilization of the processor is still low. In this situation, request batching can save additional energy by putting the processor to the sleep mode periodically. The algorithm in Chapter 4 can be integrated with VM consolidation algorithms to save power in different situations.

Zhu et al. [92] and Nathuji et al. [56] proposed giving fewer resources to the operating system or VM. Compared with the algorithm proposed in Chapter 4, none of these consider using a batching technique.

2.3 Power and Performance Management at Cluster Level or Datacenter Level

Energy conservation has been one of the most important design constraints for Internet server clusters and datacenters. The majority of the prior work has attempted to reduce power consumption by reducing the energy consumption of individual server components [47]. Several algorithms have been proposed to utilize DVFS to manage the energy consumption at the cluster level [83, 63, 56, 9, 30], and the data center level [84]. However, compared with the algorithms proposed in Chapter 4, Chapter 5, these algorithms cannot further reduce energy consumption when the processor is already at the lowest DVFS level. More closely related to this dissertation, Heo et al. [30] developed a power optimization algorithm using DVFS and shutting down unused servers for large-scale applications spanning several non-virtualized servers with a load balancer. In contrast to their work, the solution proposed in Chapter 5 provides even greater power savings by consolidating underutilized servers onto a smaller number of active servers.

Virtualization technology has provided a promising way to manage application performance by dynamically reallocating resources to VMs. Several management algorithms have been proposed to control application performance for virtualized servers [60, 89, 14, 59, 46]. For example, Padala et al. [59] proposed to control throughputs for virtualized servers. In contrast, the algorithm proposed in Chapter 5 controls response time, a user-perceived performance metric. In addition to performance assurance, we integrate DVFS with server consolidation for maximal power savings. Furthermore, most existing work assumes simple single-tier applications [60, 89, 14] while the solution proposed in Chapter 5 relies on MIMO control theory to deal with complex web applications that may span multiple VMs.

VM migration is an important tool for resource and power management in virtualized computing environments. Some prior work focuses on using migration to satisfy the resource requests of VMs [92, 33, 41, 85]. In contrast, the solution

proposed in Chapter 5 takes the total power consumption of the cluster as the design goal and designs a power-efficient algorithm while guaranteeing the desired performance requirements. Several recent studies propose to solve the VM-server mapping problem for power savings [62, 41, 79, 80]. For example, Raghavendra et al. [62] used a greedy bin-packing algorithm as part of their coordinated control solution to minimize power consumption in virtualized data centers. The solution proposed in Chapter 5 has two differences compared to their work. First, our solution provides response time guarantees to multi-tier applications while their work focuses on single-tier applications. Second, our solution dynamically re-allocates CPU resource among the VMs while their performance control relies only on DVFS. Thus, our solution can allocate CPU resource to VMs based on their needs, which saves additional power.

In Internet data centers, workload dispatching is another widely used approach to controlling application-level performance [82]. Kusic et al. [46] have developed an algorithm to use workload dispatching in virtualized server clusters for performance guarantee and power efficiency. This type of algorithm requires duplicating an application several times, incurring a much larger storage overhead in comparison to our migration based solution.

The VM-server mapping problem discussed in Chapter 5 falls in the category of *vector-packing* problems (or its subset bin-packing problem) in computer science. A detailed survey of this problem is given in [26]. Several recent papers [23, 29, 10] demonstrate that the Minimum Bin Slack (MBS) algorithm provides better results in terms of solution quality when compared to traditional algorithms. In this dissertation we extend the basic MBS algorithm by considering power efficiency and a more realistic constraint to solve our problem to reduce the power consumption in a cluster of heterogeneous servers.

2.4 VM Scheduling

VM scheduling is one of the most important problems in virtualization research. For example, Cherkasova et al. [15] compare three schedulers in Xen, SEDF (Simple Earlier Deadline First), BVT (Borrowed Virtual Time), and Credit. These scheduling algorithms are performance-oriented and do not optimize for energy efficiency. Some prior work also attempt to improve the default Xen schedulers for better performance. Kim et al. propose a VM scheduling algorithm that schedules VMs based on the priorities of the processes inside the VMs [42]. This technique increases the response time to IO events by 5-22%. Govindan et al. [27] advocate an algorithm that prefers providing CPU resources to the VM with the largest number of network packages to minimize the delay of the packages caused by virtualization. Ongaro et al. [58] observe that traditional VM schedulers only consider the sharing of CPU resources with no concern for I/O performance. Although an I/O-intensive VM may receive adequate total CPU resource, it may not receive the CPU resource at the right time. Their algorithm is designed to boost the I/O performance of VMs by giving priority to the VMs with pending I/O requests. In contrast to these schedulers, our algorithm improves the energy efficiency in heterogeneous multi-core systems. The Credit Scheduler in Xen has been recently improved by [88] to first saturate a package on the chip while leaving others idle, so that lower power states can be used. Our algorithm is complementary to this technique by further improving the energy efficiency of the non-idle packages on the chip.

For non-virtualization environments, several previous studies investigate CPU scheduling algorithms that improve the performance of web servers. A survey of these papers is available in [66]. Compared with these techniques, our algorithm requires no knowledge of the application-level details, thus is more transparent to the users. Gupta et al. [28] demonstrated discussed the benefits of using performance asymmetric chips to host web applications. Several prior papers propose thread scheduling algorithms for heterogeneous multi-core systems in non-virtualized

operating systems to improve the overall performance [43, 45, 71, 69, 11]. Our algorithm is completely transparent and require no knowledge of the applications running inside the VMs while the solutions in these papers require prior knowledge of the applications from offline profiling. For example, [43] requires knowing a stall threshold calibrated from the workloads. As discussed in Chapter 1, a VM scheduler should not make any assumptions on the workloads as the system administrator may not know the VMs that will run in the system. Furthermore, Kumar et al. [44] use simulation to show the potential energy benefits in dynamically switching workloads among heterogeneous cores based on the online measurement of power and performance behaviors. Compared with their work, our algorithm does not rely on an online power sensor and thus can be used to existing hardware which typically does not implement core-level power sensors due to the design and layout complexity in providing separate power domains for individual cores. In addition, our algorithm is designed for VM scheduling, which is different from thread scheduling in these papers in that VM scheduling provides the management interfaces for system administrators to adjust the CPU share of the VMs, thus provides the flexibility to integrate with application-level management algorithms, as shown in Section 6.3.3.

Chapter 3

A Server-Level Algorithm Based on DVFS for Medium Workloads

As discussed in Chapter 1, a traditional energy-efficient server design is to have runtime measurement and control of the desired application performance by adjusting the power states of the processor, *i.e.*, processor dynamic frequency and voltage scaling (DVFS). However, this approach cannot be directly applied to virtualized server environments. This is due to the fact that virtual machines on the same physical server are correlated; the state transition of any hardware component of the physical server will affect the performance of all the virtual machines.

In this chapter, we propose a novel two-layer control architecture to provide response time guarantees for virtualized enterprise servers based on DVFS. The primary control loop adopts a MIMO control strategy to maintain load balancing among all virtual machines so that they can have roughly the same response time relative to their maximum allowed response times. The secondary loop then controls the relative response times of all the virtual machines to a desired level by adapting CPU frequency for power efficiency.

Note that we assume the workloads for the VMs are independent of each other.

3.1 Control Architecture

In this section, we provide a high-level description of our control system architecture which features a two-layer controller design. As shown in Figure 3.1, the primary control loop, *i.e.*, the load balancing control loop, maintains load balancing among all virtual machines by adjusting the CPU resource (*i.e.*, fraction of CPU time) allocated to them. As a result, the virtual machines can have roughly the same *relative* response time, which is defined as the ratio between the actual response time and the maximum allowed response time for each virtual machine. The secondary control loop, *i.e.*, the response time control loop, then controls the relative average response time to the desired level (*e.g.*, 90%) for all virtual machines by manipulating CPU frequency. Consequently, power efficiency can be achieved for the physical server while the desired response times can also be guaranteed for applications in all the virtual machines. We choose to control response time instead of other metrics such as resource utilization because response time is a user-perceived metric, which captures users' experience of Internet service performance. In this chapter, we control the average response time to reduce the impact of the long delay of any single web request. However, our control architecture can also be applied to control the maximum response time.

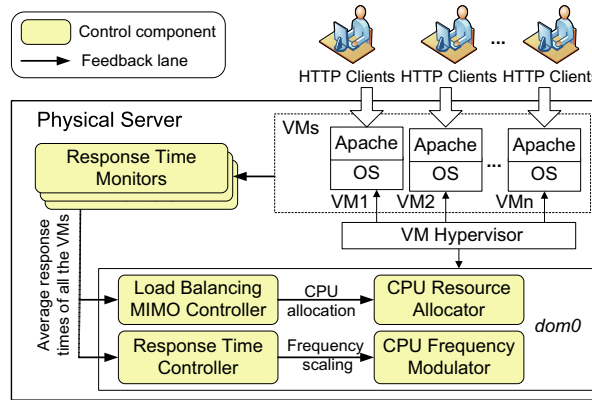


Figure 3.1: Two-layer control architecture.

As shown in Figure 3.1, the key components in the load balancing control loop include a *MIMO controller*, a *response time monitor* for each virtual machine, and a *CPU resource allocator*. The control loop is invoked periodically and its period is selected such that multiple HTTP requests can be received during a control period and the actuation overhead is acceptable. The following steps are invoked at the end of every control period: 1) The response time monitor of each virtual machine measures the average response time in the last control period, and sends the value to the controller through the feedback lane; 2) The controller calculates the relative response time for each virtual machine, and the average relative response time of all virtual machines. The difference between the relative response time of each virtual machine and the average value is used as the *controlled variable* with the set point as zero; 3) The controller computes the amount of CPU resource to be allocated to each virtual machine, *i.e.*, the *manipulated variable*, in the next control period; 4) The CPU resource allocator then enforces the desired CPU resource (*i.e.*, CPU time) allocation.

The key components in the response time control loop include a *controller*, the same response time monitors and a *CPU frequency modulator*. The control loop is also invoked periodically. Every invocation of this loop may change the CPU frequency (thus the total available CPU resource) of the physical server. Therefore, to minimize the influence to the load balancing control loop, the period of this loop is selected to be much longer than the settling time of the load balancing control loop. This guarantees that the load balancing loop can always enter its steady state within one control period of the response time control loop, so that the two control loops are decoupled and can be designed independently. In each control period of the response time control loop, the controller receives the average response times of all the virtual machines from the monitors. Since the load balancing controller is in steady state at the end of each control period of this controller, all virtual machines should have the same relative response time, which is the *controlled variable* of this loop. Based on this uniform value, the controller then computes the new CPU frequency level, *i.e.*,

the *manipulated variable*, for the processor of the physical server, and then sends the level to the CPU frequency modulator. The modulator changes the CPU frequency level of the processor accordingly.

Since the core of each control loop is its controller, we introduce the design and analysis of the two controllers in the next two sections, respectively. The implementation details of other components are given in Section 3.4.

3.2 Load Balancing Controller

In this section, we present the problem formulation, modeling, design and analysis of the load balancing controller.

3.2.1 Problem Formulation

Load balancing control can be formulated as a dynamic optimization problem. We first introduce some notation. A physical server hosts n virtual machines. T_1 is the control period. d_i is the maximum allowed response time of i^{th} virtual machine VM_i . $rt_i(k)$ is the average response time of VM_i in the k^{th} control period. $r_i(k)$ is the relative response time of VM_i . Specifically, $r_i(k) = rt_i(k)/d_i$. $r(k)$ is the average relative response time of all virtual machines in the k^{th} control period, namely, $r(k) = \sum_{i=1}^n r_i(k)/n$. $e_i(k)$ is the control error of VM_i , *i.e.*, $e_i(k) = r_i(k) - r(k)$. $e_i(k)$ is our controlled variable and its set point is zero.

To allocate CPU resource to VM_i , the virtual machine hypervisor assigns an integer *weight* $w_i(k)$ to VM_i . The CPU resource is allocated to each virtual machine proportionally to its weight. In order to linearize the system, instead of directly using $w_i(k)$ as the control input, we find a constant operating point w_i for each virtual machine, which is the typical default weight of the virtual machine in Xen. The weight change, namely, $\Delta w_i(k) = w_i(k) - w_i$ is used as our manipulated variable.

Given a control error vector, $\mathbf{e}(\mathbf{k}) = [e_1(k) \dots e_n(k)]^T$, the control goal at the k^{th} control point (*i.e.*, time kT_1) is to dynamically choose a weight change vector

$\Delta \mathbf{w}(\mathbf{k}) = [\Delta w_1(k) \dots \Delta w_n(k)]^T$ to minimize $e_i(k+1)$ for all virtual machines:

$$\min_{\{\Delta w_j(k) | 1 \leq j \leq n\}} \sum_{i=1}^n (e_i(k+1))^2 \quad (3.1)$$

3.2.2 System Modeling

In order to have an effective controller design, it is important to model the dynamics of the controlled system, namely the relationship between the controlled variables (*i.e.*, $e_i(k), 1 \leq i \leq n$) and the manipulated variables (*i.e.*, $\Delta w_i(k), 1 \leq i \leq n$). However, a well-established physical equation is usually unavailable for computer systems. Therefore, we use a standard approach to this problem called *system identification* [24]. Instead of trying to build a physical equation between the manipulated variables and controlled variable, we infer their relationship by collecting data in experiments and establish a statistical model based on the measured data.

Based on control theory [18], we choose to use the following difference equation to model the controlled system:

$$\mathbf{e}(\mathbf{k}) = \sum_{i=1}^{m_1} \mathbf{A}_i \mathbf{e}(\mathbf{k} - \mathbf{i}) + \sum_{i=1}^{m_2} \mathbf{B}_i \mathbf{w}(\mathbf{k} - \mathbf{i}) \quad (3.2)$$

where m_1 and m_2 are the orders of the control output and control input, respectively. $\mathbf{A}_i$ and $\mathbf{B}_i$ are control parameters whose values need to be determined by system identification.

For system identification, we need to first determine the right orders for the system, *i.e.*, the values of m_1 and m_2 in the difference equation (3.2). The order values are normally a compromise between model simplicity and modeling accuracy. In this chapter, we test different system orders as listed in Table 3.1. For each combination of m_1 and m_2 , we generate a sequence of pseudo-random digital white noise [24] to stimulate the system and then measure the control output (*i.e.*, $e_i(k), 1 \leq i \leq n$) in each control period. Our experiments are conducted on the testbed introduced in detail in Section 3.4. Based on the collected data, we use the *Least Squares Method*

(*LSM*) to iteratively estimate the values of parameters $\mathbf{A}_1$ and $\mathbf{B}_1$. The values in Table 3.1 are the estimated accuracy of the models in term of Root Mean Squared Error (RMSE). From Table 3.1, we can see that the model with $m_1 = 2$ and $m_2 = 1$ has a small error while keeping the orders low. We then use a step-like signal to validate the results of system identification. Figure 3.2 demonstrates that the predicted output of the selected model is sufficiently close to the actual system output. Therefore, the resultant system model from our system identification is:

$$\mathbf{e}(\mathbf{k}) = \mathbf{A}_1\mathbf{e}(\mathbf{k} - 1) + \mathbf{A}_2\mathbf{e}(\mathbf{k} - 2) + \mathbf{B}_1\Delta\mathbf{w}(\mathbf{k} - 1) \quad (3.3)$$

where $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{B}_1$ are $n \times n$ constant control parameter matrices whose values are determined by system identification.

Table 3.1: Model orders and RMSE.

	$m_1 = 0$	$m_1 = 1$	$m_1 = 2$
$m_2 = 1$	0.0930	0.0862	0.0858
$m_2 = 2$	0.0926	0.0860	0.0857
$m_2 = 3$	0.1008	0.0953	0.0949

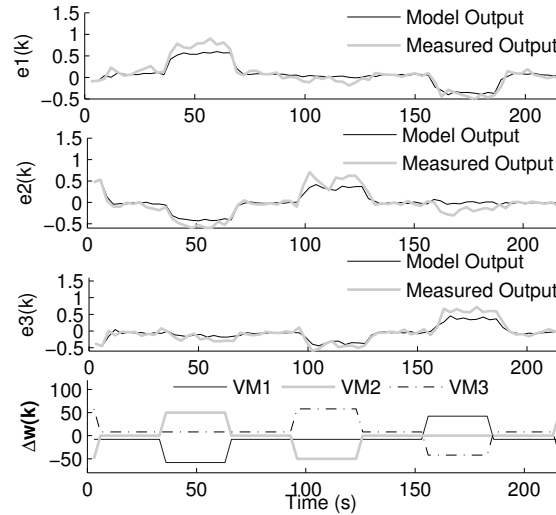


Figure 3.2: Model validation.

3.2.3 Controller Design

We apply the Linear Quadratic Regulator (LQR) control theory [18] to design the controller based on the system model (3.3). LQR is an advanced control technique that can deal with coupled MIMO control problems. Compared to other MIMO control techniques such as Model Predictive Control (MPC), LQR has a smaller runtime computational overhead. To design the controller, we first convert our system model to a state space model. The state variables are defined as follows:

$$\mathbf{x}(\mathbf{k}) = \begin{bmatrix} \mathbf{e}(\mathbf{k}) \\ \mathbf{e}(\mathbf{k} - 1) \end{bmatrix} \quad (3.4)$$

We then augment the state space model by considering the accumulated control errors for improved control performance:

$$\mathbf{v}(\mathbf{k}) = \lambda \mathbf{v}(\mathbf{k} - 1) + \mathbf{e}(\mathbf{k} - 1) \quad (3.5)$$

where λ is a parameter that has to be less than 1 to avoid the accumulated error going unbounded. In our experiments, we choose $\lambda = 0.8$. The value of λ is chosen by comparing the performance of the controller in simulations. Our final state space model is as follows:

$$\begin{bmatrix} \mathbf{x}(\mathbf{k}) \\ \mathbf{v}(\mathbf{k}) \end{bmatrix} = \mathbf{A} \begin{bmatrix} \mathbf{x}(\mathbf{k} - 1) \\ \mathbf{v}(\mathbf{k} - 1) \end{bmatrix} + \mathbf{B} \Delta \mathbf{w}(\mathbf{k} - 1) \quad (3.6)$$

where

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{0} & \lambda \mathbf{I} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}.$$

Based on the augmented model (3.6), we design the LQR controller by choosing gains to minimize the following quadratic cost function:

$$\begin{aligned} \mathbf{J} = & \sum_{k=1}^{\infty} \begin{bmatrix} \mathbf{e}^T(\mathbf{k}) & \mathbf{v}^T(\mathbf{k}) \end{bmatrix} \mathbf{Q} \begin{bmatrix} \mathbf{e}(\mathbf{k}) \\ \mathbf{v}(\mathbf{k}) \end{bmatrix} \\ & + \sum_{k=1}^{\infty} \Delta \mathbf{w}^T(\mathbf{k}) \mathbf{R} \Delta \mathbf{w}(\mathbf{k}) \end{aligned} \quad (3.7)$$

where $\mathbf{Q}$ and $\mathbf{R}$ are weighting matrices that determine the trade-off between the control error and the control gain. This optimization is conducted over an infinite time horizon. The first item in (3.7) represents the control errors and the accumulated control errors. By minimizing the first item, the closed-loop system can converge to the desired set points. The second item in (3.7) represents the control penalty. Minimizing the second item ensures that the controller will minimize the changes in the control input, *i.e.*, the weight change of each virtual machine. A general rule to determine the values of $\mathbf{Q}$ and $\mathbf{R}$ is that larger $\mathbf{Q}$ leads to faster response to workload variations, while larger $\mathbf{R}$ lets the system be less sensitive to system noise. The LQR controller can be implemented using the Matlab command `dlqry` to solve the optimization problem (3.7) and get the controller gain matrix $\mathbf{F}$ as:

$$\mathbf{F} = \begin{bmatrix} \mathbf{K}_{\mathbf{P1}} & \mathbf{K}_{\mathbf{P2}} & \mathbf{K}_{\mathbf{I}} \end{bmatrix} \quad (3.8)$$

where $\mathbf{K}_{\mathbf{P1}}$, $\mathbf{K}_{\mathbf{P2}}$ and $\mathbf{K}_{\mathbf{I}}$ are constant controller parameters. In the k^{th} control period, given the control error vectors $\mathbf{e}(\mathbf{k})$ and $\mathbf{e}(\mathbf{k} - 1)$, the control input, *i.e.*, the weight change of each virtual machine can be computed using the following controller equation:

$$\Delta \mathbf{w}(\mathbf{k}) = -\mathbf{F}[\mathbf{e}(\mathbf{k})^T \quad \mathbf{e}(\mathbf{k} - 1)^T \quad \mathbf{v}(\mathbf{k})^T]^T \quad (3.9)$$

3.2.4 Control Analysis

In this subsection, we analyze the system stability when the designed MIMO controller is used on a different server with a system model different from the nominal model described by (3.6). A fundamental benefit of the control-theoretic approach is that it gives us theoretical confidence for system stability, even when the controller is used in a different working condition. We now outline the general process for analyzing the stability of a virtualized server controlled by the load balancing controller.

1. Derive the control inputs $\Delta \mathbf{w}(\mathbf{k})$ that minimize the cost function based on the *nominal* system model described by (3.6).
2. Conduct automated system identification on the target virtualized server and get an actual system model as:

$$\begin{bmatrix} \mathbf{x}(\mathbf{k}) \\ \mathbf{v}(\mathbf{k}) \end{bmatrix} = \mathbf{A}' \begin{bmatrix} \mathbf{x}(\mathbf{k} - 1) \\ \mathbf{v}(\mathbf{k} - 1) \end{bmatrix} + \mathbf{B}' \Delta \mathbf{w}(\mathbf{k} - 1) \quad (3.10)$$

where $\mathbf{A}'$ and $\mathbf{B}'$ are actual parameter matrices that may be different from $\mathbf{A}$ and $\mathbf{B}$ in (3.6).

3. Derive the closed-loop system model by substituting the derived control inputs $\Delta \mathbf{w}(\mathbf{k})$ into the actual system model (3.10). The closed-loop system model is in the form:

$$\begin{bmatrix} \mathbf{x}(\mathbf{k}) \\ \mathbf{v}(\mathbf{k}) \end{bmatrix} = (\mathbf{A}' - \mathbf{B}'\mathbf{F}) \begin{bmatrix} \mathbf{x}(\mathbf{k} - 1) \\ \mathbf{v}(\mathbf{k} - 1) \end{bmatrix} \quad (3.11)$$

4. Derive the stability condition of the closed-loop system described by (3.11). According to control theory, the closed-loop system is stable if all the eigenvalues of matrix $(\mathbf{A}' - \mathbf{B}'\mathbf{F})$ locate inside the unit circle in the complex space.

In our stability analysis, we assume the constrained optimization problem is *feasible*, i.e., there exists a set of CPU resource weights within their acceptable ranges that can make the response time on every virtual machine equal to the average response time. If the problem is infeasible, no controller can guarantee the set points through CPU resource adaptation. The technology presented in this dissertation may not work for infeasible problems. For example, if the response time of a VM is dominated by the time spent in IO subsystems, the response time may not be controlled simply by allocating CPU resources. A Matlab program is developed to perform the above stability analysis procedure automatically.

3.3 Response Time Controller

In this section, we present the problem formulation, modeling, and design of the response time controller.

Problem Formulation. We first introduce some notation. T_2 , the control period is selected to be longer than the settling time of the load balancing controller such that all virtual machines can guarantee to have the same response time within T_2 . As defined before, $r_i(k)$ is the relative response time of VM_i in the k^{th} control period. $r(k)$ is the average relative response time of all virtual machines. R_s is the set point, i.e., the desired relative response time for the system. $f(k)$ is the current CPU frequency of the physical server, relative to the highest frequency of the CPU. For example, $f(k) = 1$ means the CPU is currently running at its highest frequency.

In the k^{th} control period, given current average relative response time $r(k)$, the control goal is to dynamically choose a CPU frequency $f(k)$ such that $r(k)$ can converge to the set point R_s after a finite number of control periods.

System Modeling. Similar to the load balancing controller, we adopt system identification to model the system. An accurate model to capture the relationship between $r(k)$ and $f(k)$ is normally nonlinear due to the complexity of computer systems. To simplify the controller design with a linear model, instead of directly

using $r(k)$ and $f(k)$ to model the system, we use their respective differences with their operating points, r and f , which are defined as the typical values of $r(k)$ and $f(k)$. Specifically, the control output of the model is $\Delta r(k) = r(k) - r$, the control input is $\Delta f(k) = f(k) - f$, and the set point of the control output is $\Delta R_s = R_s - r$. An example way to choose operating point is to select the middle value of a range of available CPU frequencies as f , and measure the average relative response time with this frequency and use it as r .

Based on system identification, the system model is:

$$\Delta r(k) = a_1 \Delta r(k-1) + b_1 \Delta f(k-1) \quad (3.12)$$

where a_i and b_i are model coefficients.

Controller Design and Analysis. *Proportional-Integral (PI)* control [24] has been widely adopted in industry control systems. An important advantage of PI control is that it can provide robust control performance despite considerable modeling errors. Following standard control theory, we design our PI controller as:

$$\Delta f(k) = k_1(\Delta R_s - \Delta r(k)) + k_2 \sum_{i=1}^k (\Delta R_s - \Delta r(i)) \quad (3.13)$$

where k_1 and k_2 are control parameters that can be analytically chosen to guarantee system stability and zero steady state error, using standard control design methods [24].

Although the system has been proven to be stable when the system model (3.12) is accurate, stability has to be reevaluated when the response time controller is applied to a different server with different workloads. The general process of analyzing the stability of the response time controller with a different system model is similar to that of the load balancing controller.

3.4 System Implementation

In this section, we introduce our testbed and the implementation details of the control loops.

3.4.1 Testbed

Our testbed includes two physical computers. One is named *Server* and hosts three virtual machines (VMs). The other one is named *Client* and emulates a number of HTTP clients that generate HTTP requests to Server. Both computers are running Fedora Core 8 with Linux kernel 2.6.21. Server is equipped with an Intel Xeon X5365 processor and 4GB RAM. The processor has 4 cores and supports 4 frequency levels: 2GHz, 2.33GHz, 2.67GHz and 3GHz. Server and Client are connected via an Ethernet switch.

Xen 3.1 is used as the virtual machine monitor. In Xen, the Xen hypervisor is the lowest layer with the most privileges. When the physical computer and the hypervisor boot, domain 0, *i.e.*, *dom0*, is the first guest operating system that is booted automatically. Dom0 has some special management privileges such as it can direct resource access requests to the hardware. In our testbed, dom0 is used to start the three VMs. In addition, dom0 is used to distribute the HTTP requests from Client to the VMs. The two control loops also run in dom0.

Each VM is allocated 500MB of RAM and 10GB of hard disk space. An Apache server is installed on each VM and runs as a virtualized web server. The Apache servers respond to the HTTP requests from Client with a dynamic web page written in PHP. This example PHP file runs a set of mathematical operations. To be more general, the Apache server on each VM is assigned a different maximum allowed response time, *i.e.*, 180ms for VM1, 190ms for VM2, and 200ms for VM3.

The client side workload generator is the Apache HTTP server benchmarking tool (ab), which is designed to stress test the capability of a particular Apache installation. This tool allows users to manually define the concurrency level, which is the number

of requests to perform in a very short time to simulate multiple clients. A concurrency level of 40 is used in our experiments to do system identification and most experiments if not otherwise indicated. Three instances of ab are configured on Client and each one is used to send requests to the Apache Server in a VM.

The power consumption of the physical server is measured with a WattsUp Pro power meter with an accuracy of 1.5% of the measured value.

3.4.2 Control Components

We now introduce the implementation details of each component in our control loops.

Response Time Monitor. To eliminate the impact of network delays, we focus on controlling the server-side response time in this dissertation. The response time monitor is implemented as a small daemon program that runs on each VM. The monitor periodically inserts multiple sample requests into the requests that are sent from Client to the Apache server. Two time stamps are used when a sample request is inserted and when the response is received. The difference is used as the server-side response time.

CPU Resource Allocator. We use Credit Scheduler [2], Xen’s proportional share scheduler to allocate available CPU resource. Credit Scheduler is designed for load balancing of virtual CPUs across physical CPUs on a host. This scheduler allocates CPU in proportion to the *weights* assigned to the VMs. In each control period, the load balancing controller computes a *weight* for every VM. The weights are then truncated to integers and given to the VMs.

CPU Frequency Modulator. We use Intel’s SpeedStep technology to enforce the desired CPU frequency. In Linux systems, to change CPU frequency, one needs to install the *cpufreq* package and then use root privilege to write the new frequency level into the system file `/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed`. However, this is more complicated with the Xen virtual machine monitor because Xen lies between the Linux kernel and the hardware, and thus prevents the kernel

from directly modifying the hardware register based on the level in the system file. As a result, the source code of Xen 3.1 has been hacked to allow the modification of the register.

The Intel CPU used in our experiments supports only four discrete frequency levels. However, the new CPU frequency level periodically received from the response time controller could be any value that is not exactly one of the four supported frequency levels. Therefore, the modulator code must locally resolve the output value of the controller to a series of supported frequency levels to approximate the desired value. For example, to approximate 2.89GHz during a control period, the modulator would output the sequence 2.67, 3, 3, 2.67, 3, 3, etc on a smaller timescale. To do this, we implement a first-order delta-sigma modulator, which is commonly used in analog-to-digital signal conversion. The detailed algorithm of the first-order delta-sigma modulator can be found in [48].

Controllers. Both the two controllers are implemented in dom0 to receive response times from the monitors, and run the control algorithms presented in Section 3.2 and Section 3.3, respectively. The control period of the load balancing controller is selected to be 3 seconds such that multiple HTTP requests can be received. Based on an analysis, the longest settling time of the load balancing controller is 12 seconds when the system model varies within a wide range. The control period of the response time controller is therefore selected to be 24 seconds, which is longer than the worst-case settling time of the load balancing controller. The set point of the response time controller is configured to be 0.88 to give some leeway to the Apache web servers. Hence, the average response time is 88% of the maximum allowed response time when the system is in steady state. Note that the leeway is selected based on the typical variations in the workload and should be tuned in real systems.

3.5 Empirical Results

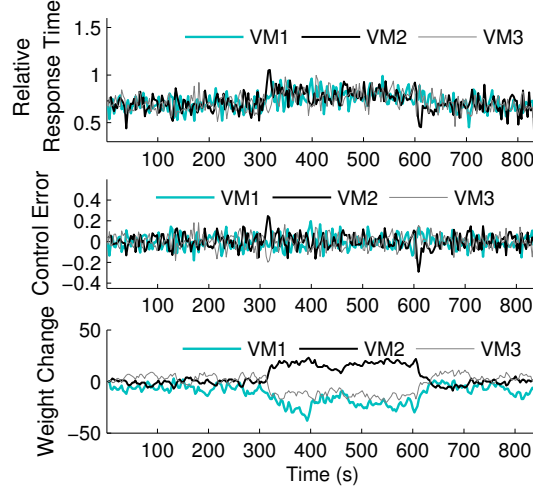
In this section, we first evaluate the performance of the load balancing controller. We then test the two-layer control solution for power efficiency. Finally, we compare our control solution with a baseline power-efficient solution.

We use two baselines for comparison in this chapter. OPEN is an open-loop solution that gives fixed CPU resource to each of the three VMs in the system. As a result, OPEN cannot achieve the same relative response time for different VMs when they have non-uniform workloads. OPEN is used in Sections 3.5.1 and 3.5.2 to compare with our control solution. OPEN always use the maximum CPU frequency. SIMPLE uses only the response time controller presented in Section 3.3 to control the *average* response time of all the VMs, without maintaining load balancing among them. SIMPLE is similar to the control algorithms designed for non-virtualized servers, and is used to demonstrate that those algorithms cannot be directly applied to virtualized computing environments.

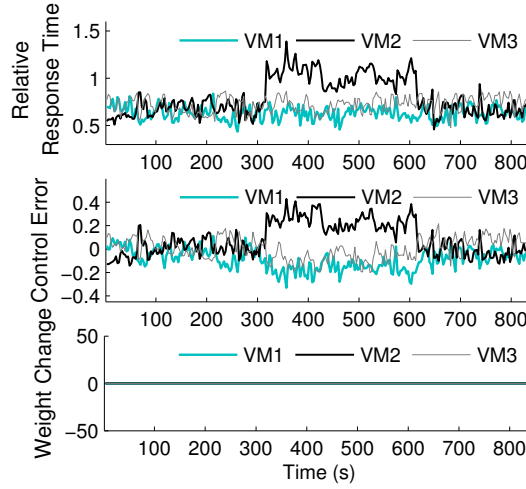
3.5.1 Load Balancing

In this experiment, we disable the response time controller to test the performance of the load balancing controller. As shown in Figure 3.3(a), the three VMs are initially configured to have the same relative response time. At time 300s, the workload of VM2, increases significantly. This is common in many web applications. For example, breaking news on a major newspaper website may incur a huge number of accesses in a short time. To stress test the performance of our controller in such an important scenario, we double the concurrency level from 40 to 80 for VM2 from time 300s to time 600s to emulate workload increase. Figure 3.3(a) shows that the load balancing controller can maintain the same relative response time for all the VMs by dynamically increasing the weight of VM2 to allocate more CPU resource to VM2. As the result of load balancing, all the VMs have a slightly longer relative response time but all

of them have relative response times lower than 1 after time 300s, which means the desired response times have been guaranteed.



(a) Load balancing controller.



(b) OPEN

Figure 3.3: Performance comparison between load balancing controller and OPEN, under a workload increase to VM2 from time 300s to time 600s.

Figure 3.3(b) shows the performance of OPEN in the same scenario. Clearly, during the workload increase of VM2, the relative response time of VM2 increases

significantly while the response times of the other two VMs remain the same. As a result, the relative response time of VM2 is higher than 1 most of the time from time 300s to time 600s, which means that VM2 violates its maximum allowed response time. Under OPEN, the control error between the relative response time of each VM and the average relative response time is also unacceptably large during the workload increase period. The inferior performance of OPEN is caused by its policy of using fixed weights to allocate CPU resource in a static way. Consequently, OPEN cannot adjust CPU resource among different VMs in response to non-uniform workload variations. This experiment demonstrates that the load balancing controller can effectively maintain the same relative response time among all the VMs.

As discussed in Section 3.2, our system model is the result of system identification with a concurrency level of 40 and a CPU frequency of 0.87. To test the robustness of the load balancing controller when it is used on a system that is different from the one used to do system identification, we conduct two sets of experiments with wide ranges of concurrency level and CPU frequency, respectively. Figure 3.4(a) shows the means and the standard deviations of the relative response times of the VMs when the concurrency level varies significantly from 6 to 86. Figure 3.4(b) shows the results when the CPU frequency varies from 0.67 to 0.99. The two figures demonstrate that the load balancing controller works effectively to achieve the same relative response time for all the VMs even when the actual system model is different from the nominal model used to design the controller.

3.5.2 Power Efficiency

In this experiment, we enable both the load balancing controller and the response time controller to examine power efficiency. We conduct the experiments in the same scenario discussed in the last subsection, with the same workload increase to VM2. Figure 3.5a shows the results of our two-layer control solution. The experiment starts with the CPU of the physical server running at the highest frequency. As discussed

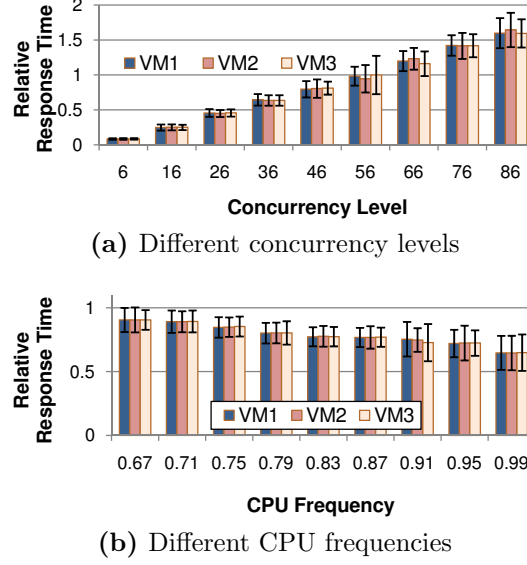
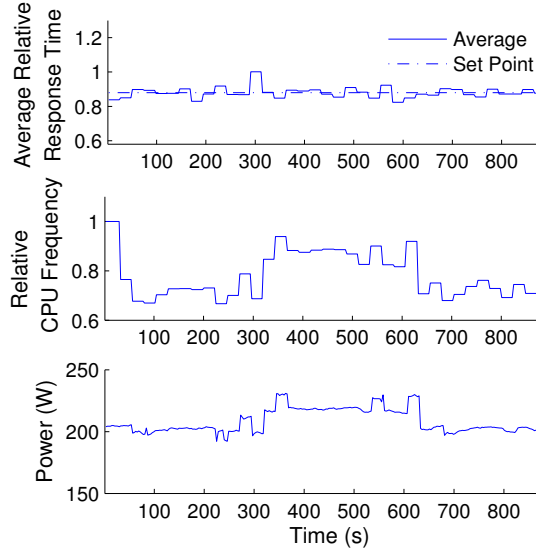


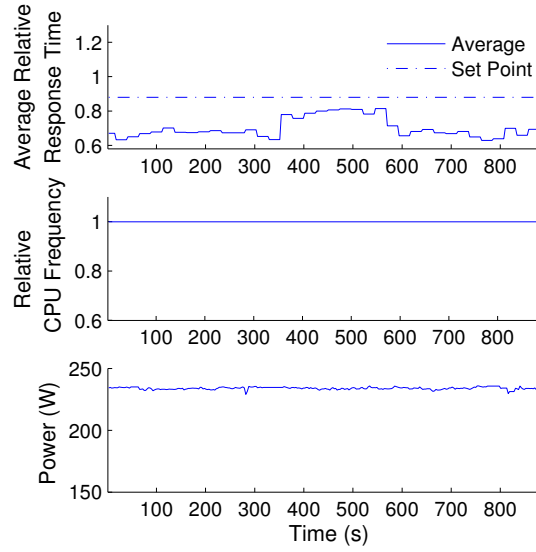
Figure 3.4: Performance of the load balancing controller under system model changes.

before, the load balancing controller works effectively to achieve the same relative response time for all the VMs. Since the response time is initially unnecessarily shorter than the desired set point, the response time controller lowers the relative CPU frequency from 1 to 0.67 for power efficiency. At time 300s, the workload of VM2 is doubled, which causes the average relative response time to start increasing. The response time controller detects the increase and then adjusts the relative CPU frequency to 0.95, such that the system can run faster to achieve the desired response time. As a result, the average relative response time has been maintained at the desired level. At time 600s when the workload of VM2 is reduced by half, the response time controller throttles the CPU again to achieve power saving while maintaining the desired response time. Throughout the experiment, the average relative response time has been successfully kept to be lower than 1 most of the time, which means the application-level performance, *i.e.*, desired response times have been guaranteed.

Figure 3.5b shows the results of OPEN. OPEN always runs the CPU at the highest frequency, just like most today's computer systems without dynamic power



(a) Two-layer control solution



(b) OPEN

Figure 3.5: Performance and power efficiency comparison between the two-layer control solution and OPEN, under a workload increase to VM2 from time 300s to time 600s.

management. From the beginning to time 300s, OPEN has an unnecessarily short average response time at the cost of excessive power consumption. At time 300s when the workload of VM2 is doubled, the average response time increases significantly as OPEN relies on fixed CPU resource allocation and CPU frequency. Note that even though it is possible to configure OPEN in a way to achieve power efficiency by running CPU at a lower frequency initially to make the average response time exactly equal to the desired level at the beginning. Doing so may cause the system to violate the maximum allowed response time when the workload increase occurs at time 300s, since OPEN cannot dynamically adjust CPU frequency for desired response time. In this experiment, the average energy consumption of the system under OPEN is 233.9 W while it is 205.5 W under the two-layer control solution. On average, with our control solution, a power saving of 28.4 W has been achieved while the desired response time has also been guaranteed.

To examine the performance and power efficiency of our control solution with different workloads, a set of experiments are conducted by changing the concurrency level. As introduced before, higher concurrency level leads to larger number of HTTP requests addressed to each VM. Figure 3.6a shows the performance results. Our control solution can achieve the desired relative response time while OPEN always has unnecessarily short response time, which comes at the cost of excessive power consumption as shown in Figure 3.6b. The power consumption of the system under our control solution is always lower than that under OPEN, and increases as the workload increases to achieve the desired response time. Note that our control solution has a relative response time shorter than the set point when the concurrency level is 36, as shown in Figure 3.6a. That is because the response time controller saturates at the lowest available CPU frequency and thus cannot further increase the response time.

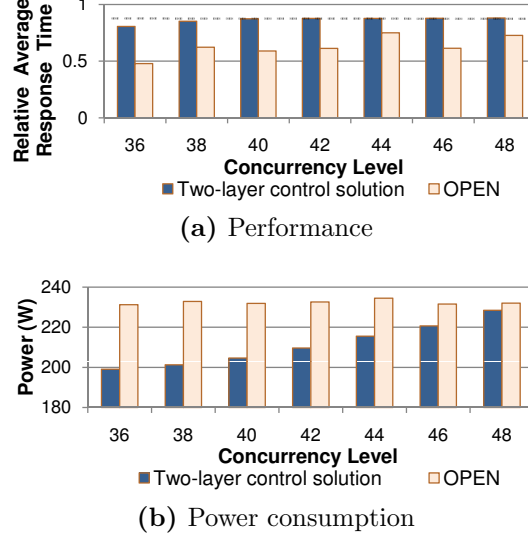
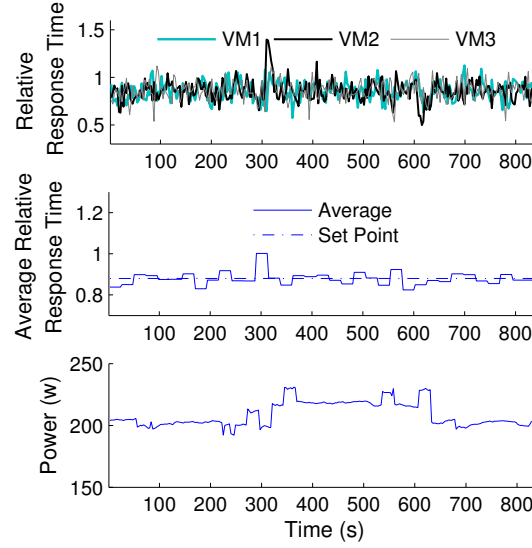


Figure 3.6: Performance and power consumption with different workloads.

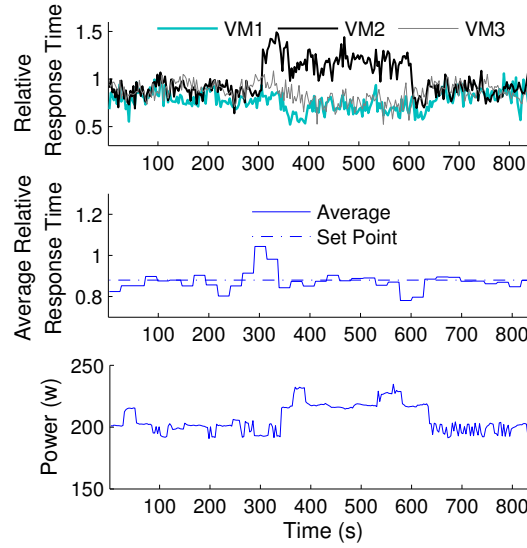
3.5.3 Comparison to SIMPLE

In this experiment, we compare our two-layer control solution with SIMPLE to show that traditional approaches to power management for non-virtualized servers cannot be easily applied to virtualized computing environments. Figure 3.7a shows the results of the two-layer control solution while Figure 3.7b shows the results of SIMPLE. The two control solutions work similarly to control the average relative response time by adjusting CPU frequency except an important difference: the two-layer solution has a load balancing controller which effectively maintains the same relative response time among all the VMs. For both solutions, the desired average response times have been achieved at the beginning with considerable power saving. However, when VM2 has a doubled workload at time 300s, the response time of VM2 under SIMPLE significantly violates the maximum allowed value. This is because SIMPLE only controls the average relative response time of all the VMs without load balancing. In contrast, the two-layer solution quickly allocates more CPU resource to VM2 for load balancing after VM2 starts to violate the maximum allowed response time at time 300s. The two-layer solution then adjusts CPU frequency based on the uniform

relative response time achieved by load balancing. As a result, all the VMs achieve their desired response times. This experiment demonstrates that maintaining load balancing among all VMs is important for power management in virtualized enterprise servers.



(a) Two-layer control solution



(b) SIMPLE

Figure 3.7: Performance comparison between two-layer control solution and SIMPLE.

3.6 Summary

In this chapter, we have presented a two-layer control architecture based on well-established control theory. The primary control loop adopts a MIMO control strategy to maintain load balancing among all virtual machines so that they can have approximately the same relative response time. The secondary performance control loop then manipulates CPU frequency for power efficiency based on the uniform relative response time achieved by the primary loop. Empirical results demonstrate that our control solution can effectively reduce server power consumption while achieving required response times for all virtual machines.

Chapter 4

Virtual Batching: A Server-Level Algorithm Based on Request Batching for Light Workloads

As discussed in Chapter 1, the DVFS-based in Chapter 3 effectively reduces the power consumption of the computer systems when the server has a medium intensity workload (we define workload intensity in Section 4.1). However, the capability of this approach to reduce power consumption is limited when the server has a low-intensity workload. This chapter proposes Virtual Batching, a novel request batching solution for virtualized servers with primarily light workloads. Our solution dynamically allocates CPU resources such that all the virtual machines can have approximately the same performance level relative to their allowed peak values. Based on this uniform level, our solution determines the time length for periodically batching incoming requests and putting the processor into sleep. When the workload intensity changes from light to moderate, request batching is automatically switched to DVFS to increase processor frequency for performance guarantees. Empirical results based on a hardware testbed and real trace files show that Virtual Batching can achieve

the desired performance with more energy conservation than several well-designed baselines, *e.g.*, 63% more, on average, than a solution based on DVFS only.

4.1 System Architecture

In this section, we provide a high-level description of the Virtual Batching system architecture. The goal of Virtual Batching is to control the response times of all the virtual machines on a physical server to their administrator-defined set points while minimizing the energy consumption of the server. To achieve this goal, we adopt three techniques: request batching, DVFS, and CPU resource allocation. Figure 4.1 provides an overview of the system architecture. In this dissertation, we assume that every VM runs a web application and the applications in different VMs are independent of each other. We also assume that the web requests to different VMs are independent of each other. These assumptions are usually valid because the VMs can belong to different customers.

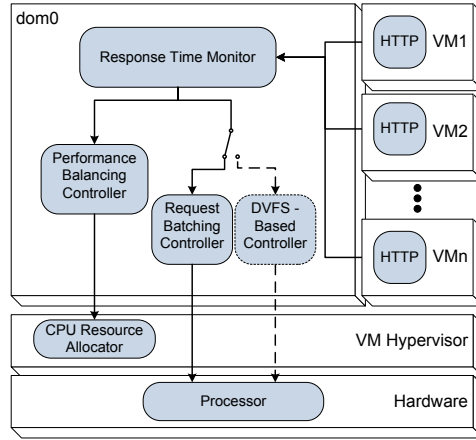


Figure 4.1: Request batching system architecture.

4.1.1 Performance Balancing

Implementing a technique such as request batching or DVFS directly in a virtualized environment is challenging because both methods rely on modifying the performance

of the physical system (via CPU hardware states) to manage energy consumption. Because hardware state transitions affect the performance of all virtual machines hosted on a physical server, the virtual machines are correlated. Without effective resource management, the methods of controlling the response times of enterprise servers inside the virtual machines are limited; using the average response time has a chance of allowing the busier servers to violate their response time goals while using the longest response time among the servers may waste CPU resource and energy. To overcome this limitation, our design uses a *performance balancing controller* to balance the relative response times of all the virtual machines by dynamically allocating the CPU resource. The relative response time is defined as $Rt_{measured,i}/Rt_{setpoint,i}$ where $Rt_{measured,i}$ is the measured response time of the server in the i^{th} virtual machine and $Rt_{setpoint,i}$ is the corresponding set point defined by the system administrator for the i^{th} virtual machine. This method allows the server in each virtual machine to have a different set point which gives flexibility in how virtual machines are provisioned on physical servers.

The performance balancing controller periodically uses the response time monitor to measure the average response time of the server applications in each virtual machine hosted on the server. It then adjusts the fraction of CPU time allocated to each virtual machine and gives more CPU time to virtual machines with relative response times above the average. Once the CPU allocations have been determined, they are enforced by a CPU Resource Allocator (*e.g.*, Xen’s credit scheduler).

To design the performance balancing controller, we need to address the following challenges. First, the number of VMs running on the server may change at runtime, so the performance balancing controller should be able to adapt to any of these changes. Second, when the workload is light, it is possible that a VM may become completely idle for a certain interval of time such that the response time monitor cannot give a reading of the response time. To address the two challenges, we design our performance balancing controller as a collection of VM-level controllers. Every VM-level controller reads the relative response time of the VM from the VM-level

response time monitor, collects the average relative response time of all non-idle VMs running on the server, and controls the VM-level relative average response time to the average relative response time.

4.1.2 Integration of Request Batching and DVFS

With a well-designed performance balancing controller, the relative response times of the individual VMs can be controlled to be close to each other. Thus, the average relative response time of the VMs indicates the server-level performance. We adopt two server-level power management techniques, request batching and DVFS, to minimize the energy consumption while guaranteeing the average relative response times of the VMs hosted on the server. Both techniques provide a compromise between the average relative response time and energy consumption. A request batching-based controller periodically controls the average relative response time by tuning the *duty cycle*, *i.e.*, the fraction of time that the processor is in the active state. The system then enforces the duty cycle by switching the processor between the sleep and active states. A longer duty cycle results in a shorter response time but higher energy consumption. Similarly, the DVFS-based controller manages the response time by tuning the DVFS level. A higher DVFS level means a shorter response time and higher energy consumption.

The two techniques are targeted for different workload intensity ranges of the server. As discussed in Section 1, the request batching-based controller can be more power efficient when the workload intensity is relatively low and the DVFS-based controller works better when the workload intensity is moderate. Virtual Batching uses both controllers, dynamically switching between them. When the request batching-based controller is running, it uses the average relative response time to determine the duty cycle and drive the average relative response time to the set point. If the average relative response time is longer than the set point, the controller will increase the duty cycle to allow the processor to operate in the active

state for a larger fraction of time. When the relative average response time is still higher than the set point and the duty cycle is already increased to 100%, we assume that the current intensity of the workload has become medium. We then deactivate the request batching-based controller and switch to the DVFS-based controller. The DVFS-based controller will then increase the CPU frequency to control the response time to the set point. When the DVFS-based controller is running, it tunes the CPU frequency to control the average relative response time. When the average relative response time is still shorter than the set point after the CPU frequency has been decreased to the lowest level, we assume that the current intensity of the workload is light. We then deactivate the DVFS-based controller and activate the request batching-based controller. It is important to note that light and medium workloads are *not* defined based on a preset CPU utilization point. Instead, once the request batching controller needs to switch to the DVFS controller in order to guarantee application-level performance, the workload intensity is assumed to have increased from light to medium, and vice versa.

In this chapter, the DVFS-based controller is designed based on the control algorithm presented in Chapter 3. The focus of this chapter is on the request batching controller and the integration of request batching with DVFS to handle different workload intensities.

4.1.3 Request Batching Policy

To minimize the server energy consumption under the response time constraint, we design a request batching policy that includes two steps.

In the first step, when the server is completely idle, the processor is put into Deep Sleep until new web requests arrive. This step requires that the server should be able to automatically wake up on demand when a web request comes. These mechanisms are often available in computer systems. For example, network adapters are capable of waking up the processor from sleep modes using the support of Wake-on-LAN

feature. The ACPI interface allows waking up the processor at a given point of time in the future. After the processor is waken up, the system enters the second step.

In the second step, when the server is not completely idle, it is possible that the response time is unnecessarily shorter than the desired value at the cost of more energy consumption. In this case, it is desirable that the processor be put into Deep Sleep in short periods to allow reduced energy usage. To achieve response time guarantees, the power state of the processor is switched between Deep Sleep and the active mode on a time scale much shorter than the response time requirement of a web request based on a value called *duty cycle*, *i.e.*, the fraction of time when the processor is put into Deep Sleep. When the processor is sleeping, incoming requests are batched by the network adapter. When there are no pending requests to process, the system is switched to the first step and put into Deep Sleep again. We design a request batching controller to dynamically tune the duty cycle to control the response time of the VMs to a certain set point. The details of the request batching controller are available in Section 4.3.

Note that though different VMs sharing the server may have different workloads, they share the same duty cycle. When the processor is in the active mode, the performance balancing controller gives more CPU resource to the VMs with heavy workloads such that all VMs will have their desired performance.

The duty cycle is enforced by switching the processor between the sleep and active states on a small time scale. Based on a recent study [53], the transition time of putting a processor into the Deep Sleep mode is about 30 μs for many processors, such as the Intel Pentium M processor. For processors that do not directly support Deep Sleep, such as Xeon 5400 series, clock gating can be used instead, which can effectively reduce the processor power from 80W to 16W in nanoseconds [53]. This small overhead makes it feasible to apply request batching in real server systems. In addition, it has been demonstrated that some other components in a server, such as DRAM DIMM, can also be transitioned into the sleep mode in less than 1 μs in future server design. As a result, the Virtual Batching technique can be extended

to put those components into sleep as well, which is our future work. Note that for processors that can achieve very low power consumption at the lowest DVFS level, the Virtual Batching technique can be applied to other components in the server, such as DRAM, for energy savings.

4.2 Performance Balancing Controller

As discussed in Section 4.1, when the server is not completely idle, a performance balancing controller is designed to balance the VM-level response times relative to their response time requirements by dynamically adjusting the CPU resource allocated to the VMs. In this section, we present the problem formulation, modeling, and design of the performance balancing controller.

4.2.1 Problem Formulation

Performance balancing control can be formulated as a dynamic optimization problem. We first introduce some notation. A physical server hosts N virtual machines. T_1 is the control period. d_i is the maximum allowed response time of i^{th} virtual machine VM_i . $rt_i(k)$ is the average response time of VM_i in the k^{th} control period. $r_i(k)$ is the relative response time of VM_i . Specifically, $r_i(k) = rt_i(k)/d_i$. $r(k)$ is the average relative response time of all the virtual machines in the k^{th} control period, namely, $r(k) = \sum_{i=1}^N r_i(k)/N$. Note that different VMs may have different response times because they have different workloads and run different applications. The goal of the performance balancing controller is to balance the relative response time of all the VMs, *i.e.*, to make $r_i(k) = r(k)$.

To achieve this goal, we adjust a parameter in the virtual machine hypervisor, *weight* $w_i(k)$. The scheduler of the VM hypervisor allocates CPU time to VM_i proportionally to $w_i(k)$.

The performance balancing problem can be formulated as:

$$\min_{\{w_j(k)|1 \leq j \leq N\}} \sum_{i=1}^N (r_i(k) - r(k))^2 \quad (4.1)$$

4.2.2 System Modeling and Controller Design

The response time model, *i.e.*, the relationship between $r_i(k)$ and $w_i(k)$, is normally nonlinear due to the complexity of computer systems. Since nonlinear control can lead to unacceptable runtime overhead, a standard method used in control theory to handle such systems is to linearize the system model. Therefore, we use a linear model as:

$$r_i(k) = \sum_{j=1}^{m_i} \alpha_{ij} r_i(k-j) + \sum_{j=0}^{n_i} \beta_{ij} w_i(k-j) + \gamma_i \quad (4.2)$$

where α_{ij} , β_{ij} and γ_i are constant parameters.

We conduct extensive analysis using workloads in a wide range of intensities on the testbed introduced in detail in Section 4.5. The results show that $m_i = 1$ and $n_i = 1$ provides a reasonable compromise between model accuracy and model simplicity. Therefore, we use $m_i = 1$ and $n_i = 1$. Model (4.2) can be rewritten as the following difference equation:

$$r_i(k) = r_i(k-1) + \alpha_i(r_i(k-1) - r_i(k-2)) + \beta_i \Delta w_i(k-1) \quad (4.3)$$

where $\Delta w_i(k-1) = w_i(k) - w_i(k-1)$ and α_i and β_i are constant parameters whose nominal values can be decided based on least square method in an offline system identification experiment, as introduced in [85]. For each specific VM, the real values of α_i and β_i can be dynamically updated using a standard online model estimator [51].

Based on Proportional-Integral-Derivative (PID) control theory, we design the VM-level response time controller as a P (Proportional) controller:

$$\Delta w_i(k) = K_i(r(k) - r_i(k)) \quad (4.4)$$

where $K_i = \frac{1}{2\beta_i}$.

Note that we use a P controller instead of a more sophisticated PI controller because the actuator, *i.e.*, the weight in the VM hypervisor, includes an integration step (because $w_i(k) = \sum_{j=0}^k \Delta w_i(k)$) such that zero steady-state error can be achieved without resorting to a PI controller.

In each control period, the VM-level controller measures the relative response time of VM_i , $r_i(k)$, compares it with the average relative response time of all VMs, $r(k)$, and decides the change in *weight* based on the controller in (4.4). All of the VM-level controllers together compose the performance balancing controller which dynamically equalizes the relative response times of all the VMs.

4.2.3 Controller Analysis

In this subsection, we analyze the stability of the performance balancing controller. A fundamental benefit of the control theoretic approach is that it gives us confidence for system stability, even when the controller is used under different working conditions.

Closed-loop Transfer Function

For N VMs sharing a physical server, the N VM-level controllers are coupled because they all rely on the same average response time, which is dependent on the response time of every VM. Thus, to analyze the stability of the controller, we need to consider all the VMs together.

The closed-loop behavior of the performance balancing controller can be derived by substituting (4.4) into (4.3), so we get:

$$\begin{aligned}
r_i(k+1) &= r_i(k) + \alpha_i(r_i(k-1) - r_i(k-2)) \\
&\quad + \frac{1}{2} \left(\frac{1}{N} \sum_{j=1}^N r_j(k) - r_i(k) \right)
\end{aligned}$$

This is equivalent to:

$$X(k+1) = AX(k) \tag{4.5}$$

$$Y(k+1) = CX(k+1) \tag{4.6}$$

where:

$$X(k) = [R(k)^T, R(k-1)^T, R(k-2)^T]^T$$

$$R(k) = [r_1(k), r_2(k), \dots, r_N(k)]^T$$

$$A = \begin{bmatrix} S & P & -P \\ 0_N & S & 0_N \\ 0_N & 0_N & S \end{bmatrix}$$

with 0_N is the N by N zero matrix, and:

$$S = \begin{bmatrix} \frac{1}{2} + \frac{1}{2N} & \frac{1}{2N} & \cdots & \frac{1}{2N} \\ \frac{1}{2N} & \frac{1}{2} + \frac{1}{2N} & \cdots & \frac{1}{2N} \\ \cdots & & & \\ \frac{1}{2N} & \frac{1}{2N} & \cdots & \frac{1}{2} + \frac{1}{2N} \end{bmatrix}$$

$$P = \begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_N \\ 0 & 0 & \cdots & 0 \\ \cdots & & & \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

$$Y(k) = [r_1(k) - r_2(k), r_2(k) - r_3(k) \cdots, r_{N-1}(k) - r_N(k)]^T$$

$$C = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 1 & -1 & \cdots & 0 \end{bmatrix}$$

We introduce a parameter d_i for each VM to represent the unmodeled system disturbance and noise that may influence the response time of a VM. Thus the model becomes:

$$X(k+1) = AX(k) + BD(k) \quad (4.7)$$

$$Y(k+1) = CX(k+1) \quad (4.8)$$

where $D(k) = [d_1(k), d_2(k), \dots, d_N(k)]^T$, $B = [I_N, 0_N, 0_N]^T$ and I_N is an N by N identity matrix.

System Stability

To examine the behavior of our system, we study the stability of the closed-loop system under the modeled system disturbance $D(k)$. In control theory, Bounded-Input Bounded-Output (BIBO) Stability [68] guarantees that the output of the closed-loop system, *i.e.*, $Y(k)$, will not go unbounded if the input of the system, *i.e.*, the system disturbance $D(k)$, remains bounded.

Based on control theory, the system in (4.7) and (4.8) is BIBO stable if and only if all poles in the transfer function (in z-domain) $G(Z) = C(zI - A)^{-1}B$ have magnitude less than unity [68].

Example: We now apply the stability analysis approach to the server used in our experiment under light workload. Based on an offline analysis, we know that $\alpha_i = 0$ when the workload is light. This is because when the workload is sparse, the server queue is less likely to build up, thus the condition in the last period is less likely to

affect the response time in the current period. The poles of the transfer function are $\{0.5, 0.5\}$. Thus, the system is stable.

When the controller is applied to a real system, model variations are common, *i.e.*, the value of the parameter β_i in (4.3) may change at runtime due to changes in workloads or in the VMs. However, the parameter K_i is designed based on the system with a nominal value of β_i . Thus, it is important to evaluate the stability of the system when the controller is used with model variations. Now we analyze the stability when parameter β_i in the model changes to β_i/g_i due to the changes in workload in two different cases. In the first case, we assume that all the VMs have a uniform model variation, *i.e.*, $g_i = g, 1 \leq i \leq n$. We derive the range of g as $0 < g \leq 2$. Since the parameter β_i indicates the impact of the change in CPU allocation on the response time, the stability range means that a system controlled by the controller designed based on the estimated model will remain stable as long as impact is less than twice the estimated value we used in system design. To demonstrate the system behavior under different values of g , we perform a series of simulations in Matlab and present the values of $\|Y\|$ in Figure 4.2. $\|Y\|$ is the Euclidean norm of the output vector Y of the closed-loop system (4.7) and (4.8). Before the controllers run, the relative response times of the 3 VMs are 0.7, 0.1 and 0.6, *i.e.*, the VM-level relative response times are unbalanced. A white-noise signal is used as a disturbance to the system, *i.e.*, $D(k)$ in (4.7). For all values of g within the stability range, $\|Y\|$ converges to 0, which means that the relative response times of all VMs are balanced. In comparison, in the case when $g = 4$, $\|Y\|$ always stays above 0.5 in the 100 control periods shown in the figure.

In the second case, each VM has a different model variation, our second way of analyzing the system stability is to analyze one VM at a time. Our results show that the range of g_i is $0 < g_i \leq 5.4$. Other model variation patterns can be analyzed in a similar way. This analysis of stability range is similar to the gain margin analysis of closed-loop control systems [76]. We used a Matlab program to perform the above stability analysis procedure automatically.

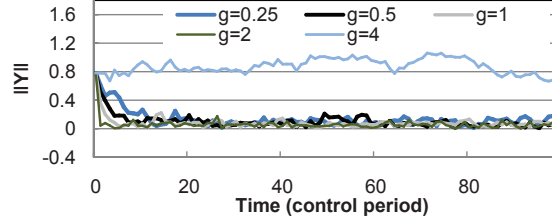


Figure 4.2: Euclidean norm of the output vector Y of the closed-loop system (4.7) and (4.8) under different levels of uniform model variations.

To handle systems with an actual g that is outside the established stability range, an online model estimator [51] can be adopted to dynamically correct the system model based on the real response time measurements. Also, more advanced adaptive control theory, such as Model Reference Adaptive Control (MRAC)[77], can be adopted such that the system stability can be guaranteed despite significant model variations.

4.3 Request Batching Controller

In this section, we present the problem formulation, modeling, and design of the request batching controller.

4.3.1 Problem Formulation

As discussed in Section 4.1, when the server is not completely idle, the request batching policy periodically puts the processor into Deep Sleep to save power while guaranteeing the desired response time.

We denote the processor duty cycle as $\lambda(k)$, *i.e.*, $0 < \lambda(k) \leq 1$. $\lambda = 1$ means that the CPU is active 100% of the time, *i.e.*, there is no sleeping time. The request batching controller controls $r(k)$, the average relative response time of all of the VMs, to a desired set point r_s by dynamically adjusting $\lambda(k)$ to put the processor into Deep Sleep to save power based on $\lambda(k)$.

4.3.2 System Modeling and Controller Design

As discussed in Section 4.1, in this dissertation we focus on controlling the server-side average response time. The server-side response time depends on the time spent by the processor in processing the web requests and the time spend on IO subsystems, etc. Therefore, we model the server-side response time as two parts: the first part depends on the active time of the processor and can be delayed when the processor is put into Deep Sleep. The second part remains relatively constant when the processor shifts between Deep Sleep and active modes. Suppose the average relative response time of the server when its duty cycle is 100% is r_0 , given a duty cycle $\lambda(k)$, the response time is modeled as:

$$r(k) = \lambda^{-1}(k)r_0 + r_{IO} \quad (4.9)$$

Note that since the request batching controller is only designed for light workloads, as discussed in Section 4.1, we assume that the queuing delay is not dominant in the response time of a web request when request batching controller is activated. Note that since the request batching controller is only designed for light workloads, as discussed in Section 4.1, we assume that the queuing delay is not dominant in the response time of a web request when request batching controller is activated. For some servers on which the queuing of workloads significantly affects the accuracy of the model in (4.9), the model can be replaced with a linearized equation inferred from system identification, the same technique we used to build our response time model in Section 4.2.

This model can be rewritten as a difference equation:

$$r(k) = r(k-1) + r_0\Delta\lambda^{-1}(k-1) \quad (4.10)$$

We design the controller as a P controller:

$$\Delta\lambda^{-1}(k) = K_{p2}(r(k) - r_s(k)) \quad (4.11)$$

where $K_{p2} = 1/r_0$.

In every control period, the controller gets the measured relative average response time $r(k)$ from the response time monitors of all the VMs. Then it decides a new duty cycle based on the controller in (4.11).

We now analyze the stability of the system. The system model (4.10) may change at runtime due to changes in workloads, the number of VMs, etc. To model the variation in the parameter r_0 , we write (4.10) as:

$$r(k) = r(k-1) + gr_0\Delta\lambda^{-1}(k-1) \quad (4.12)$$

Note that in this dissertation, we assume that the applications in different VMs are independent of each other. However, for multi-tier applications that may span multiple VMs and even multiple physical servers, the response time of each application relies on the performance of every VM that hosts the application. To provide response time guarantees to multi-tier applications, our algorithm can be extended to include an additional step that decomposes the response time requirement of a multi-tier application into a series of response time requirements for VMs that host a tier of the application. We can then apply the proposed local performance balancing controllers to balance the performance of VMs on the same physical server, and the proposed local request batching controllers to put the processor into Deep Sleep for power savings, while guaranteeing the desired the average relative response time of all of the VMs.

The controller is designed based on the model with $g = 1$. We have proven that the system remains stable when $0 < g \leq 2$. This means that a system controlled by the controller based on the estimated model will remain stable as long as the average response time of the server when the duty cycle is 100% is smaller than twice the value used to design the controller.

4.4 Integration of Virtual Batching and Server Consolidation

Modern data centers increasingly rely on server consolidation for energy conservation. In this section, we discuss how to integrate Virtual Batching with server consolidation for further energy conservation.

4.4.1 Motivation Studies

As discussed in Chapter 1, request batching works best when the utilization of the server is low. However, server consolidation aims to fully utilize a small number of active servers in the data center while keeping the rest servers shut down, which increases the utilization of the active servers and thus may reduce the benefits of request batching. However, the active servers after consolidation can still be underutilized for two reasons. First, servers cannot be consolidated too frequently due to the large overheads of VM migrations. Between two invocations of a consolidation algorithm, it is possible that the utilizations of some servers become very low due to changes in their workloads. Second, the number of VMs that can be consolidated onto a single physical server is normally limited due to various real-world constraints, such as memory requirements.

To examine server utilization when an effective consolidation algorithm is running, we test pMapper, a recently proposed server consolidation algorithm [79], in the simulator described in Section 4.5. In the simulation, we test 15 VM memory requirements from 64MB to 8GB, which are typical memory sizes for low-end VMs [81] and high-end VMs [25] sold as Virtual Private Servers (VPS) in the web hosting market in July 2010, respectively. The memory sizes of the servers are listed in Table 4.1. Figure 4.3 plots the average utilization of the servers under these VM memory size requirements. When the memory requirement increases, the maximum number of VMs that can be consolidated onto a server without violating the memory constraint

decreases. This simulation shows that the average utilization of the server is low when the memory constraint is tight. For example, the server utilization is about 33% when the average memory requirement of the VMs is 1 *GB*, which is a typical memory size for VMs hosting business applications [81]. The underutilization makes it possible to integrate request batching for further energy savings.

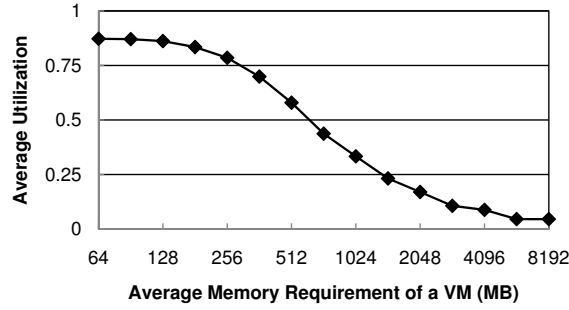


Figure 4.3: Average server CPU utilization under different VM memory requirements.

4.4.2 Integration of Virtual Batching and pMapper

We first take pMapper as an example to briefly explain how server consolidation algorithms usually work. There currently exist two kinds of consolidation algorithms: static and dynamic. Static consolidation algorithms are normally performed offline (or over a long period, *e.g.*, weeks) for best consolidation results, so the overhead is not a serious concern. In contrast, dynamic consolidation algorithms, such as pMapper, are performed periodically (*e.g.*, hours) to handle workload variations. Therefore, dynamic consolidation algorithms must have low overheads.

pMapper conducts consolidation incrementally in two phases. In the first phase, it sorts the servers based on a metric called power-efficiency and sorts the VMs based on their application sizes (*e.g.*, CPU resource demands). It then computes the ideal consolidation result by trying to fit the VMs into the servers using a first-fit algorithm, beginning with the most power-efficient server and largest VM. Note that

in this phase, the VMs are not actually migrated because it would lead to significant overheads to achieve the ideal result by migrating all the VMs around. The purpose of this step is to compute the ideal utilization each server would have in such an ideal consolidation result. The resultant ideal utilizations are used in the next phase to incrementally migrate servers.

In the second phase, pMapper compares the ideal utilization and current real utilization for each server and computes the list of servers that require a higher utilization in the ideal consolidation result. Those server are called (utilization) receivers because they need to receive utilization from other servers in order to approximate the ideal consolidation result computed in the first phase. Similarly, those servers with the ideal utilization lower than the current utilization are called (utilization) donors. pMapper selects the VMs with smallest-sized applications from the donors and adds them to a VM migration list. It then runs first-fit decreasing (FFD) to migrate the VMs in the migration list to the receivers. In this way, pMapper can achieve a reasonably good consolidation result with acceptable overheads.

Since the focus of this section is to demonstrate that the integration of Virtual Batching and pMapper can lead to more energy conservation than pMapper alone, we adopt a simple but effective approach to integrate the two strategies. We then discuss possible ways to improve this integration as our future work. In this simple approach, we first use pMapper directly for server consolidation, as we described above. After pMapper decides which servers should stay active, we then apply Virtual Batching only to the active servers to put their processors periodically into Deep Sleep for further energy conservation. As demonstrated in our experiments in Section 4.6.5, this simple approach is effective and can lead to about 41% to 56% more power savings than pMapper alone.

There certainly can be more sophisticated ways to integrate Virtual Batching and pMapper at the cost of higher computation overheads. One example way is to redefine the power-efficiency metric used by consolidation algorithms, because those algorithms commonly move VMs from servers with lower power-efficiency to

servers with higher power-efficiency for reduced server power consumption. As a result, servers with higher power-efficiency are used first and thus more likely to stay active during the consolidation process. Power-efficiency is typically defined by many consolidation algorithms (*e.g.*, [79][80]) as the ratio of the peak power consumed by the server to the capacity of the server. The server capacity can be defined as some benchmark value (*e.g.*, RPE2 metric from IDEAS [34]) or the CPU resource provided by the server (in terms of CPU frequency). Therefore, a common definition of power-efficiency is:

$$E = \frac{P_{max}}{F_{max}} \quad (4.13)$$

where P_{max} is the peak power consumption of the server and F_{max} is the highest CPU frequency of the server. Since the same CPU frequency on different servers can lead to different application performance, to make power-efficiency a comparable metric among the servers, the values of F_{max} for all the servers are usually adjusted such that a VM will consume the same amount of CPU resource on different servers [80].

When considering Virtual Batching, this common definition of power-efficiency can lead to energy waste because it neglects the processor sleep power. For example, when two servers have the same peak power and the same CPU frequency but significantly different processor sleep power, the definition in (4.13) may indicate that the two servers are equally power-efficient. However, it is clear that the one with the lower processor sleep power is more power-efficient because it consumes less power when its processor is put into Deep Sleep by Virtual Batching. To better integrate Virtual Batching with pMapper, we can redefine power-efficiency in (4.13) to consider the power benefits brought by Virtual Batching. We now consider the power consumption of an active server that is managed by Virtual Batching as:

$$p_{vb} = \lambda p_{awake} + (1 - \lambda) p_{sleep} \quad (4.14)$$

where λ is the average duty cycle of the server in the next period of the consolidation algorithm. When the server is managed by Virtual Batching, we have the CPU

resource usage $c = F_{max}\lambda$. We can now redefine power-efficiency with Virtual Batching by using p_{vb} and $c = F_{max}\lambda$ to replace P_{max} and F_{max} in (4.13) as $E_{vb} = p_{vb}/c$. Unfortunately, although this new definition is reasonable, the value of λ is hard to determine ahead of time because it depends on which VMs the server will host after the consolidation. Therefore, finding an optimal or near-optimal solution may need rounds of iterations and the complexity is non-trivial. Since the focus of this dissertation is to present a request batching solution designed for virtualized servers, a more sophisticated integration strategy based on advanced optimization techniques is beyond the scope of this dissertation. We plan to address this in our future work. As our simple integration method already achieves significant power savings, we believe that a more sophisticated strategy can lead to even more power savings.

4.5 System Implementation

4.5.1 Testbed

Our testbed includes a cluster of six physical servers named S1 to S6 with the configurations detailed in Table 4.1. Linux 2.6.18 kernel is used for the privileged/administrative domain (dom0) and three VMs (domU) on each server. All the servers are connected via a gigabyte Ethernet switch. In each VM, we install Apache 2.2.3 to serve a dynamic web page via PHP as a web application. The applications in the VMs are independent of each other. The servers are configured such that the VMs can be moved between servers via VM migration based on the server consolidation algorithm.

The performance balancing controller, request batching controller, and CPU frequency modulator are configured to run as daemons in dom0 along with a response time monitor. The two controllers are invoked periodically and the period is selected to be 3 seconds such that multiple HTTP requests can be received during a control

Table 4.1: Models and parameters of the six real servers used in the testbed.

Server	Processor	Chips	Cores/Chip	Memory	Platform
S1	AMD Opteron 2222 SE	1	2	4Gb	Xen 3.1
S2	Intel Xeon X5160	1	2	4Gb	Xen 3.3
S3	AMD Opteron 2222 SE	1	2	4Gb	Xen 3.3
S4	Intel Xeon X5160	1	2	4Gb	Xen 3.3
S5	Intel Xeon X5365	1	4	4Gb	Xen 3.3
S6	AMD Opteron 2222 SE	2	6	16Gb	Xen 4.0

period and the actuation overhead is acceptable. The response time monitor accepts incoming connections for all the web servers. When a request is received, the following steps are conducted. First, a time stamp is recorded. Second, the HTTP request is passed to the appropriate virtual machine. Third, when the HTTP server in the virtual machine generates a response, a second time stamp is taken. Fourth, the response generated by the HTTP server in the virtual machine is returned to the requesting client. Finally, the difference between the timestamps generated in step one and step three is taken to calculate the response time for that particular request.

The response time monitor is also capable of replaying the web requests based on a trace file or a concurrency level. When it generates workloads based on a trace file, it sends requests based on the time stamps of recorded activities in the trace file. When it generates workloads based on a concurrency level, it dynamically issues requests to make the number of requests being processed in the server the same as the concurrency level, *i.e.*, the workload is continuous. In this dissertation, we test our algorithms using workloads generated in both ways.

In our experiments, since Linux has incompatibility issues with the ACPI implementation of the Dell servers deployed in our testbed, we use emulation to place the processor into Deep Sleep. Specifically, when the processor needs to be put into sleep, the request batching controller issues a SIGSTOP to all the Apache processes on each virtual machine, waits for the appropriate time based on the current duty cycle, then issues a SIGCONT to all the Apache processes. Note that the transition overheads for the server to sleep and wake up have been accounted by deducting the typical overhead ($30 \mu s$ per transition [53]) from the desired sleep interval in our

energy measurements. Therefore, our emulation results can provide accurate insights for the performance evaluations.

To record the energy use of the testbed, power and energy measurements are taken with a *Wattsup PRO* meter. For experiments involving request batching, logs are kept of the amount of time the processor would have spent in Deep Sleep allowing net energy consumption to be approximated using $e_{net} = e_{gross} - t_{sleep} * (p_{awake} - p_{sleep})$, where e_{net} is the approximate net energy usage, e_{gross} is the measured energy usage, t_{sleep} is the total amount of time the processor would have spent in Deep Sleep, p_{awake} is the average power used by the server during the waking periods, and p_{sleep} is the estimated power usage of the server when the processor would be in Deep Sleep.

In every control period, the request batching controller decides a new duty cycle based on the controller equation in (4.11). The system then enforces the duty cycle by switching the processor between the sleep and active states on a small time scale denoted as a subinterval T_s . We use a modulator to approximate the duty cycle. For example, to enforce a duty cycle of 0.25, the modulator will place the processor into Deep Sleep for $3T_s$ and the active state for T_s .

The length of T_s needs to be carefully selected. A longer T_s means that the processor will be kept in Deep Sleep for a longer period of time, thus increases the probability that a request arriving during the sleep interval may fail to meet the response time requirement. A shorter T_s means that the processor may be switched between the Deep Sleep and active modes more frequently, thus incurs a higher overhead. Furthermore, the resolution of the timer in the operating system limits the length of T_s . We select the length of T_s to be $\min\{100T_o, T_{timer}\}$, where T_o is the time spent to switch the processor between the Deep Sleep and active states and T_{timer} is the resolution of the timer in the OS, such that the overhead of enforcing the duty cycle is less than 1%.

4.5.2 Simulation Environment

To evaluate the integration of Virtual Batching and server consolidation in large-scale data centers, we have developed a simulator that uses a trace file from real-world data centers [84] to simulate the CPU utilization variations. The trace file includes the utilization data of 5,415 servers from ten large companies covering the manufacturing, telecommunications, financial, and retail sectors. The trace file records the average CPU utilization of each server every 15 minutes for about a week in July 2008. We treat the utilization data of each server as the CPU demand of a VM. The simulated servers are randomly generated from the 5 server models listed in Table 4.2. The active power consumption values of the servers are obtained from [75]. In order to estimate the server power when Virtual Batching puts the processors of the servers into sleep, we conservatively use the maximum server power from [75] minus the power difference of the processors in the active and sleep states. The power values of the processors in different states are obtained from [4].

Table 4.2: Server models and parameters used in large-scale simulations.

Type	Active Power		Sleep	Processor			RAM
	Max	Min		Max Freq.	Min Freq.	Cores	
Transport GT24 (B2912-E)	264W	121W	51W	2700MHz	800MHz	2X4	16G
Supermicro 2021M-UR+	269W	138W	56W	2500MHz	800MHz	2X4	16G
HP ProLiant DL385 G5p	257W	147W	44W	2700MHz	800MHz	2X4	8G
Dell PowerEdge 2970	302W	139W	86.4W	2300MHz	1150MHz	2X4	16G
HP ProLiant DL385 G6	258W	120W	47.6W	2600MHz	800MHz	2X6	16G

In our simulation, we assume that the applications in the VMs are independent of each other. Each loop of the simulation represents a 15-minutes interval of the data center. When the simulator is initialized, the VMs are assigned to the servers randomly. In each simulation loop, the simulator reads a new line from the trace file as the CPU demand of the VMs and calculates the power consumption of each server based on the CPU utilization of the VMs hosted on it. Then, the simulator runs the server consolidation algorithm presented in Section V subject to the memory constraint of the servers and moves the VMs between the servers when necessary.

When all lines in the trace file are used, the simulator calculates the total energy consumption of all the servers in the data center.

4.6 Evaluation

The testbed described in Section 4.5 is used in four experiments to show the effectiveness of our solution. We use server S1 in the first three experiments and all the six servers in the last experiment. The first experiment examines the performance balancing and request batching controllers. In the second experiment, we show the effectiveness of integrating DVFS and request batching to handle both light and moderate workloads. The third experiment compares Virtual Batching against three baselines with bursty workloads. In the last experiment, we test the integration of Virtual Batching and the pMapper consolidation algorithm.

4.6.1 Baselines

The first baseline, called Conservative, is a simple extension of the request batching policy for non-virtualized servers used in [22] to allow it to function in a virtualized environment. The Conservative policy places the processor into Deep Sleep whenever there is no workload for the entire physical server *i.e.*, no virtual machines are processing requests. Once new incoming traffic is detected for any web server on a network interface, the system waits for a batching timeout then wakes up the CPU to process requests. The batching timeout is determined periodically by an ad-hoc controller to drive the web server with the longest response time (no performance balancing is used) to the set point. The ad-hoc controller simply increments the batching timeout by 10ms if the response time is at or below the set point and decrements the batching timeout by 10ms if the batching timeout is above the set point. Using this method, all the virtual machines can operate at or below the set point.

The second baseline, called Aggressive, uses a request batching controller for each virtual machine to determine batching timeouts and when it is safe for the controller’s virtual machine to be put into sleep. A master controller is used to place the processor into Deep Sleep when more than half of the virtual machine controllers determine it is safe to sleep. When new requests are detected after the master controller has placed the processor in Deep Sleep, the system waits for the batching timeout associated with the virtual machine that is handling the requests, then wakes up the CPU to respond to the requests.

The third baseline, called DVFS-Only, only runs the DVFS-based method in Chapter 3 to save energy without putting the processor into Deep Sleep.

4.6.2 Request Batching

In this experiment, the web server in each virtual machine is subjected to a light and constant workload to demonstrate the functionality of the performance balancing and request batching controllers. The controllers are activated at time 750s. Figure 4.4(a) shows that the *relative* response times, defined in Section 4.1, of all the three virtual machines are controlled to be approximately equal (all three Web servers had an equal relative response time goal) after the controller is activated at time 750s. To balance the web server response times, the performance balancing controller manipulates the scheduling weight of each VM, as shown in Figure 4.4(b). Here the scheduling weight is based on Xen’s credit scheduler [2]. Xen allots CPU time to each virtual machine based on this weight; a VM with a weight of 512 will get twice as much CPU time as a VM with a weight of 256. The performance balancing controller calculates a new weight for each VM in every control period and the new weights are given to the credit scheduler for enforcement.

The request batching controller is used to control the response time of the web servers in the virtual machines to the desired set point. Figure 4.4(c) plots the average

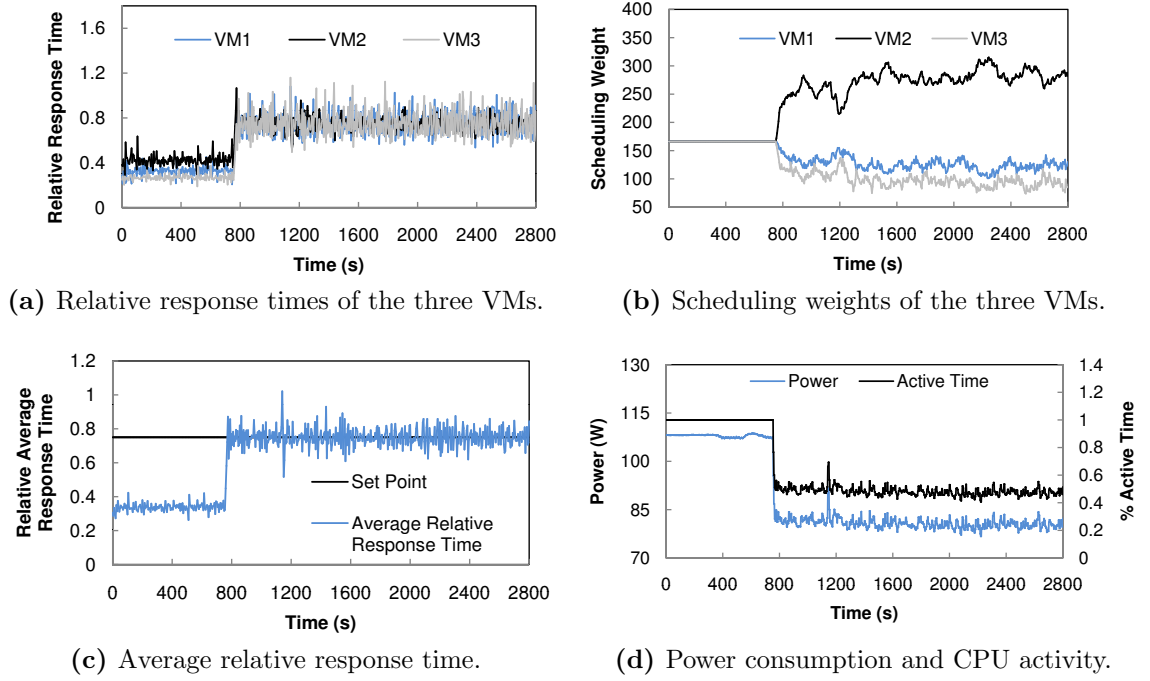


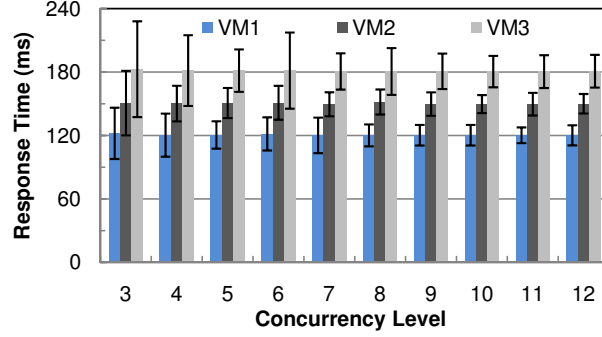
Figure 4.4: A typical run of Virtual Batching. The controllers are activated at time 750s.

of the relative response times shown in Figure 4.4(a) along with the administrator-defined set point of 0.75. This figure shows that the average response time has been controlled to the set point after time 750s. Figure 4.4(d) shows the power consumption and the percentage of time the processor would spend in Deep Sleep during each control period. These results show that Virtual Batching can effectively divide the resources of a single physical server among multiple virtual machines and control the response times of the web servers in each virtual machine to the desired set point.

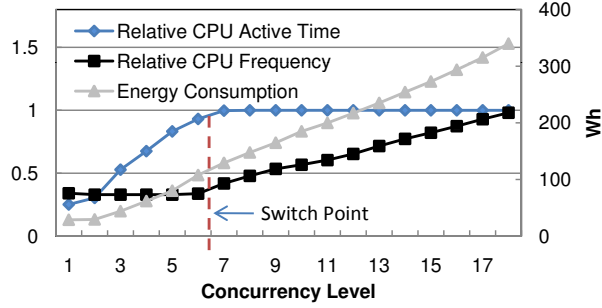
4.6.3 Integration of Request Batching and DVFS

The previous experiment shows that Virtual Batching is capable of controlling each virtual machine to a specified set point. This experiment subjects the server to workloads ranging from light to heavy by manipulating the number of concurrent requests addressed to each web server throughout the experiment.

Figure 4.5(a) shows the relative response time of each web server for each virtual machine across different concurrency levels. The response times are controlled very closely to their set points. Note that in this experiment, each virtual machine has a different set point; VM1's set point is 120ms, VM2's set point is 150ms, and VM3's set point is 180ms.



(a) VM response times under different workload intensities.



(b) CPU frequency and system energy consumption under different workload intensities. The dashed line shows the switch point between request batching and DVFS.

Figure 4.5: Integration of request batching and DVFS.

Figure 4.5(b) plots the percentage of time the processor is active and its relative frequency under different levels of concurrent requests, as well as the energy consumption on the secondary axis. As expected, the processor spends very little time in the active state when the concurrency level is one to three: 25% to 50% respectively, and spends a larger percentage of time being active after the concurrency

level increases to six. When the concurrency level is smaller than seven, the request batching controller is used to manage response time. When the concurrency level reaches seven, the CPU is active 100% of the time, which shows that the system has automatically switched to the DVFS-based controller to manage the response time. The switch point is denoted as the dashed line in Figure 4.5(b). For concurrency levels one through six, the processor is at its lowest frequency. This is the range for which the request batching controller is activated. After the concurrency levels increase to seven, the DVFS-based controller begins to manage response time and increases the CPU frequency to keep the response times of the web servers at their desired set points.

The energy consumption of the system increases with the concurrency level from a minimum of 28.9Wh at concurrency level one to a maximum of 339.8Wh at concurrency level eighteen. The points of note for energy usage, however, are for concurrency levels one through six; for these concurrency levels, the energy consumption is markedly lower than what could be achieved with DVFS alone. At concurrency level seven, the energy consumption is 129.2Wh which is the minimum energy usage available for a DVFS only solution; compared to the minimum energy consumption of 28.9Wh, request batching provides an additional energy savings of 77.6%. This set of experiments demonstrates that request batching is an effective way for further energy conservation when the workload is light, a situation commonly observed in today's data centers.

Figure 4.6 shows the average of the response times of all the virtual machines in a typical run, in which the controller switches from request batching to DVFS. From time zero to 300s the workload is very low (at the concurrency level of 3 in this example). At time 300s the workload is increased heavily from the concurrency level of 3 to 23 for VM2, leading to the spike shown in the figure. Once the increase in workload is detected by the system, it manipulates the duty cycle to give more processor time to the VMs until the duty cycle reaches 1, *i.e.*, the processor is active

100% of the time. After that, the request batching controller is switched to the DVFS-based controller to increase the CPU frequency to drive the response time back to the set point. Despite the controller switch, the average response time has been controlled to the desired set point most of the time.

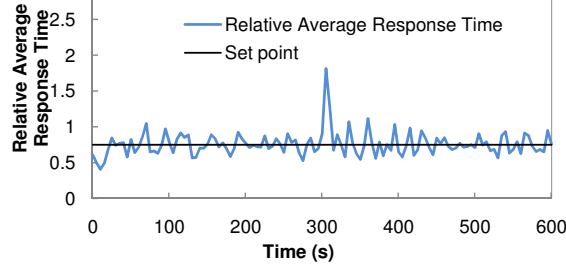


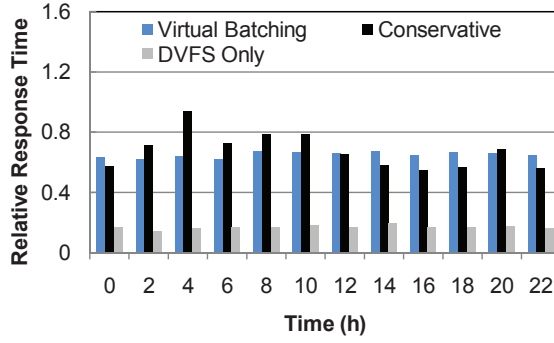
Figure 4.6: Request batching switching to DVFS at time 300s during a typical run.

These results show that Virtual Batching is able to use a combination of request batching and DVFS to control the response time of web servers in the virtual machines under different workload intensities.

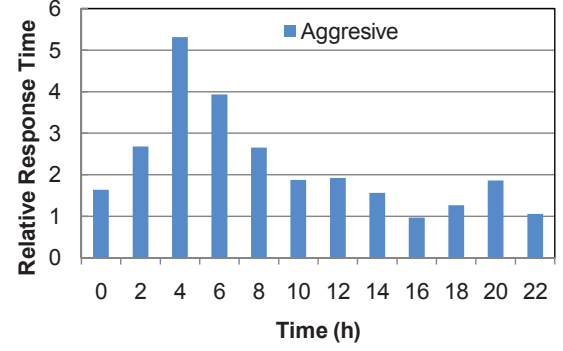
4.6.4 Comparison with Baselines for Bursty Workloads

In this experiment, we test our system using trace files from several real-world servers [3]. We use three trace files SDSC-HTTP, EPA-HTTP and ClarkNet-HTTP from three different HTTP servers to generate workloads in an entire day (24 hours starting from 00:00 AM) for the three VMs running on our server, and compare our solution to three baselines, Conservative, Aggressive, and DVFS-Only. The characteristics of the trace files are available in [3] and the total number of requests in the trace files are presented in Figure 4.7(f).

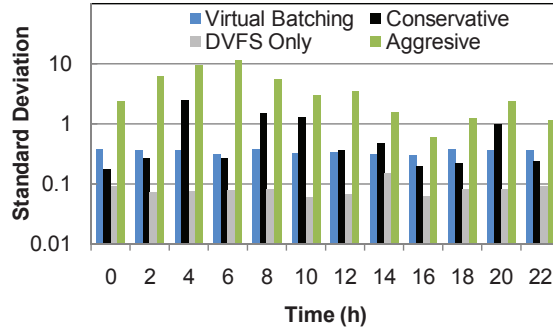
Figure 4.7(a) plots the average relative response time, with a set point of 0.8, every 2 hours in the execution. The response time of Conservative is the closest to that of Virtual Batching due to its ability to batch requests, albeit with an ad-hoc controller. DVFS-Only has a substantially lower relative response time because it does not batch requests. Conservative sometimes has longer average response times



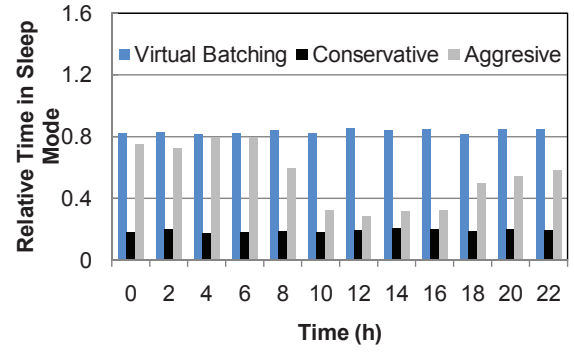
(a) Average relative response time



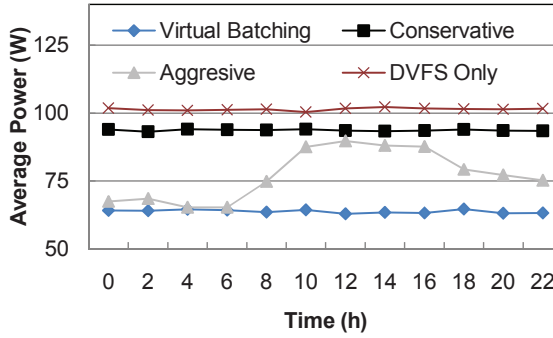
(b) Average relative response time - Aggressive



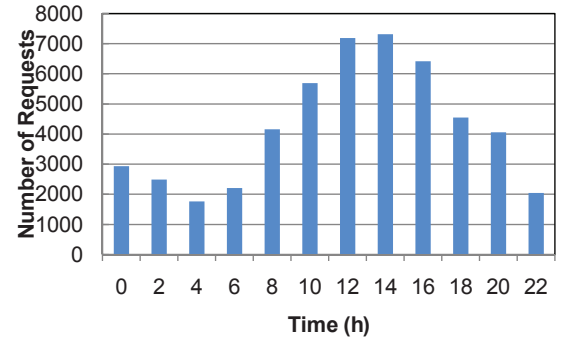
(c) Standard deviations of the relative response times plotted on logarithmic scale



(d) Time in Deep Sleep



(e) Power Usage



(f) Number of Requests

Figure 4.7: Comparison with baselines for bursty workloads based on a real 24-hour trace file.

than Virtual Batching because it cannot adjust CPU resource allocations between VMs as it does not have a performance balancing controller. When the VMs face different workload patterns in different time periods of a day, some VMs may have very long response times, making the average response time long. Figure 4.7(c) plots the standard deviations of the relative response times on logarithmic scale.

The response time of Aggressive is shown in Figure 4.7(b) because it is significantly longer than those of other solutions. Aggressive strives to minimize power consumption, but does so at the expense of response time violation. At four hours into the experiment, the relative response time peaks at 5.3. With a set point of 0.8, this is a huge performance violation and would be unacceptable in a real-world application.

Figure 4.7(d) displays the relative amount of time the processor spends in Deep Sleep for Virtual Batching, Conservative, and Aggressive. Virtual Batching remains active for approximately 20% throughout the day while Aggressive starts high and keeps the processor busier throughout the morning and afternoon then backs off as traffic decreases for the evening. This shows that Aggressive is not as efficient at handling the workload fluctuations. Under Conservative, the processor is active approximately 80% throughout the day, which leads to a higher power consumption than Virtual Batching.

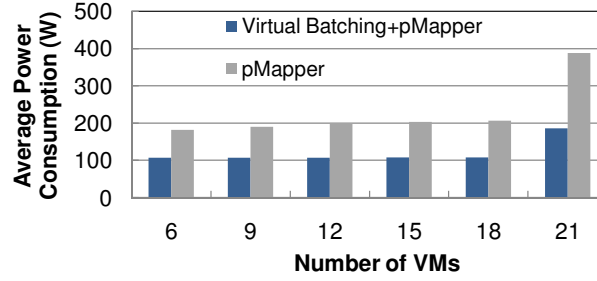
Figure 4.7(e) shows the average power comparison. Aggressive is the closest contender to Virtual Batching with regard to power usage, but does so at the expense of response time and shows excessive response time violations shown in Figure 4.7(b). Conservative uses considerably more power than Virtual Batching, approximately 93.7W on average which translates to a large amount of energy waste. This is expected because Conservative focuses on managing response time over energy usage. DVFS-Only shows the highest power consumption of all, averaging 101.4W. This is because once the processor is at its lowest DVFS level, no more power can be saved by DVFS-Only. Virtual Batching achieves an average power saving of 63% over DVFS-Only which translates into real-world energy conservation.

4.6.5 Integration with Server Consolidation

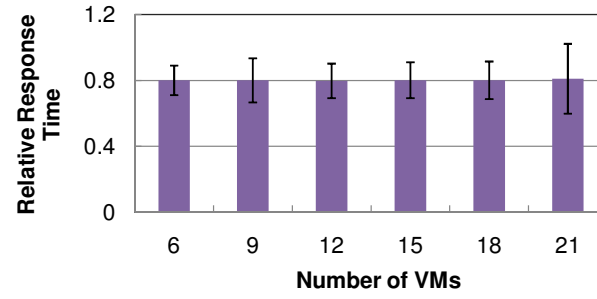
In this experiment, we evaluate the integration of Virtual Batching with an example server consolidation algorithm, pMapper (introduced in Section 5). Virtual Batching can be integrated with other consolidation algorithms in a similar way.

We first test the integrated solution on our testbed composed of 6 physical servers. Figure 4.8(a) shows the average power consumption of the cluster under two management strategies when the number of hosted VMs varies from 6 to 21. The first strategy is pMapper [79] alone while the second strategy integrates pMapper with Virtual Batching using the technique described in Section 4.4. In the experiments, when the number of VMs hosted in the cluster changes from 6 to 18, both strategies consolidate all the VMs onto a single server, S6, which is the most power-efficient one in the cluster. When the number of VMs increases to 21, both strategies start to use a second server, S1, to host the increased VMs because S6 can no longer host more VMs due to the memory constraints. The integrated solution has lower power consumption than pMapper alone, because the integrated solution periodically puts the processors of the active servers (after consolidation) into sleep for further power conservation. The additional power savings range from 41% to 52%. On the performance side, the integrated solution has average response times controlled to the set point, as shown in Figure 4.8(b), due to the response time controller provided by the Virtual Batching, while pMapper has unnecessarily short response times (shown in Figure 4.8(c)), which come at the cost of excessive power consumption shown in Figure 4.8(a).

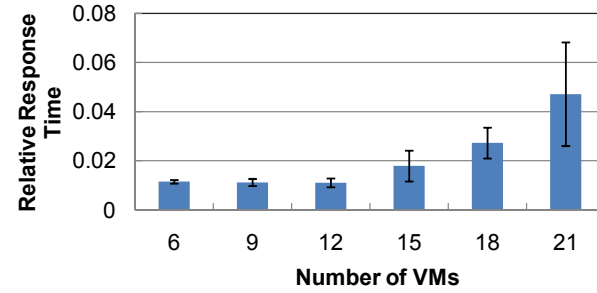
To stress test the integrated solution in large-scale data centers, we perform extensive simulations in the trace-driven simulation environment introduced in Section 4.5. We simulate a data center with 5,415 VMs. Since the trace file records the utilization variations of all the VMs in a period of one week, we conduct pMapper every two hours to dynamically consolidate the VMs to simulated servers that are randomly generated based on the parameters listed in Table 4.2. Throughout the simulation, pMapper consolidates the VMs to 545.51 servers on average. For the



(a) Comparison of cluster power consumption



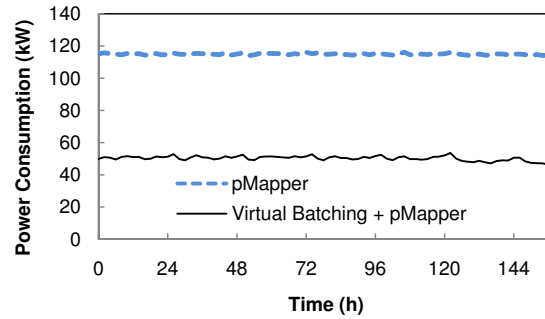
(b) Performance of the integrated solution (Virtual Batching + pMapper)



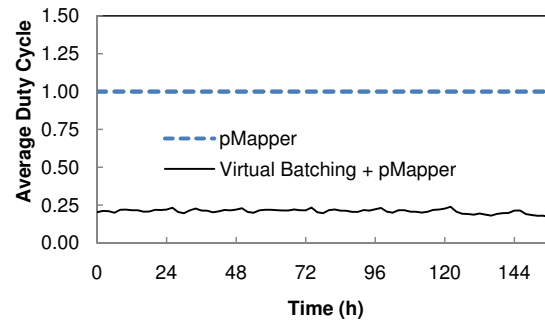
(c) Performance of pMapper alone

Figure 4.8: Power and performance comparison between the integrated solution and pMapper on the testbed.

integrated solution, we then apply Virtual Batching to the active servers determined by pMapper for further power savings. Figure 4.9(a) shows the power consumption of the integrated solution and pMapper in the one-week period. Every point in Figure 4.9(a) represents the average power consumption of the data center in every 2 hours of the week (*i.e.*, 84 sampling points in total). On average, the integrated solution shows 56.2% less power consumption than pMapper alone. Figure 4.9(b) plots the average duty cycle of the active servers in the data center. The average duty cycle of the servers under the integrated solution is 21.14%. In contrast, the duty cycle of the servers under pMapper alone is always 100% because pMapper does not put the processors of active servers into sleep. The experiments demonstrate that the integration of Virtual Batching and server consolidation can lead to considerably more energy conservation than server consolidation alone.



(a) Total power consumption of the servers in the data center



(b) Average duty cycle of the active servers in the data center

Figure 4.9: Comparison between the integrated solution and pMapper in a simulated large-scale data center with 5,415 VMs from a real trace file.

4.7 Summary

In order to save more energy (than what DVFS can) for servers with light workloads, request batching can be conducted to group received requests into batches and putting the processor into Deep Sleep between the batches. However, it is challenging to perform request batching on a virtualized server because the performance of all the virtual machines on a physical server is correlated due to sharing the same hardware platform. In this dissertation, we have proposed Virtual Batching, a novel request batching solution for virtualized web servers with primarily light workloads. Virtual Batching dynamically allocates the CPU resource such that all the virtual machines can have approximately the same performance level relative to their allowed peak values. Based on the uniform level, Virtual Batching then determines the time length for periodically batching incoming requests and putting the processor into Deep Sleep. When the workload intensity changes from light to medium, request batching is automatically switched to DVFS to increase processor frequency for performance guarantees. Empirical results based on a hardware testbed and real trace files show that Virtual Batching can achieve the desired performance with more energy conservation than several well-designed baselines, e.g., 63% more, on average, than a solution based on DVFS only. In addition, Virtual Batching is also extended to integrate with server consolidation to achieve maximized energy conservation with performance guarantees for virtualized data centers. Empirical and simulation results demonstrate that the integrated solution provides 41% to 56% additional power savings than server consolidation alone.

Chapter 5

A Datacenter-Level Algorithm Based on DVFS and Server Consolidation

Though both the DVFS-based approach in Chapter 3 and the Virtual Batching in Chapter 4 effectively reduce the dynamic power of the system, they cannot minimize the system leakage power for maximized power savings. For example, when the Virtual Batching places the processor into sleep mode, system components like the memory, hard disk and motherboard still consume a large amount of power.

Virtualization technologies can consolidate applications previously running on multiple physical servers onto a smaller number of physical servers, effectively reducing the power consumption of a server cluster by shutting down unused servers. More importantly, live migration [16] allows the movement of a virtual machine (VM) from one physical host to another with a reasonably short downtime [31]. This function makes it possible to use server consolidation as an online management approach, *i.e.*, having run-time estimation of resource requirements of every VM and dynamically re-mapping VMs to physical servers using live migration. When a more power-efficient VM-server mapping is found, unused servers can be put into the

sleep mode for reduced power consumption. A recent study [57] shows that server consolidation can achieve a power saving of 34%, which is significantly more than that achieved by DVFS (17%). However, the down side of consolidation is that it migrates VMs from one host to another, which may incur a high overhead in terms of both time and server resources.

This chapter proposes an integrated management solution to minimize power consumption for virtualized server clusters while providing application-level performance assurance. Each application-level performance controller adopts a MIMO control strategy to maintain the desired performance and reduce power consumption through DVFS and dynamic CPU resource reallocation. The cluster-level power optimizer then consolidates the VMs onto the most power-efficient servers and places unused servers into the sleep mode for power savings on a much longer time scale, to amortize the migration overhead.

5.1 System Architecture

In this section, we provide a high-level description of our system architecture, which includes an application-level response time controller and a cluster-level power optimizer.

As shown in Figure 5.1, our power management solution includes two levels. At the application level, for every application running in the cluster, there is a performance controller that dynamically controls the performance of the application by adjusting the CPU resource (*i.e.*, fraction of CPU time) allocated to the virtual machines running the application. We choose to control the 90-percentile response time of each multi-tier web application as an example SLA metric, but our management solution can be extended to control other SLAs such as average or maximum response times. We assume that the response time of a web application is independent from that of another web application. This is usually a reasonable assumption because they may belong to different customers. Hence, we choose to have a response time controller for

each multi-tier application. A server-level CPU resource arbitrator then collects the CPU resource demands of all VMs hosted on the server, allocates the CPU resource to the VMs, and uses DVFS to save power, if the server has more CPU resources than the VMs require.

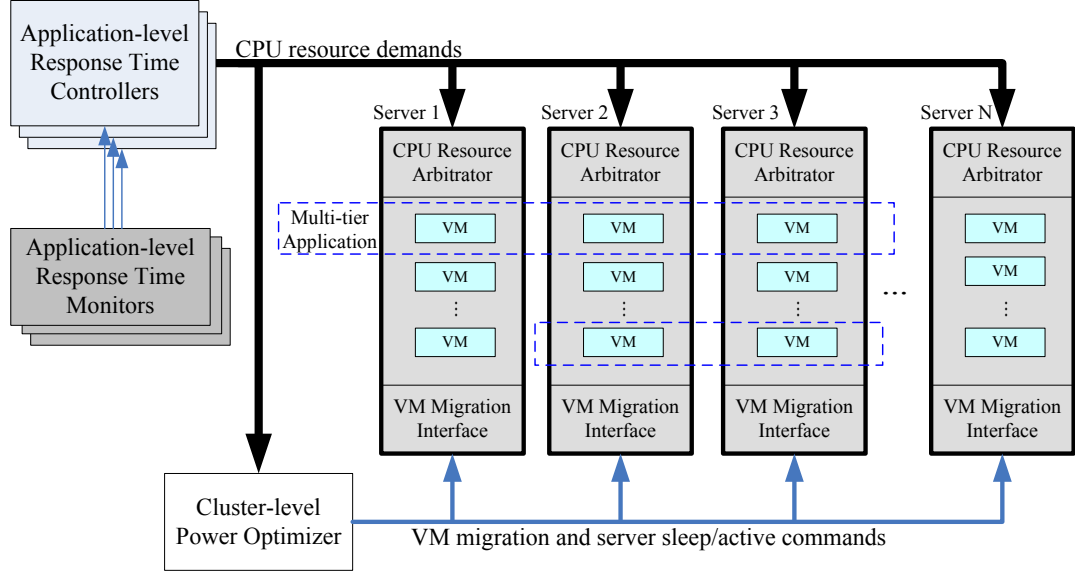


Figure 5.1: System architecture, including a response time controller for every multi-tier application and a cluster-level power optimizer.

At the cluster level, a power optimizer collects the CPU resource demands of all the VMs running in the cluster, then uses a server consolidation algorithm to find the most power-efficient VM-server mapping while satisfying the CPU resource requirements of all the VMs. The power optimizer then sends VM migration commands to the VM migration interfaces on every server, if necessary. It also puts selected servers into the sleep/active mode.

Server consolidation makes it possible to put unused servers into the sleep mode, which can typically save more power than DVFS. However, server consolidation may incur a higher overhead. Due to variations in workload, the power optimizer may require the migration of a VM from one server to another, or require a server to awaken. These operations are time consuming, especially when the processors of

the related servers are busy, or when the network bandwidth is limited. Thus, the optimizer should not be invoked too frequently. On the other hand, the response time controller at the application level uses CPU resource allocation and DVFS as its actuators, both of which have much smaller overheads than VM migration. As a result, the response time controller is invoked on a small time scale (several seconds) to deal with short-term variations in workload, while the power optimizer is invoked on a longer time scale (hours to days).

Between two consecutive invocations of the cluster-level optimizer, it is possible that an unexpected increase of the workload can cause a severe overload on a server. To deal with this problem, the solution in this chapter can be integrated with algorithms to move VMs from the overloaded servers to idle servers in an on-demand manner.

We introduce the design and analysis of the response time controller and cluster-level power optimizer in the next two sections, respectively. The implementation details of other components are provided in Section 5.4.

5.2 MIMO Response Time Controller

In this section, we present the problem formulation, modeling, and design of the MIMO response time controller designed for multi-tier applications.

5.2.1 Problem Formulation

Response time control can be formulated as a dynamic optimization problem. We first introduce some notation. A cluster with N physical servers ($S_1, S_2, \dots, S_N$) hosts L applications ($App_1, App_2, \dots, App_L$). Some applications can be multi-tier applications running in multiple VMs. A VM running the j^{th} tier of App_i is named as VM_{ij} , $j = 1, 2, \dots, r_i$. CPU allocation of VM_{ij} is defined as c_{ij} . Note that this number is expressed as the absolute number of CPU cycles allocated to VM_{ij} in

terms of GHz. For example, If VM_{11} is allocated 20% of a 5GHz CPU, we say $c_{11} = 20\% \times 5GHz = 1GHz$. $\mathbf{c}_i = [c_{i1}, c_{i2}, \dots, c_{ir_i}]^T$ is a column vector of the CPU allocations of all the VMs running application App_i . The 90-percentile response time of App_i is t_i seconds. The absolute CPU frequency of server S_i is f_i .

The control period of the response time controller is T seconds. Note that $x(k)$ denotes the value of x in the k^{th} control period, *e.g.*, $\mathbf{c}_i(k)$ means the value of $\mathbf{c}_i$ at time kT . Specially, $\Delta\mathbf{c}_i(k)$ is the difference between $\mathbf{c}_i(k+1)$ and $\mathbf{c}_i(k)$, *i.e.*, $\Delta\mathbf{c}_i(k) = \mathbf{c}_i(k+1) - \mathbf{c}_i(k)$.

At the end of every control period, the response time controller decides $\Delta\mathbf{c}_i(k)$ to achieve the desired response time for application App_i .

We assume that the constrained optimization problem is feasible, *i.e.*, there exists a set of CPU resource allocations within their acceptable ranges that can make the response time of the application achieve the desired value. If the problem is infeasible, *e.g.*, the application is highly I/O-intensive, no controller can guarantee the set points through CPU resource adaptation. In that case, the response time controller needs to be integrated with other resource allocation techniques, such as I/O scheduling, to achieve the desired response time, which is our future work.

5.2.2 Response Time Controller Design

In order to have an effective controller design, it is important to model the dynamics of the controlled system. The response time model of the application App_i is the relationship between t_i and $\mathbf{c}_i$. A well-established theoretical equation is usually unavailable for computer systems, therefore, we use a standard approach to this problem called *system identification* [24]. Rather than building a physical equation between the manipulated variables and the controlled variable, we infer their relationship by collecting data in experiments and then establish a statistical model based on the measured data. For example, by conducting system identification

to an application (App_1) in the prototype system introduced in Section 5.4, we get the system model as:

$$t_1(k) = \alpha_{11}t_1(k-1) + \beta_{11}\mathbf{c}_1(k-1) + \beta_{12}\mathbf{c}_1(k-2) + \gamma_1(k-1) \quad (5.1)$$

where α_{11} is a constant parameter while β_{11} and β_{12} are constant parameter vectors. The values of α_{11} , β_{11} and β_{12} can be determined in system identification.

We apply Model Predictive Control (MPC) theory [52] to design the controller, based on system model (5.1). MPC is an advanced control technique that deals with coupled MIMO control problems. This characteristic makes MPC well suited for response time control in multi-tier web applications.

A model predictive controller optimizes a cost function defined over a time interval in the future. The controller uses the system model to predict the control behavior over P control periods, called the prediction horizon. The control objective is to select an input trajectory that minimizes the cost function. An input trajectory includes the control inputs in the following M control periods, $\Delta\mathbf{c}(k)$, $\Delta\mathbf{c}(k+1|k)$, . . . $\Delta\mathbf{c}(k+M-1|k)$, where M is called the control horizon. The notation $x(k+i|k)$ means that the value of variable x at time $(k+i)T$ depends on the conditions at time kT . Once the input trajectory is computed, only the first element $\Delta\mathbf{c}(k)$ is applied as the control input to the system. At the end of the next control period, the prediction horizon slides one control period and the input is computed again based on feedback $t(k)$ from the response time monitor. Note that it is important to re-compute the control input because the original prediction may be incorrect due to uncertainties and inaccuracies in the system model used by the controller. MPC enables us to combine performance prediction, optimization, constraint satisfaction, and feedback control into a single algorithm. The controller includes a least squares solver, a cost function, a reference trajectory, and a system model. At the end of every control

period, the controller computes the control input $\Delta \mathbf{c}(k)$ that minimizes the following cost function:

$$J(k) = \sum_{i=1}^P \|t(k+i|k) - ref(k+i|k)\|_{\mathbf{Q}}^2 + \sum_{i=0}^{M-1} \|\Delta \mathbf{c}(k+i|k)\|_{\mathbf{R}(\mathbf{i})}^2 \quad (5.2)$$

where P is the prediction horizon, and M is the control horizon. Q is the *tracking error weight*, and $\mathbf{R}(\mathbf{i})$ is the *control penalty weight vector*. The first term in the cost function represents the *tracking error*, *i.e.*, the difference between the response time $t(k+i|k)$ and a reference trajectory $ref(k+i|k)$. The reference trajectory defines an ideal trajectory along which the response time $t(k+i|k)$ should change from the current value $t(k)$ to the set point T_s (*i.e.*, desired response time). Our controller is designed to track the following exponential reference trajectory so the closed-loop system behaves like a linear system.

$$ref(k+i|k) = T_s - e^{-\frac{T}{T_{ref}}i} (T_s - t(k)) \quad (5.3)$$

where T_{ref} is the time constant that specifies the system response speed. A smaller T_{ref} causes the system to converge faster to the set point but may lead to a larger overshoot.

By minimizing the tracking error, the closed-loop system will converge to the response time set point T_s if the system is stable. The second term in the cost function (5.2) represents the *control penalty*, which causes the controller to decrease the change of the control input, *i.e.*, the CPU allocation. The control weight vector, $\mathbf{R}(\mathbf{i})$, can be tuned to represent a preference among the VMs. For example, a higher weight may be assigned to a VM if the process running it has a larger CPU demand so that the controller can give preference to increasing its CPU allocation.

A fundamental benefit of the control-theoretic approach is the confidence for system stability. We say that a system is *stable* if the response time $t(k)$ converges to the desired set point T_s , that is, $\lim_{k \rightarrow \infty} t(k) = T_s$. However, an MPC controller without additional constraint can be unstable. In optimal control theory [52, 49], the stability of an MPC controller can be ensured by adding a *terminal constraint*. The constraint forces the state to take a particular value at the end of the prediction horizon. Therefore, we add the terminal constraint to our optimization problem requiring the response time of the application to converge to the set point at the end of the prediction horizon:

$$t(k + M|k) = T_s \quad (5.4)$$

Based on the above analysis, the response time control has been modeled as a MIMO optimal control problem.

The controller must minimize the cost function (5.2) under the terminal constraint. This constrained optimization problem can be transformed to a standard least-squares problem [52]. The controller uses a standard least-squares solver to solve the optimization problem online.

5.2.3 CPU Resource Arbitrator

As introduced in Section 5.1, the response time controller determines the CPU resource demand of every VM. A server-level CPU resource arbitrator then allocates the CPU resource to the VMs hosted on the server based on their demands.

Specifically, the arbitrator on each server collects the CPU resource demand of every VM hosted on the server, in terms of CPU cycles per second (GHz), decides what CPU frequency the server should have in order to satisfy the aggregated demands, and then throttles the processor of the server to the desired CPU frequency. For example, a server S_1 (with the maximum CPU frequency of $5GHz$) hosts two VMs,

VM_{11} and VM_{32} . VM_{11} requires a CPU frequency of $1GHz$ and VM_{32} requires a CPU frequency of $2GHz$. Another $0.5GHz$ is reserved for the VM hypervisor. In this case, the arbitrator on S_1 decides that the CPU frequency of S_1 should be $1GHz + 2GHz + 0.5GHz = 3.5GHz$. Thus, the CPU frequency of S_i is dynamically set to:

$$f_i(k) = \min\left\{ \sum_{VM_{jl} \in S_i} c_{jl}(k) + c_0, F_{max}^i \right\} \quad (5.5)$$

where c_0 is the constant portion of the CPU resource reserved for the virtual machine hypervisor. If the arbitrator cannot satisfy the demands of all the hosted VMs even when the processor is running at the highest CPU frequency, the server is assumed to be overloaded. To deal with this case, the solution in this chapter can be integrated with algorithms to move VMs from the overloaded servers to other servers in an on-demand manner.

Please note that we mainly focus on CPU-intensive workloads (e.g., dynamic web content creation) in this dissertation. However, our design can be extended for IO-intensive workloads by adding IO requirements as constraints in the arbitrator, which is our future work. Please also note that if a VM is migrated to a server with different hardware, the same CPU frequency may have very different performance. In this case we assume that the CPU frequency value is already scaled based on an offline performance benchmark such as the Reference Performance Estimate 2 (RPE2) value [1].

Today's processors only support several discrete frequency levels. However, the new CPU frequency level periodically decided by the CPU resource arbitrator could be a value that is not one of the supported frequency levels. Our arbitrator includes a CPU frequency modulator which resolves the output value of the controller to a series of supported frequency levels to approximate the desired value. For example, to approximate 2.89GHz during a control period, the modulator would output a sequence of supported levels: 2.67, 3, 3, 2.67, 3, 3, etc on a smaller timescale. We implement the first-order delta-sigma modulator presented in [48] to perform this function.

5.3 Server Consolidation for Power Optimization

As discussed in Section 5.1, a cluster-level power optimizer is used to find the most power-efficient VM-server mapping, and reconfigure the cluster by VM migration. The optimizer then dynamically places selected servers into the sleep mode or wakes up selected servers for maximum power savings and guaranteed application performance.

In this section, we first formulate the power optimization problem. We then build a power model to capture the impact caused by the migration of a VM to a server. Finally, we propose Incremental Power Aware Consolidation (IPAC), a heuristics-based optimization algorithm, to find the most power-efficient VM-server mapping.

5.3.1 Problem Formulation

We first introduce the following notation. The processor of server S_l has a maximum CPU frequency of $F_{max}^{(l)}$ and a minimum frequency of $F_{min}^{(l)}$, *i.e.*, $f_l \in [F_{min}^{(l)}, F_{max}^{(l)}]$. $VM_{ij} \in S_l$ means VM_{ij} is hosted by server S_l . S_l consumes p_l Watts of power. When it is placed into the sleep mode, we assume $p_l = p_0$, which is a small constant value.

The power optimizer is invoked every T_o hours. As discussed in Section 5.1, the invocation period of the optimizer is much longer than the control period of the response time controller, *i.e.*, $T_o \gg T$. Therefore, during the invocation period of the optimizer, the response time controller makes a series of decisions regarding the CPU resource demands for every virtual machine VM_{ij} . We place the series of CPU resource demands for VM_{ij} into a vector $\vec{c}_{ij}$. We adopt the workload analysis method in [80] to estimate the CPU resource requirements in the next period based on the requirements in the previous control periods.

The optimizer minimizes the total power consumption of the cluster

$$\min \sum_{l=1}^N p_l(k) \quad (5.6)$$

The optimization problem is subject to the following constraints. First, every VM should be hosted by one server. Second, the CPU resource of each server should be enough to satisfy the CPU requirements of all the VMs hosted on it during the entire invocation period. Third, the memory resource of each server should be enough to satisfy the memory requirements of all the VMs hosted on it. The constraints are modeled as

$$\forall VM_{ij}, \exists 1 \leq l \leq n, s.t. VM_{ij} \in S_l \quad (5.7)$$

$$\max\left\{\sum_{VM_{ij} \in S_l} \vec{c}_{ij}\right\} \leq F_{max}^{(l)} \quad (1 \leq l \leq n) \quad (5.8)$$

$$\sum_{VM_{ij} \in S_l} MEM(VM_{ij}) \leq MEM(S_l) \quad (1 \leq l \leq n) \quad (5.9)$$

We include constraints (5.7), (5.8) and (5.9) in the experiments in Section 5.5. Other constraints defined by the system administrators can be added to the optimization algorithm. We list several possible constraints here.

1. The constraint of the available IO or network bandwidth and other computing resources.
2. The constraint of the power distribution infrastructure and power capacity.
3. The constraint of thermal densities and heat dissipation.

Please note that here we adopt a simple CPU resource constraint (5.8) to focus on the power optimization algorithm itself. More complex formulations of CPU resource constraints, *e.g.*, the 90-percentile CPU resource constraint [80], can be easily integrated into our power optimization algorithm.

5.3.2 Power Model

To have an effective power optimizer design, it is important to model the power consumption of the servers. It is well-known that DVFS can allow cubic reductions

in power density relative to performance loss for the processor in a server. Recent studies (*e.g.*, [21]) present a model between the power consumption of a server and the frequency of the processor as:

$$p_i(k) = a_i f_i^3(k) + b_i \quad (5.10)$$

where a_i and b_i are generalized parameters that may vary for different servers.

As described in Section 5.2.3, the CPU resource arbitrator enforces (5.5) by dynamically changing the frequency of the physical processor. As a result, the power consumption of a server is modeled as:

$$p_i(k) = a_i (\min\{ \sum_{VM_{jl} \in S_i} c_{jl}(k) + c_0, F_{max}^i \})^3 + b_i \quad (5.11)$$

The parameters a_i and b_i are then estimated from an offline analysis by curve fitting [83].

Using the power models (5.10) and (5.11), it is easy to ascertain 1) the power efficiency of the server, *i.e.*, the ratio between the maximum CPU frequency and maximum power consumption of the server, and 2) how the power consumption changes if a VM is moved from one server to another. The information is used to design the optimization algorithm.

5.3.3 Optimization Algorithm

The optimization problem formulated in Section 5.3.1 falls in the category of *vector-packing* problems which are known to be NP-hard [41]. Therefore, we propose a heuristics-based optimization algorithm to find a polynomial time approximate solution.

Minimum slack problem for a single server: We begin with a sub-problem named the minimum slack problem. The problem can be presented as: given a server

(not necessarily empty) and a list of unallocated VMs, select several VMs from the list, and allocate them to the server, such that the server has the least amount of unallocated CPU resource. This problem is a special case of the minimum bin slack (MBS) problem. Although it is an NP-hard problem, the MBS problem can be solved in pseudo-polynomial time [23]. The algorithm to solve the minimum slack problem is summarized in Algorithm 1. This algorithm is extended from the MBS algorithm in [23] by evaluating a more general constraint (5.8) in each step, instead of checking if the total size of the items exceeds the size of the bin.

Power aware consolidation for a list of servers: Another sub-problem, the power aware consolidation problem, can be presented as: given a list of servers (some servers are possibly not empty) and a list of VMs, consolidate the VMs to the servers, such that the total power consumption of the servers is minimized. Our proposed heuristic algorithm to solve this problem is described in Algorithm 2. In the first step, the servers are sorted by power efficiency, *i.e.*, the ratio between the maximum CPU frequency and maximum power consumption of the server, which can be derived from (5.10). Beginning from the most power-efficient server, we use Algorithm 1 to select several VMs from the remaining unallocated VMs, and then pack these VMs to this server such that the unused CPU resource in this server is minimized. We repeat this process with the next most power-efficient server until every VM in the list is allocated to a server.

Incremental Power Aware Consolidation (IPAC) algorithm: Using Algorithm 2 to directly consolidate all the VMs may incur a very large number of VM migrations with a large overhead. Therefore, Algorithm 2 is invoked incrementally such that only a small number of VMs in a *migration list* are considered for consolidation each time. In each invocation period, some servers may be unable to host their VMs due to the possible workload increase. The algorithm first selects some VMs from these overloaded servers and adds them to the migration list to resolve the overload problem. Then, the VMs on the least power efficient server are added to the migration list. Algorithm 2 is invoked to consolidate the VMs in the migration

list to the servers. After the consolidation, if the number of active servers is reduced, Algorithm 2 is invoked again to consolidate the VMs on the next least power efficient server until the number of active servers no longer decreases.

Cost-aware VM migration: The cost of the VM migration can be considerable. For example, if the network bandwidth is a bottleneck in a data center, a VM migration with high bandwidth consumption is the least preferred method. As a result, when the IPAC algorithm requests a migration, benefits and costs should be compared to decide if the migration should be allowed or rejected. The benefits of the migration include power savings which can be easily inferred from the model in (5.11). The cost of migration depends highly on the condition of the data center such as the network architecture, the bandwidth usage, the application itself and the memory usage of the VMs. Thus, the cost function can be highly different for different data centers. As a result, we provide an interface for data center administrators to define their own cost functions based on their various policies.

Overhead: The complexity of the IPAC algorithm depends on the complexity of Algorithm 1, which can be analyzed in a similar way as in [23]. It is fast enough in practice. In an experiment to several standard bin-packing problems to pack 1,370 items in [23], an algorithm with the same complexity as Algorithm 1 used an average CPU time of 0.02s to finish. In our simulation with 5,415 VMs in Section 5.5, the total CPU time used in 7 days to run IPAC is less than 314s.

The latency caused by switching servers between sleep and active modes adds additional overhead to our algorithm. However, the latency can be eliminated by always keeping one or two idle servers active. In that way, the optimizer can immediately use an idle server when it needs to wake up a server. After the idle server is taken by the optimizer, another server is waken up to get prepared for the optimizer. As a result, the optimizer never actually needs to wait for a server to wake up.

Algorithm 1 Minimum Slack

q : list of unallocated VMs

S : the server in consideration

ε : allowed slack

s^* : minimum slack

A^* : the collection of VMs best fits S

Minimum-Slack (q)

begin

```
1: for all VM  $VM_i$  in  $q$  do
2:   Pack  $VM_i$  into  $S$ .
3:   if  $S$  and the VMs allocated to it meets the constraint (5.8) then
4:     if Remaining CPU resource in  $S < \varepsilon$  then
5:       Exit;
6:     end if
7:     Minimum-Slack( $q - VM_i$ );
8:   else
9:     Remove  $VM_i$  from  $S$ ;
10:  end if
11:  if  $Slack(S) < s^*$  then
12:     $s^* = Slack(S)$ 
13:     $A^* = \text{All VMs} \in S$ 
14:  end if
15:  if The algorithm does not finish in certain steps then
16:    Increase  $\varepsilon$  by one step.
17:  end if
18: end for
end
```

5.4 System Implementation

In this section, we first introduce our testbed and the implementation details of each component. We then present the simulation environment used to test our IPAC algorithm in various cluster configurations.

5.4.1 Testbed

Our testbed includes a cluster of four physical computers named $S1$ to $S4$. A fifth computer named *Storage* is used as the storage server for the Network File System (NFS) and is not part of the cluster. All the computers run Fedora Core 8 with Linux

Algorithm 2 Power Aware Consolidation

Power efficiency of a server: the ratio between the maximum CPU frequency and the maximum power consumption.

Invoke in the end of each operation period

Assume all servers are empty. All VMs are placed in a waiting list to be allocated to servers.

Allocate-VMs-to-Servers ($\vec{c}_{ij}$)

begin

- 1: Sort the servers by power efficiency;
- 2: **while** VM list not empty **do**
- 3: Pick the most power-efficient non-full server;
- 4: Pack VMs into this server with Algorithm 1;
- 5: **end while**

end

kernel 2.6.21. $S1$ and $S3$ are each equipped with 4GB RAM and an AMD Opteron 2222SE processor, which supports 8 frequency levels from 1GHz to 3GHz. $S2$ and $S4$ are each equipped with 4G of RAM and an Intel Xeon X5160 processor, which has 4 frequency levels from 2GHz to 3GHz. All the servers are connected via a gigabyte Ethernet switch. Every server is connected with a power meter to measure and log the power consumption.

Xen 3.3 is used as the virtual machine monitor on all four servers in the cluster. We use a PHP implementation of the RUBBoS benchmark [40], a standard bulletin board benchmark, as our server side workload. Each instance of RUBBoS is configured to be a two-tier application running in two VMs. The first tier has an Apache server installed and works as a webserver running application scripts. The second tier has a MySQL database installed and acts as a database server. We run 8 applications in our cluster. Each is an instance of RUBBoS. Therefore, we use 16 VMs on the four servers. The configuration is shown in Figure 5.2. In our server consolidation experiment, to demonstrate that the power optimizer consolidates the servers when the cluster has light workloads, we enable 6 VMs to run 3 applications ($App3 - App5$).

The client-side workload generator is the Apache HTTP server benchmarking tool (ab). This tool allows users to manually define the concurrency level, which is

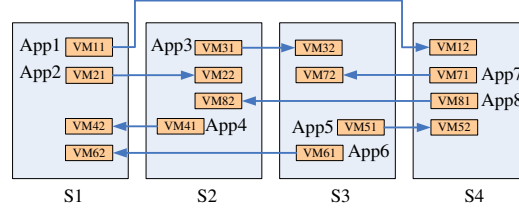


Figure 5.2: Our testbed consisting of 4 servers.

the number of requests to perform in a short time, to emulate multiple clients. A concurrency level of 40 is used in our experiments to do system identification and most experiments, if not otherwise indicated. The workload generator runs on the Storage computer.

5.4.2 Control Components

We now introduce the implementation details of each component in our response time control loop.

Response Time Monitor. To eliminate the impact of network delays, we focus on controlling the server-side response time in this chapter. The response time monitor is implemented as an Apache module in each VM running Apache. For every request added to the application, two time stamps are used when a request is received and when the corresponding response is made, respectively. The difference value is used as the server side response time of the request. At the end of each control period, we calculate the 90-percentile response time of all requests finished during the control period. This is used as the server-side response time and is sent to the corresponding response time controller.

Response Time Controller. As introduced in Section 5.1, a response time controller exists for each application. The controller implements the control algorithm presented in Section 5.2. A cluster-level daemon written in C++ sequentially runs the controller of every application on Storage and computes the total CPU resource

requested by all the VMs. The daemon program calls the CPU resource arbitrator to enforce the desired CPU allocation, based on the CPU resource requests, and enforces the CPU frequency. The control period of the response time controller is selected to be 6 seconds such that multiple HTTP requests can be processed in one period. The set point of the response time controller (*i.e.*, the desired response time) is selected to be 1 second. This number can be chosen by the system administrator based on the nature of the application and can be changed online. The system parameters in (5.1) resulting from the system identification are $\alpha_{11} = 0.2663$, $\beta_{11} = [-0.9116, -1.527]$, $\beta_{12} = [0.2717, -0.6833]$. In the controller design, we set $Q = 1000$ and $\mathbf{R} = \text{diag}\{300, 100\}$ in (5.2) to provide preferential treatment to the Apache VM in each application since it typically requires more CPU resource. We set $T_{ref} = 2T$ in (5.3) for a trade-off, as a smaller T_{ref} causes the system to converge faster to the set point but may lead to larger overshoot.

CPU Resource Arbitrator. On every server, a daemon program written in Perl runs a CPU resource arbitrator as introduced in Section 5.2.3. In every control period, the arbitrator collects the CPU resource requirements, allocates the CPU resource, and changes the CPU frequency using the approach introduced in Section 5.2.3. For the CPU resource allocation, we use Credit Scheduler, Xen’s proportional share scheduler. In Credit Scheduler, each VM is assigned a *cap* and a *weight*. Cap represents the upper limit of CPU time (as a percentage) that can be allocated to the VM. Weight is used to give the preference among different VMs. In this chapter, we use cap to allocate CPU resource and use the same fixed weight for all VMs.

Power Monitor: The power consumption of each server in the cluster is measured with a WattsUp Pro power meter [20] by plugging the server into the power meter, which is then connected to a standard 120-volt AC wall outlet. The WattsUp power meter has an accuracy of 1.5% of the measured value.

5.4.3 Simulator

To evaluate our power optimizer in large-scale data centers, we have developed a C++ simulator that uses a trace file from real-world data centers [84] to simulate the CPU utilization variations. The trace file includes the utilization data of 5415 servers from ten large companies covering the manufacturing, telecommunications, financial, and retail sectors. The trace file records the average CPU utilization of each server every 15 minutes from 00:00 on July 14th (Monday) to 23:45 on July 20th (Sunday) in 2008. We treat the utilization data of each server as the CPU demand of a VM. We randomly generate 3000 simulated servers to host these VMs based on the parameters used in the simulated servers listed in Table 5.1. The parameters are obtained from the results using real servers and standard web application benchmarks, as reported in [75].

Table 5.1: Server models and parameters used in large-scale simulations.

Type	Active Power		Sleep	Processor			RAM
	Max	Min		Max Freq.	Min Freq.	Cores	
Transport GT24 (B2912-E)	264W	121W	51W	2700MHz	800MHz	2X4	16G
Supermicro 2021M-UR+	269W	138W	56W	2500MHz	800MHz	2X4	16G
HP ProLiant DL385 G5p	257W	147W	44W	2700MHz	800MHz	2X4	8G
Dell PowerEdge 2970	302W	139W	86.4W	2300MHz	1150MHz	2X4	16G
HP ProLiant DL385 G6	258W	120W	47.6W	2600MHz	800MHz	2X6	16G

5.5 Experimentation

In this section, we present our empirical and simulation results. We first evaluate the response time controller and examine the power optimizer on the hardware testbed. We then present the simulation results of our power optimization algorithm in a data center with 3000 servers and about 5415 VMs.

We use pMapper, a heuristic-based server consolidation algorithm proposed in a recent paper [79], as our baseline. PMapper is an incremental algorithm with two phases. In the first phase, it sorts the servers based on their power efficiency,

then consolidates the VMs to the servers using a first-fit algorithm, beginning with the most power efficient server. Note that in this phase, the VMs are not actually migrated. In the second phase, pMapper computes the list of servers that require a higher utilization in the new allocation, and labels them as receivers. For each donor (servers with a target utilization lower than the current utilization), it selects the smallest-sized applications and adds them to a VM migration list. It then runs first-fit decreasing (FFD) to migrate the VMs in the migration list to the receivers.

Several key differences exist between pMapper and our IPAC algorithm. First, pMapper is adapted from FFD while IPAC is adapted from Minimum Slack. Typically, Minimum Slack provides a better solution in terms of power consumption, especially when facing constraints such as memory constraint, bandwidth constraint, etc. Second, although Minimum Slack generally has a greater overhead compared with FFD, the IPAC algorithm considers only a very small number of VMs each time, while pMapper considers all the VMs in the first phase. Therefore, the computational overhead of IPAC is comparable to that of pMapper. The execution time comparison result can be found in Section 5.5.3. Third, IPAC is combined with our response time controller to save more power using DVFS, based on the description in Section 5.2.3. Thus IPAC provides further power savings. Finally, the response time controller combined with IPAC can provide desired performance assurance in response to short-term workload variations.

5.5.1 Response Time Control

In this experiment, we disable the power optimizer to evaluate the response time controllers of the 8 applications running in the cluster as shown in Figure 5.2. We first set the response time target for all applications to be $1000ms$. Figure 5.3 plots the means and the standard deviations of the response times of the applications in the cluster. This figure demonstrates that the response time controller works effectively to achieve the desired response time for all the applications. Figures 5.4(a) and (b)

show a typical run of the response time controller for a randomly selected application, *App5*. At the beginning of the run, the controller achieves the desired response time set point, *i.e.*, 1000ms, after a short settling time. The workload of *App5* increases significantly at a time of 600s. This is common in many web applications, *e.g.*, breaking news on a major newspaper website may incur a large number of accesses in a short time frame. To stress test the performance of our controller in such a scenario, we increase the concurrency level of *App5* from 40 to 80 between time 600s and time 1200s to emulate the workload increase. The suddenly increased workload causes *App5* to violate its response time limit at time 600s. The response time controller responds to the violation by allocating more CPU resource to the two VMs in both tiers. As a result, the response time of *App5* converges to 1000ms again and the power consumption of the cluster increases slightly due to the increased CPU resource usage, as shown in Figure 5.4(b).

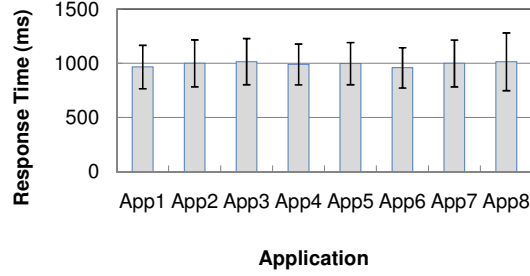
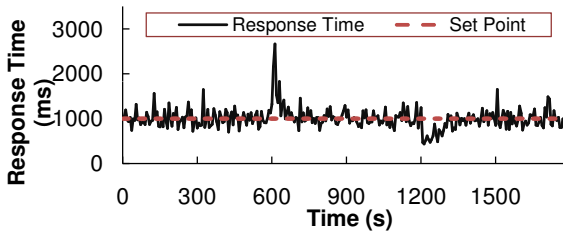


Figure 5.3: Response time of all 8 applications in the cluster.

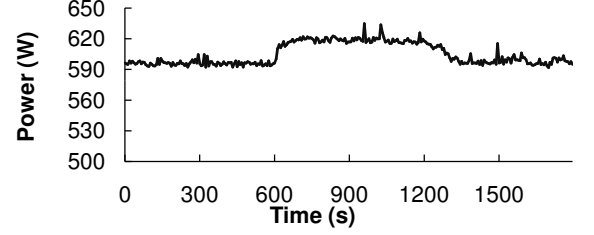
We use the baseline pMapper* for comparison in this experiment. Since pMapper does not manage CPU resource, we assume that pMapper equally allocates CPU resource to all the VMs running on every server. PMapper represents a typical solution that relies only on server consolidation to save power without applying DVFS and resource allocation to guarantee performance and save power on short time scales.

Figures 5.4 (c) and (d) show the performance of pMapper in the same scenario. Clearly, during the workload increase, the relative response time of *App5* increases

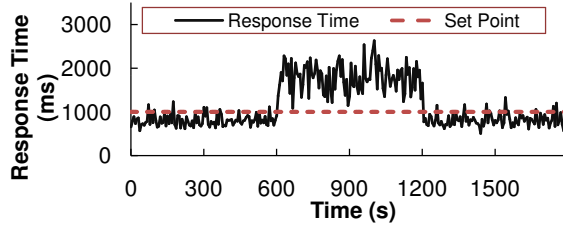
*The authors of pMapper have acknowledged the need to integrate resource management with server consolidation and mentioned it as a possibility to extend pMapper [79].



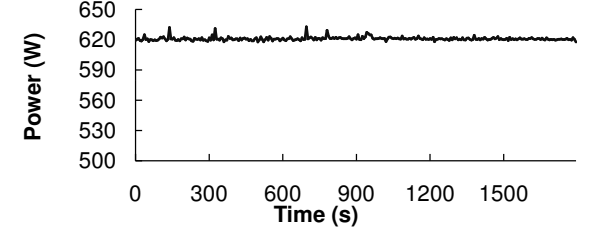
(a) Response time of App5 (response time controller)



(b) Power of the cluster (response time controller)



(c) Response time of App5 (pMapper)



(d) Power consumption of the cluster (pMapper)

Figure 5.4: Typical runs of the response time controller and the baseline pMapper under a workload increase from time 600s to 1200s on App5.

significantly. The inferior performance of pMapper is caused by its policy to allocate CPU resource in a static way. Consequently, pMapper cannot adjust CPU resource among different VMs in response to workload variations. This experiment demonstrates that the response time controller can effectively maintain application-level response times. In addition, a server consolidation algorithm alone may not be able to provide application-level performance assurance without a performance controller. Therefore, it is important to integrate dynamic CPU resource allocation, DVFS, and server consolidation for maximized power savings with performance assurance.

To test the robustness of the response time controller when it is applied to a system that is different from the one used to do system identification, we conduct a set of experiments with wide ranges of concurrency levels. Figure 5.5 shows the average response times (with standard deviations) achieved by the controller when the concurrency level varies from 30 to 80. Figure 5.6 shows the average response times (with standard deviations) achieved by the controller when the response time set

point increases from 600ms to 1300ms. The controller achieves the desired response time for all the currency levels and set points. The experiments demonstrate that the response time controller works effectively to achieve the desired response time when the actual system is under a workload different from the one used to design the controller.

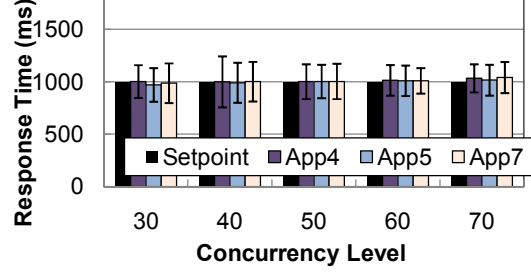


Figure 5.5: Response time of App5 under different workloads.

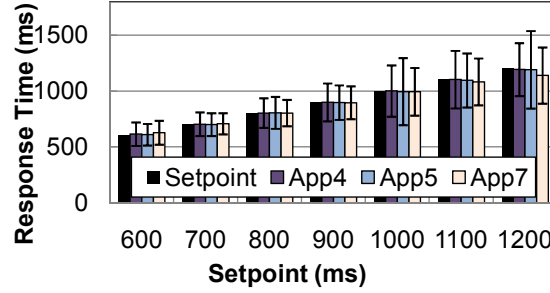


Figure 5.6: Response time of App5 under different set points.

5.5.2 Server Consolidation for Power Optimization

In this subsection, we evaluate our power optimizer designed for dynamic server consolidation on our tesdtbed.

Figure 5.7 shows the power consumption of the cluster measured using a power meter before and after a server consolidation. Initially, all servers are active. The power consumption is approximately 544 W. At time 600s, the power optimizer is

activated and consolidates all 6 VMs to the most power-efficient server. After 102s, the three idle servers are put to sleep, reducing the power consumption of the cluster to approximately 165W. This leads to a power saving of 69.6%.

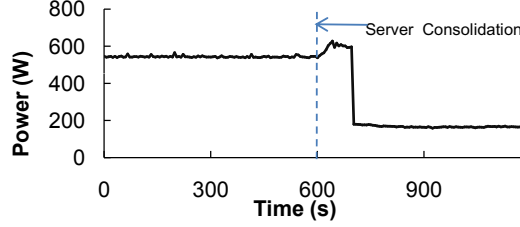


Figure 5.7: Power consumption of the cluster. Power optimizer is activated at time 600s.

In a typical server consolidation process, virtual machine live migration is used to shift a VM from one server to another. It is important to understand how migration affects the application-level performance. Figure 5.8 demonstrates the response time of an application when the VM running its database service migrates from one server to another. Initially, the response time has been successfully controlled by the response time controller. When the optimizer starts the VM migration, the response time has a temporary increase since the migration operation consumes some CPU and bandwidth resources. After two control periods, the application-level response time again achieves its desired set point. This experiment demonstrates that our management solution effectively reduces the power consumption of the cluster while the application-level response time is only affected for a very short time during the migration.

We conduct a series of experiments to test the efficacy of our solution under different number of active applications. We measure the power savings of our solution from the power meter and compare it with a baseline when all servers are kept active and all VMs are deployed among the servers in a balanced way. Figures 5.9 and 5.10 show the power savings and the average performance of the applications in these experiments, respectively. The power saving changes from 294.0W to 502.6W as the

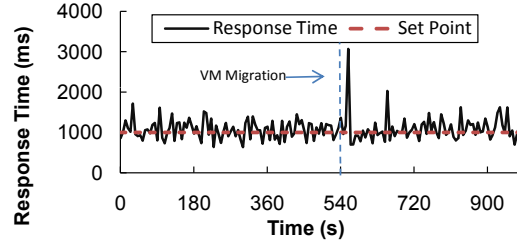


Figure 5.8: Response time of an application during the VM running its database service migrates to another server.

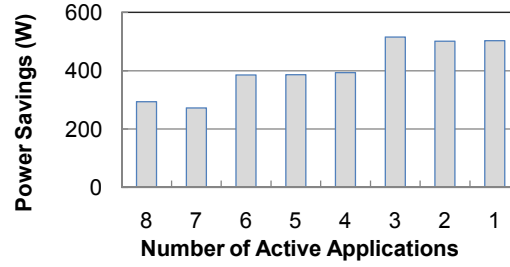


Figure 5.9: Average power savings of the cluster with different number of applications.

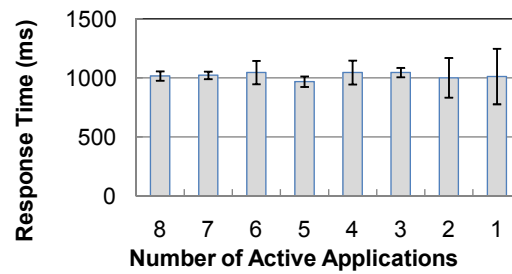


Figure 5.10: Response time of the applications in the cluster with different number of applications.

number of active applications changes from 8 to 1. The power saving significantly increases as the number of application changes from 7 to 6 and from 4 to 3. This is because the number of active servers decreases in these experiments. The response time controller is able to meet the set point of 1000ms for all the applications in all these experiments. These experiments show that our solution provides more power savings when the utilization of the cluster decreases, by putting more servers into the sleep mode.

5.5.3 Simulation Results for Large Data Centers

In this experiment, we test our power optimizer in large-scale data centers using the simulation environment introduced in Section 5.4.3.

We simulate 54 data centers with different number of VMs, ranging from 30 to 5,415. Every data center is assumed to have enough inactive servers which will be waken up and used if necessary. The parameters of the servers are introduced in Section 5.4.3. As an example of administrator-defined real world constraints, in addition to the constraints in (5.8), we add a restriction to the optimization algorithm such that the memory size of every server should be greater than the total memory allocations of the hosted VMs.

Figure 5.11 plots the average energy consumption per VM of IPAC and pMapper in 7 days under different number of VMs. In comparison to pMapper, IPAC shows lower energy consumption in all these simulations. On average, IPAC has a 51.2% more energy saving than pMapper. It is important to note that the energy savings of IPAC are due to two reasons. First, as discussed earlier, pMapper is adapted from FFD while IPAC is adapted from Minimum Slack. Therefore, IPAC consolidates VMs to fewer servers than pMapper does. Second, IPAC is integrated with DVFS for power savings on a short time scale between two consecutive invocations of the optimization algorithm. Thus, IPAC saves more power when a lower CPU frequency is allowed due to short-term variations on the CPU requirements. Note that the testbed experiments

in Section 5.5.1 show a smaller power saving because only DVFS is tested and the CPU requirements of the workloads are relatively stable. To demonstrate the contribution to energy savings from the two reasons, we plot the fraction of energy savings over pMapper resulted from DVFS in Figure 5.12. On average, DVFS accounts for 67% of the total energy savings over pMapper and the rest 33% energy savings are resulted from the better consolidation quality that IPAC provides.

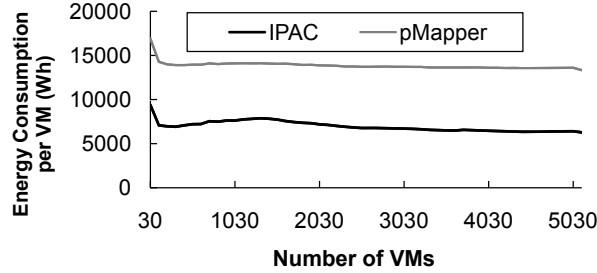


Figure 5.11: Energy consumption per VM in 7 days under different numbers of VMs.

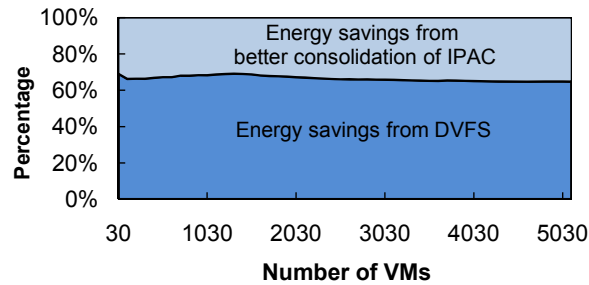


Figure 5.12: Percentage of energy savings (over pMapper) as results of DVFS and better consolidation of IPAC.

Figure 5.13 demonstrates that IPAC conducts a smaller number of migrations than pMapper. On average, IPAC moves each VM 0.13 times in a week while pMapper moves each VM 6.76 times. This is due to the reason that, in each period, IPAC only considers a much smaller list of VMs for migration. The migration list for IPAC contains selected VMs from the overloaded servers and the VMs on one or more least power efficient servers (see Section 5.3 for details). When the number of servers is

large, the major part of the migration list is the selected VMs from the overloaded servers. On the other hand, the migration list of pMapper contains selected VMs from each donor *i.e.*, the server whose utilization is higher than an ideal utilization target determined by an FFD algorithm. Every overloaded server is a donor since the utilization of a overloaded server ($\geq 100\%$) is higher than any reasonable ideal utilization target ($\leq 100\%$). Therefore, the migration list of pMapper is longer than IPAC when the number of servers is large, leading to more migrations needed for pMapper.

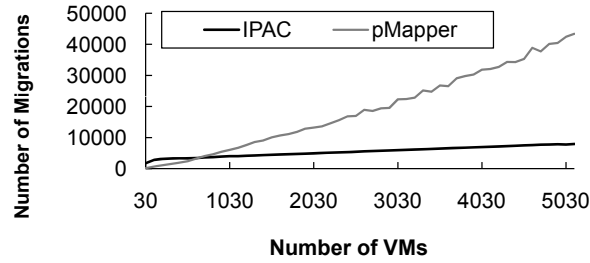


Figure 5.13: Number of migrations of IPAC and pMapper in 7 days under different numbers of VMs.

To compare the complexity of the two algorithms, we measure the execution time of them in the 54 simulations for the 54 data centers. Results show that the average execution time of IPAC and pMapper in all these simulations are 340s and 109s, respectively. These results demonstrate that IPAC uses a longer and still acceptable execution time but outperforms pMapper significantly in terms of both power efficiency and migration penalty.

5.6 Summary

In this chapter, we have presented a performance-controlled server consolidation solution for virtualized data centers to achieve power efficiency and application-level performance assurance. While existing solutions rely on DVFS or server consolidation in a separate manner, our solution integrates feedback control with optimization

to utilize both DVFS and server consolidation for maximized energy savings with performance guarantees. At the application level, a MIMO controller is designed based on advanced control theory to achieve the desired 90-percentile response time for applications spanning multiple VMs, on a short time scale, by reallocating CPU resource and DVFS. At the cluster level, a power optimizer is proposed to dynamically consolidate VMs onto the most power-efficient servers on a much longer time scale. Empirical results on a hardware testbed demonstrate that our solution outperforms pMapper, a state-of-the-art server consolidation algorithm, by having greater power savings and much smaller migration overheads while achieving the required application performance. Extensive simulation results, based on a trace file of 5,415 real servers, demonstrate the efficacy of our solution in large-scale data centers.

Chapter 6

Energy-Efficient Virtual Machine Scheduling in Performance-Asymmetric Multi-Core Architectures

Multi-core architectures with asymmetric core performance have recently shown great promise, because applications with different needs can benefit from either the high performance of a fast core or the high parallelism and power efficiency of a group of slow cores. This performance heterogeneity can be particularly beneficial to applications running in virtual machines (VMs) on virtualized servers, which often have different needs and exhibit different performance and power characteristics. Therefore, scheduling VMs on performance-asymmetric multi-core architectures can have a great impact on a system's overall energy efficiency. Unfortunately, existing VM managers, such as Xen, have not taken the heterogeneity into account and thus often result in low energy efficiencies.

In this chapter, we propose a novel VM scheduling algorithm that exploits core performance heterogeneity to optimize the overall system energy efficiency. We first

introduce a metric termed energy-efficiency factor to characterize the power and performance behaviors of the applications hosted by VMs on different cores. We then present a method to dynamically estimate the VM’s energy-efficiency factors and then map the VMs to heterogeneous cores, such that the energy efficiency of the entire system is maximized. We implement the proposed algorithm in Xen and evaluate it with standard benchmarks on a real testbed. The experimental results show that our solution improves the system energy efficiency (*i.e.*, performance per watt) by 13.5% on average and up to 55% for some benchmarks, compared to the default Xen scheduler.

6.1 Background and Motivations

In this section, we first describe the energy efficiency metric used in our work. We then briefly discuss CPU scheduling in Xen and use an example to show how the existing scheduling algorithm may result in poor energy efficiency. Finally, we discuss how to use models to estimate performance and power for CPU cores and calculate a system’s energy efficiency.

6.1.1 Energy Efficiency

The main objective of the work is to optimize energy efficiency. Performance per watt has been widely used to measure the energy efficiency. The performance metric used depends on the scenario. For a physical server or virtual machine, the performance can be measured as the system’s throughput, *i.e.*, the number of instructions per second. The energy efficiency is hence defined as number of instructions per second per watt. At the application level, an application’s throughput can be used to measure the performance, and define the energy efficiency as throughput per watt, *e.g.*, 15 transactions/sec per watt. Since our work is focused on low level VMs scheduling and has no application level knowledge, we use the number of instructions per second

as the performance metric. More specific, energy efficiency of the whole system is defined as

$$EnergyEfficiency = \frac{Performance}{Power} \quad (6.1)$$

where *Performance* denotes the throughput of a computer server in Billions of Instructions Per Second (BIPS) and *Power* is the power consumption of the whole system. Note that the energy efficiency in terms of BIPS per watt is equivalent to Billion of Instructions per Joule. Hence, maximizing the energy efficiency in (6.1) also means minimizing the energy consumption in Joule required to complete a predetermined number of instructions.

Our algorithm does not directly optimize the application-level performance, such as the throughput or the response time of web requests, the total execution time of applications. This is due to the fact that a VM scheduling algorithm, such as the one proposed in this dissertation, typically has no knowledge of the application-level metric. Thus it is impractical to optimize the application-level performance directly. However, our algorithm can be integrated with other application-level management algorithms to achieve an energy-efficient management solution. For example, in virtualized platforms designed by Amazon for running user-submitted tasks that process vast amounts of data [5], it is important to boost the applications that have slower progress to minimize the total execution time. However, a scheduler itself cannot achieve this by scheduling the applications with slow progress on fast cores, because the scheduler has no knowledge of the application-level details and thus, has difficulties in determining the progress of the applications. A more practical approach is to integrate our scheduling algorithm with an application-oriented management algorithm which balances the progress of the applications by giving the application with a slower progress more CPU time using the public interface of the scheduler, *e.g.*, the weight and cap parameters in the Credit Scheduler discussed in Section 6.1.2. Because the scheduling algorithm attempts to maximize the energy efficiency

defined in (6.1), integrating our algorithm with an application-oriented management algorithm reduces the energy consumption of the entire system. We present a detailed example of the integration in Section 6.5.2.

Additionally, the proposed algorithm does not directly change the core configurations, *e.g.*, the frequency levels of the cores. Thus our algorithm can be integrated with many power oriented algorithms which tune the frequency levels of the cores based on the power management objectives, *e.g.*, fitting the processor or the server into a certain power or thermal envelope [84, 54].

6.1.2 CPU Scheduling in Xen

In this section, we provide a brief introduction to Credit Scheduler, the default CPU scheduler in Xen.

In Xen, each VM has one or more Virtual CPUs (VCPUs) when it is created. These VCPUs are the CPUs visible to the guest operating systems. Xen uses CPU schedulers to run VCPUs on physical cores.

For the system administrators who desire to control the share of CPU time each VM receives, Credit Scheduler provides two parameters, weight and cap, for each VM in the management interface. A non-zero cap limits the amount of CPU time a VM receives. For example, a cap value of 50 indicates the VM can receive at most 50% of the CPU time of a core. The value of the weight determines the CPU share between VMs. A VM with a weight of 512 may receive twice as much as CPU time as a VM with a weight of 256.

The VMs' weight and cap parameters determine the way how VCPUs' priority levels change as they consume CPU time slices. Credit Scheduler usually uses two levels of priorities to indicate whether a VCPU has exceeded its portion of CPU time defined by the weight and cap of the corresponding VM. In this dissertation we

refer to the two priorities of VCPUs as **high** (the portion of CPU time has not been exceeded) and **low** (the portion of CPU time has already been exceeded)*.

Each core has a local run queue of VCPUs that are scheduled to run on the core and has 0 or 1 VCPUs currently running on the core. When the currently running VCPU blocks, yields, or completes its time slice, the core looks at the local run queue. If the local run queue is empty, or all VCPUs in the queue have low priority levels, the core looks for high priority VCPUs from queues of other cores to do load balancing. Load balancing guarantees that in the entire system, low-priority VCPUs will not run while any high-priority VCPUs are still waiting in the run queues. In this way, VCPUs with high priorities are balanced among the cores. Each core is numbered when the system starts. When a core performs load balancing, it searches cores in numeric order beginning from the next one in sequence. When a core performs load balancing on the run queue of another core, it locks the run queue in a thread-safe way such that the operations to the run queue by other cores are temporally disabled. Thus, the load balancing operation is designed to be highly light-weighted to minimize the performance loss introduced by load balancing.

6.1.3 Motivation Example

We use an example system consisting of two fast cores and one slow core to illustrate the motivation to design an energy efficient scheduler. In this example system, Core 1 and Core 2 are fast cores and Core 3 is slow. The power and performance behaviors of the three running VCPUs are listed in Table 6.1 and Figure 6.1. The power consumption of the components other than the CPU cores is 165.7 W. These numbers are obtained from a real system in our testbed. We study two cases.

Case 1: VCPUs 1, 2, and 3 are running on Cores 1, 2, and 3, respectively. The power efficiency defined in (6.1) is: $\frac{0.303+0.247+0.031}{27.2+49.0+24.7+165.7} = 0.00218$ BIPS per watt.

*Credit Scheduler refers to the high and low priorities as UNDER and OVER. A less frequently used priority named BOOST is available in some special cases. The details on how priorities change as a VCPU runs are available in [2].

Case 2: In the same system, however, if VCPUs 1, 2, and 3 run on Cores 1, 3, and 2, respectively, the power efficiency becomes: $\frac{0.303+0.073+0.121}{27.2+19.6+61.8+165.7} = 0.00181$ BIPS per watt.

Table 6.1: Power and performance behaviors of 2 VCPUs on fast and slow cores in an example.

VCPU	On fast cores		On slow cores	
	Power	Performance	Power	Performance
VCPU 1	27.2W	0.303 BIPS	10.9W	0.111 BIPS
VCPU 2	49.0W	0.247 BIPS	19.6W	0.073 BIPS
VCPU 3	61.8W	0.121 BIPS	24.7W	0.031 BIPS

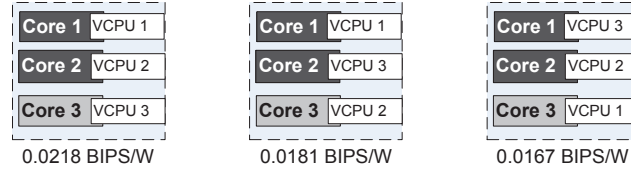


Figure 6.1: The energy efficiency of the system can be improved by selecting an energy-efficient VCPU-core mapping.

We further demonstrate the differences in energy efficiencies in different VM-core mapping strategies on an experiment in our testbed detailed in Section 6.4. Of the 12 cores in the system, 6 cores run at a frequency of 2 GHz and the other 6 cores run at 0.8 GHz. We create 6 VMs to run the *gobmk* benchmark [73] and another 6 VMs to run *sjeng* [73]. We run the experiments in two different configurations. In the first configuration we always pin the VMs running *gobmk* on the fast cores and the VMs running *sjeng* on the slow cores, with one VM running on one core. In the second configuration we pin the VMs in the opposite way: those running *sjeng* on fast cores and the ones running *gobmk* on slow cores. The two configurations show different energy efficiencies: the first configuration achieves a 9% better energy efficiency than the second configuration. Thus, in this example, the energy efficiency of the system can be improved by selecting an energy-efficient VCPU-core mapping.

The observation in these examples is that mapping VMs with different performance and power characteristics to heterogeneous cores will have an impact on the overall energy efficiency of the system. In the default Xen Credit Scheduler, VCPUs are moved between different cores based only on their priorities without considering their power and performance behaviors. As a result, it is highly possible that the VCPUs are mapped to cores in an energy-inefficient way.

6.1.4 Performance and Power Models

In this subsection, we present two models to estimate the performance, power consumption and energy energy.

Modern processors often provide general-purpose performance counters which provide unit-by-unit views of a wide variety of processor performance events [39, 7]. As discussed in Chapter 1, we focus on multiple cores with heterogeneous CPU frequencies. The dynamic power of a core, *i.e.*, the part of the power that changes with the activities of the VCPU and type of the core, can be modeled as

$$Pd_i = a_i f_i \tag{6.2}$$

where f_i is the frequency of the i^{th} core and a_i is a parameter related to the activities of the VCPU [61]. Note that if the processor implements dynamic voltage and frequency scaling (DVFS) for individual cores, the change in the frequency and voltage of a core has a cubic impact on the dynamic power [67]. However, the linear model in (6.2) still provides an acceptable approximation because real processors only have a limited frequency range [86].

Therefore, the power consumption of the whole system with n active cores can be presented as

$$Power = a_1 f_1 + a_2 f_2 + \cdots + a_n f_n + B \tag{6.3}$$

where B is the power consumption of all components other than the CPU cores in the system.

The performance (BIPS) of a VCPU when it is running on a core with a frequency of f_i can be modeled as

$$Perf_i = c_i f_i \quad (6.4)$$

where c_i is the value of the Instructions per Cycle (IPC) of the VCPU. Moreover, we assume that c_i is not a function of the frequency. In reality, IPC may slightly change with the frequency because of off-chip accesses. However, this dependency is typically negligible because the change in an application thread's IPC when the frequency changes is typically smaller than the IPC change across different application threads [78].

The parameters a_i and c_i in the models (6.2) and (6.4) characterize the power and performance of a VCPU on cores with different frequencies. The models can be calibrated through a set of benchmark experiments.

6.2 Energy-Efficiency Factor

As shown in the examples in Section 6.1.3, the energy efficiency of the system can be improved by scheduling the VCPUs based on the detailed knowledge of their performance and power characteristics shown in Figure ???. Theoretically, if we can obtain the performance and powers characteristics for each VCPU characteristics on different types of cores as described in Fig. 1, we can find out an optimal VM-core mapping to maximize the energy efficiency. However, it's difficult to obtain such parameters in practice. First, we need to know the the applications running inside each VCPU, which is often impractical. Second, we need to conduct a comprehensive offline profiling to measure the characteristics of each application on each type of cores. However, core-level power measurement is difficult in hardware implementation and is not widely available in today's hardware. Further, such a profiling may involve a

large number of applications and types of cores and make it a very time consuming task. Finally, the power and performance behaviors of the applications often show significant variations from time to time. Thus, the static performance and power characteristics from offline profiling cannot be applied for the entire life cycle.

In this section, we introduce energy-efficiency factor, a metric to characterize the power and performance behaviors of a VCPU on different cores. We show this simple metric can well characterize how efficiently a VCPU runs on different types of CPU cores and be used to guide VCPU-core mapping to optimize the energy efficiency. We further propose an approach to estimating the energy-efficiency factor using the hardware performance counters in the cores.

6.2.1 Definition of Energy-Efficiency Factor

Definition 1. *energy-efficiency factor:* We define θ_i in (6.5) as the energy-efficiency factor of a VCPU i .

We now show that for a two core system (*i.e.*, a slow core and a fast core), scheduling the VCPU with a higher energy-efficiency factor on the fast core achieves better energy efficiency.

We consider a system with two heterogeneous cores: Core 1, a fast core with frequency f_1 , and Core 2, a slow core with frequency f_2 , ($f_1 > f_2$), and two VCPUs: VCPU 1 and VCPU 2 with different power and performance characteristics. For two different VCPU-core mappings, we calculate their energy efficiency as follows.

Mapping 1: VCPU 1 on Core 1 and VCPU 2 on Core 2. The power consumption of the entire system is $a_1f_1 + a_2f_2 + B$. The performance is $c_1f_1 + c_2f_2$. Hence, the energy efficiency as defined in Section 6.1.1 is $\frac{c_1f_1+c_2f_2}{a_1f_1+a_2f_2+B}$.

Mapping 2: VCPU 1 on Core 2 and VCPU 2 on Core 1. Similarly, the energy efficiency is $\frac{c_1f_2+c_2f_1}{a_2f_1+a_1f_2+B}$.

Now we prove that Mapping 1 is more energy-efficient than Mapping 2, that is

$$\frac{c_1 f_1 + c_2 f_2}{a_1 f_1 + a_2 f_2 + B} \geq \frac{c_2 f_1 + c_1 f_2}{a_2 f_1 + a_1 f_2 + B}$$

if and only if

$$\theta_1 \geq \theta_2$$

where

$$\theta_i = \frac{c_i}{a_i(f_1 + f_2) + B}. \quad (6.5)$$

Proof: Equation (6.5) is equivalent to

$$\begin{aligned} (c_1 f_1 + c_2 f_2)(a_2 f_1 + a_1 f_2 + B) &> (c_2 f_1 + c_1 f_2)(a_1 f_1 + a_2 f_2 + B) \\ c_1(f_1 a_2 f_1 + f_1 B - f_2 a_2 f_2 - f_2 B) &> \\ c_2(f_1 a_1 f_1 + f_1 B - f_2 a_1 f_2 - f_2 B) & \\ c_1(f_1 - f_2)(a_2(f_1 + f_2) + B) &> \\ c_2(f_1 - f_2)(a_1(f_1 + f_2) + B) & \end{aligned} \quad (6.6)$$

Because $f_1 > f_2$, (6.6) is equivalent to

$$\frac{c_1}{a_1(f_1 + f_2) + B} > \frac{c_2}{a_2(f_1 + f_2) + B} \quad (6.7)$$

□

From the above discussion, we conclude that, for a two-core system, mapping a VCPU with a higher energy-efficiency factor (*e.g.*, VCPU 1) on a fast core (*e.g.*, Core 1) leads to improved energy efficiency. Note that, however, this strategy may not always achieve the global optimal energy efficiency for the entire system when there are more than 2 cores in the system. However, it is important to make this simplification in order to avoid the unacceptable overhead of finding a global optimal

energy-efficient mapping. We further explain the overhead of a global optimal energy-efficient mapping in Section 6.3.

6.2.2 Online Estimation of energy-efficiency factor

As shown in Figure 6.2, our framework for estimating the energy-efficiency factor has offline and online parts. The offline part is performed with workloads in a *Training Set* and the online part is designed to provide estimations for the energy-efficiency factors for the runtime workloads which are typically unknown at design time.

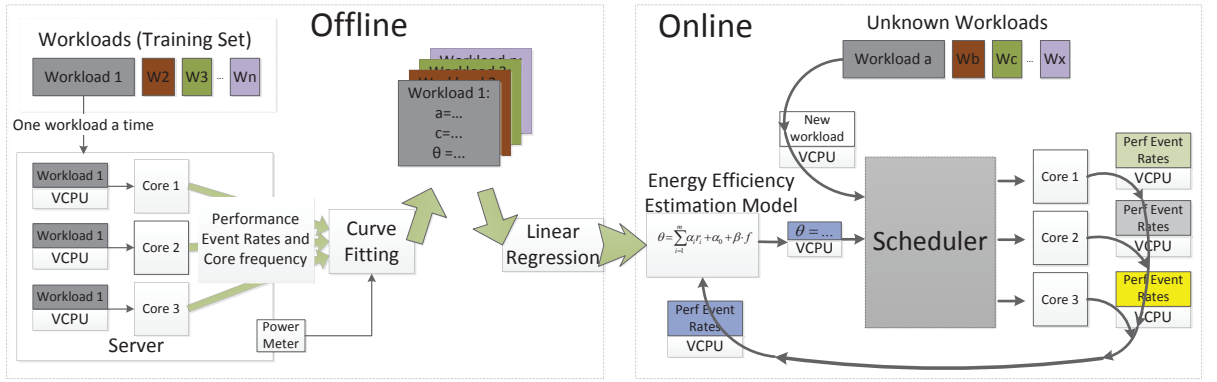


Figure 6.2: Proposed framework for estimating the energy-efficiency factor. The offline part derives an energy-efficiency model which is used online.

Offline Analysis on Workloads in a Training Set

The offline part provides an energy efficiency estimation model for the online part as follows.

We randomly select 8 benchmark in SPEC CPU 2006 as our training set and collect data for training as follows. For each benchmark, we perform several experiments. In each experiment we run 12 copies of the same benchmark in 12 VMs on the 12 cores of the system. All cores are set to the same frequency level in each experiment and we repeat the experiments using all frequency levels. In each experiment, we collect 22 performance events listed in Table 6.2 and measure the average power consumption using a power meter.

Table 6.2: Performance events measured

Dispatched FPU Operations	Retired SSE Operations
Retired Move Ops	Retired Serializing Ops
Segment Register Loads	Retired CLFLUSH Ins
Retired CPUID Ins	Data Cache Accesses
Data Cache Misses	Data Prefetcher
MAB Requests	Requests to L2 Cache
L2 Cache Misses	Ins Cache Fetches
Ins Cache Misses	Retired Instructions
Retired uops	Retired Branch Ins
Retired Mispredicted Branch Ins	Retired Taken Branch Ins
Interrupts-Masked Cycles	DRAM Accesses

Using the collected training data, for each VCPU, we measure its *performance event rates* (*i.e.*, performance event counts per instruction) and energy-efficiency factor. The energy-efficiency factor of each VCPU is calculated based on (6.5) where the variables are inferred using a standard curve fitting technique in the power and performance models, (6.2) and (6.4).

We then use linear regression to build a linear model to approximate the energy-efficiency factor of a VCPU

$$\theta = \sum_{i=1}^m \alpha_i r_i + \alpha_0 + \beta f \quad (6.8)$$

where r_i is the event rate of the i^{th} hardware performance event. α_i and β are the constant parameters derived from linear regression. The integer m is the number of performance event rates that are used in our model. f is the frequency of the core on which the VCPU is running.

To infer the relationship between the performance events and energy-efficiency factors, we first need to determine the value of m , in (6.8), for the system. With a greater m , *i.e.*, with the more different performance event rates considered, a better accuracy can generally be achieved. However, m is limited by the number of available

general-purpose performance counters in the hardware, because the model (6.8) is used online and a VCPU can only run on one core at a time. Note that during the collection of the training data in our offline analysis, the number of different performance event rates are not limited by the number of general-purpose performance counters per core, because different performance event rates can be measured by running identical VCPUs repeatedly on multiple cores. We use $m = 2$ in our model due to the hardware limitations in our testbed. The processor used in our testbed only provides 4 general-purpose performance counters for each core. Of these 4 counters, the first one needs to be dedicated to measuring the performance of the VCPU in BIPS to provide a reasonable evaluation of our algorithm. Another performance counter is dedicated to the event count of CPU-Clocks-Not-Halted, *i.e.*, the number of CPU clocks the VCPU has in a running state, such that the number of performance events counted by other performance counters can be converted to performance event rates. Only the remaining 2 counters can be used to count other performance events. We test all combinations of 2 among the 22 performance rate values of interest. The following equation shows the best accuracy among these combinations

$$\theta = \alpha_1 r_1 + \alpha_2 r_2 + \alpha_0 + \beta f_i \quad (6.9)$$

where r_1 is the rate of the Retired Taken Branch Instructions, r_2 is the rate of Interrupts-Masked Cycles, $\alpha_1 = 0.07762$, $\alpha_2 = -0.20914$, $\beta = 0.000398$, and $\alpha_0 = 0.016965$. The model fits the measured data with an R^2 value of 0.58.

To test the accuracy of the estimation in (6.9), we select 26,948 pairs from the traces of VCPUs running benchmarks in our training set. Each pair of trace contains the data acquired in our offline analysis for 2 benchmarks. We then compare the energy-efficiency factors of the two benchmarks derived from offline analysis with the energy-efficiency factors estimated in (6.9) using the performance event rates in the traces. Note that in our algorithm, the energy-efficiency factor is only used in comparing VCPUs. Thus, the estimation is not evaluated on how close it is to the real

values. Instead, we evaluate whether the estimation can provide a correct estimation regarding which VCPU has a higher energy-efficiency factor in a pair. In 87.2% of these 26,948 pairs, the estimations of energy-efficiency factor are correct.

Online Estimation for Unknown Workloads

When working online, we treat every VCPU as an unknown VCPU. Each time the VCPU changes from the running state to another state, we measure its performance event rates in the time slice it runs. We next estimate its energy-efficiency factor using the model in (6.9).

The energy-efficiency factor of a VCPU is estimated using the performance event rates in the *previous* time slice that the VCPU runs. Because the energy-efficiency factor of a VCPU varies from time to time as the applications running in the VCPU change phases, our algorithm should map the VCPUs based on their energy-efficiency factors in the next time slice. However, the difference between a VCPU’s energy-efficiency factor in the previous time slice and the next time slice is insignificant. A related study has shown that in a wide variety of workloads [37], using a last-value predictor to predict the behavior of an application on a time scale of tens of milliseconds provides an acceptable accuracy (*e.g.*, 1% error between 10ms samples [37]).

6.3 Energy-Efficient VM Scheduling

Based on the conclusion in Section 6.2.1, our goal is to dynamically keep the VCPUs with a greater energy-efficiency factor running on faster cores.

In this section, we first assume that the system only has two types of cores: fast cores and slow cores indicating which cores have higher or lower frequencies. We then introduce how to extend our technique to more frequency levels.

Algorithm 3 Energy-Efficient Load Balancing

Destination_Core: the core that is performing the current load balancing operation

Fast_Cores: list of fast cores; **Slow_Cores:** list of slow cores

Θ : bar of the energy-efficiency factors; **H:** The number of fast cores

Load_Balancing (Destination_Core)

begin

```
1:  $\Theta \leftarrow$  the  $H^{th}$  biggest in all running VCPUs' energy-efficiency factors
2: if Destination_Core is a fast core then
3:   for all  $Core_i$  in Slow_Cores do
4:     for all VCPUs  $VCPU_j$  in  $Core_i$ 's run queue do
5:        $\theta \leftarrow VCPU_j$ 's energy-efficiency factor
6:       if  $\theta > \Theta$  and  $VCPU_j$  has a high priority then
7:         return  $VCPU_j$ 
8:       end if
9:     end for
10:  end for{Failed to find a preferred VCPU in the system}
11:  Run normal load balancing
12: else
13:   {Destination_Core is a slow core}
14:   for all  $core_i$  in Fast_Cores do
15:     for all VCPUs  $VCPU_j$  in  $core_i$ 's run queue do
16:       if  $\theta < \Theta$  and  $VCPU_j$  has a high priority then
17:         return  $VCPU_j$ 
18:       end if
19:     end for
20:   end for{Failed to find a preferred VCPU in the system}
21:   Run normal load balancing
22: end if
end
```

6.3.1 Algorithm

Our algorithm inherits the majority of the Credit Scheduler algorithm and only uses a different strategy for load balancing. We do not move VCPUs among cores when the VCPUs are currently running because moving a running VCPU incurs a large overhead including context switches and cache warm up. Because load balancing operations only move VCPUs that are waiting for CPU cores to idle cores, the overhead, especially the context switch overhead, is significantly lower than moving running VCPUs.

Our load balancing operation is presented in Algorithm 1 and Figure 6.3. Before load balancing, a threshold of the energy-efficiency factor is calculated based on the running VCPUs. The threshold is defined as the H^{th} highest energy-efficiency factors among all the running VCPUs where H is the number of fast cores.

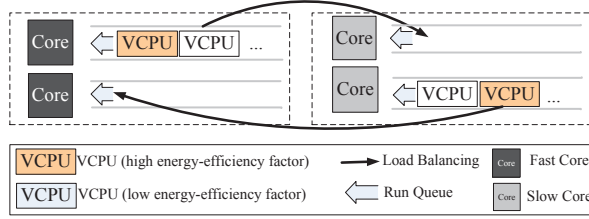


Figure 6.3: Energy-efficient load balancing. VCPUs with high energy-efficiency factors are dynamically moved to fast cores.

As presented in the pseudo-codes in Algorithm 1, when a slow core performs load balancing, *i.e.*, when it fails to find a high-priority VCPU on its local run queue, it searches for one on other cores, beginning with the fast cores. Preference is given to the VCPUs with an energy-efficiency factor lower than the threshold calculated before load balancing. Similarly, when a fast core performs load balancing, it searches the run queues of cores for a VCPU, starting with the slow cores. Preference is given to the VCPUs with an energy-efficiency factor higher than the threshold.

Note that our algorithm does not try to maintain an energy-efficient mapping for the currently running VCPUs due to the overhead of moving running VCPUs between cores. Instead, we attempt to move the VCPUs with high energy-efficiency factors

to the run queues of fast cores through the load balancing operations, which are frequently performed in the VM scheduler of Xen. Since the cores will run the VCPUs in their local run queues first whenever the priority allows, our algorithm drives the system toward a state such that VCPUs with high energy-efficiency factors run on fast cores.

6.3.2 Discussions

As discussed in Section 6.2, our strategy does not necessarily result in a VCPU-core mapping that has global maximum energy efficiency. Achieving a global maximum energy efficiency incurs a large overhead in practice for two reasons. First, every time when we need to apply a global optimal VCPU-core mapping to the system, many currently running VCPUs need to be migrated to new cores, which incurs a large overhead such as context switches and cache warm up. In addition, a global optimal solution may remain optimal only for a short period of time because the applications in the VMs frequent change execution phases with different performance and power behaviors. It has been shown in a study [38] that applications frequently switch between phases. For example, 17 of the 25 applications in SPEC CPU 2006 benchmarks [73] spend more than 70% of the their runtime in phases that last between 200 ms and 2000 ms [37]. Second, though it is feasible to derive a global optimal VCPU-core mapping strategy using algorithms such as integer programming, the algorithm may spend a significant amount of CPU time to find a global optimal VCPU-core mapping when the number of VCPUs and cores is large.

The algorithm presented in Section 6.3.1 uses only two frequency levels for simplicity. The algorithm can be extended to a system with more frequency levels by repeatedly performing the same algorithm between every two adjacent frequency levels. For example, when a core needs to retrieve a VCPU in load balancing, it first looks at the cores whose frequency is one level higher than its own, and tries to find a VCPU from those cores with a low energy-efficiency factor. If it fails to find such a

VCPU, it then looks at the cores whose frequency one level lower for a VCPU with a high energy-efficiency factor.

In our algorithm, the VCPUs with low energy-efficiency factors may be scheduled on slow cores the majority of the time, and thus may have degraded performance. In the systems where maintaining fairness is crucial in VM management, fairness can be maintained in two ways. First, the length of time slices allocated to the VCPUs running on slow cores can be extended. Second, the scheduling algorithm can be integrated with an application-level performance balancing algorithm. An example of integration is shown in the next subsection.

6.3.3 Integration with an Application-Level Performance Management Algorithm

We now discuss an example way of integrating the proposed VM scheduling algorithm with an application-level performance balancing solution as follows. We integrate our algorithm with a simple but effective application-level performance management solution that checks and balances the application-level progress of all VMs in every minute. Note that our algorithm is transparent to the application-level performance management algorithm. However, the application-level algorithm affects the scheduling decisions of our algorithm when adjusting the weight parameter in the management interface of the scheduler because the weight parameter determines the priority of the VCPUs. For the VMs with faster progress than the average, the management algorithm decreases the corresponding weight parameter by a level of 40. As discussed in Section 6.1.2, decreasing the weight parameter means to decrease the portion of CPU time the VM is allowed to consume. Likewise, for the VMs with slower progress than the average, the algorithm increases the corresponding weight parameters. Thus, this application-level management algorithm balances the progress of the VMs by shifting allocated CPU time from the VMs with a faster progress to

those with a slower progress to minimize the overall execution time of all applications. We evaluate the integration in an experiment in Section 6.5.

6.4 System Implementation

In this section, we introduce our physical testbed and benchmarks, as well as the implementation details of our algorithm in Xen hypervisor.

6.4.1 Testbed and Workloads

Our testbed is a Dell PowerEdge 2970 rack server equipped with two Six-Core AMD Opteron(tm) 2423 HE processors. Xen 4.0 is used as the virtual machine monitor.

We use three sets of workloads, SPEC CPU2006 [73], SPECjbb2005[74], and RUBBoS[40], in our experiments. SPEC CPU2006 includes a collection of 29 benchmarks and is divided into CINT2006 and CFP2006, which consist of integer and floating-point benchmarks, respectively. SPECjbb2005 is a benchmark for evaluating the performance of server side Java [74]. SPECjbb2005 emulates a 3-tier system, the most common type of the server-side Java applications. RUBBoS is a bulletin board benchmark modeled after an online news forum [40]. In our experiments, each VM runs a separate copy of a workload which is initiated by a master process in Domain-0 via a Secure Shell (SSH) connection. We randomly select 8 benchmarks in SPEC CPU2006 as our training set in Section 6.2.2, including *gcc*, *gobmk*, *bzip2*, *hammer*, *astar*, *h264ref*, *omnetpp* and *sjeng*.

6.4.2 Implementation in Xen Hypervisor

Our algorithm is implemented in Xen hypervisor. In virtualized systems powered by Xen, the hypervisor is an open source software layer that lives between the server’s hardware and the VMs’ operating systems. In our prototype system, the changes to the Xen’s source codes include three major parts: a set of virtual performance counters

which are used to count the performance events of VCPUs, a new implementation of the load balancing algorithm in Credit Scheduler, and modifications to the core frequency scaling interface.

Virtual Performance Counters: Hardware performance counters are only available in physical cores. In order to measure the performance event rates of the VCPUs and then use it to estimate the energy-efficiency factor defined in (6.5), we implement virtual performance counters in each VCPU. The virtual performance counters are a set of integer numbers associated with the VCPUs and are initialized to zeros. Before a VCPU runs on a core, the readings of the performance counters on the core are stored. After the VCPU is scheduled out of the core, the performance counters on the core are read again. The difference between the two readings is used to increment the virtual performance counters of the VCPU.

Load Balancing Algorithm: We implement the algorithm presented in Section 6.3 on top of the Credit Scheduler in Xen by rewriting the load balancing operation and implementing the estimation codes of the energy-efficiency factors.

Frequency Scaling: We adapt the frequency scaling interface of the cores into two steps. In the first step, the hypervisor of Xen is informed such that our algorithm in the scheduler tags the cores with the correct new frequency. In the second step, the frequencies of the cores are scaled using the XenPm interface.

Overall, our implementation is light-weight. From a memory perspective, our algorithm adds 40 bytes per VCPU and 16 bytes per core of the memory requirement of Credit Scheduler. The additional computations introduced by our algorithm over the Credit Scheduler are mostly used to maintain the virtual performance counters and estimate the energy-efficiency factor. Therefore, the additional overheads are negligible.

6.5 Evaluation

In this section, we compare the proposed energy-efficient algorithm with the default Credit Scheduler in Xen to demonstrate the improvements in energy efficiency.

6.5.1 Improvement in Energy Efficiency

In our first set of experiments, we select 2 out of the 8 benchmarks in our training set as different combinations to test heterogeneous workloads. We then do the same to 8 other benchmarks not in our training set to compose a different group of heterogeneous workloads. For each of the two benchmarks in a combination, we run 12 independent copies in 12 VMs (*i.e.*, 24 VMs in total). Each VM is allocated with one VCPU. The experiments are conducted in 4 different configurations of fast cores (2 GHz) and slow cores (0.8 GHz).

Figure 6.4 presents the energy efficiency (*i.e.*, performance per watt) of the proposed algorithm relative to the energy efficiency of the default Credit Scheduler. In 54 out of the 64 experiments conducted to the benchmarks in our training set, as shown in Figure 6.4 (left side), the proposed algorithm outperforms the baseline, *i.e.*, the relative energy efficiency is greater than 100%. On average, the proposed algorithm improves the energy efficiency by 13.5%. In the other 10 experiments, the proposed algorithm shows a worse energy efficiency than the default Credit Scheduler. Our algorithm does not always result in better energy efficiency in all cases due to several reasons. First, our model to estimate the energy-efficiency factor in (6.9) may be inaccurate in some cases. Mapping VCPUs to cores based on an inaccurate estimation of the energy-efficiency factor may result in a degraded energy efficiency. Second, because our algorithm only moves VCPUs between run queues of different cores during load balancing operations, the algorithm may not always have a chance to switch to an energy-efficient VCPU-core mapping because load balancing operations are not always performed.

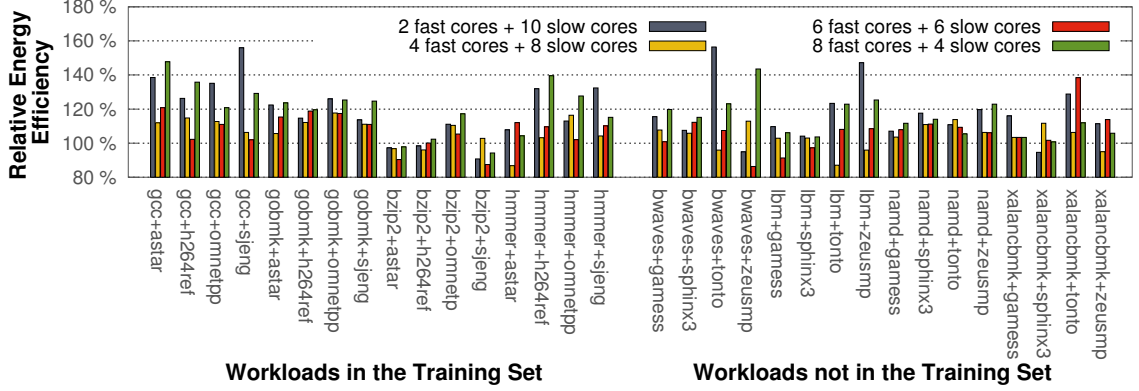


Figure 6.4: Relative energy efficiency of our energy-efficient algorithm and the default Xen Credit Scheduler in 4 sets of core frequency settings.

Similarly, in 56 out of the 64 experiments conducted to the benchmarks not in our training set, the proposed algorithm outperforms the default Credit Scheduler. On average, our algorithm improves the energy efficiency by 10.2%. This set of experiments shows that, although our algorithm requires building a model using the workloads in a training set, the algorithm works with other workloads without rebuilding a model.

In Figure 6.5, we randomly select 24 benchmarks from SPEC CPU 2006 and run them in our 24 VMs. In 4 experiments, we run 2, 4, 6 and 8 out of the 12 total cores as fast cores, respectively. In all these 4 experiments, our algorithm shows a better energy efficiency compared with the default Credit Scheduler. The average improvement is 7.3%.

Because virtualized platforms are often used in web hosting applications, we evaluate the proposed algorithm in server workloads: a PHP implementation of RUBBoS and a java-based benchmark SPECjbb2005. We build 9 VMs. Each VM is allocated with 6 VCPUs. The first two VMs run SPECjbb2005 and the other VMs run RUBBoS. In 4 experiments, we run 2, 4, 6 and 8 out of the 12 total cores as fast cores, respectively. Figure 6.6 shows the energy efficiency of the proposed algorithm relative to the default Credit Scheduler. On average, the proposed algorithm improves the energy efficiency by 9.5%.

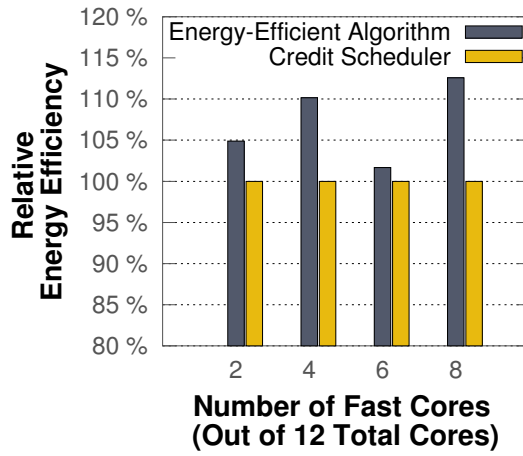


Figure 6.5: Relative energy efficiency of proposed energy-efficient algorithm and the default Credit Scheduler running a mix of 24 different benchmarks.

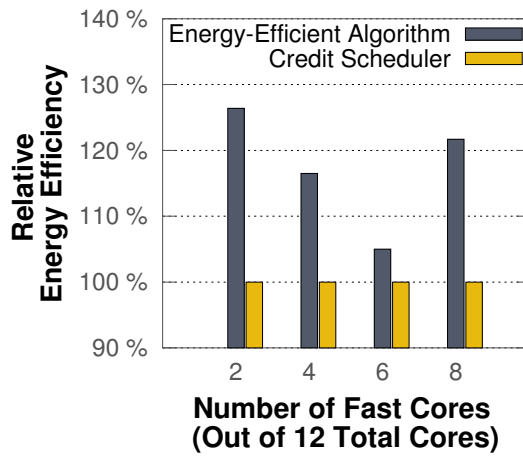


Figure 6.6: Relative energy efficiency of proposed energy-efficient algorithm (relative to that of the default Credit Scheduler) running web workloads (RUBBoS and SPECjbb).

One may think that the proposed energy-efficient algorithm improves energy efficiency at the cost of degraded system performance. To examine the impacts on performance, we plot the relative application-level performance of RUBBoS and SPECjbb2005 in Figure 6.7. On average, the performance of the proposed energy-efficient algorithm (relative to the Xen Credit Scheduler) is 100.2% for RUBBoS and 97.9% for SPECjbb2005, respectively. Therefore, the proposed algorithm improves the system energy efficiency with negligible impacts on the application-level performance.

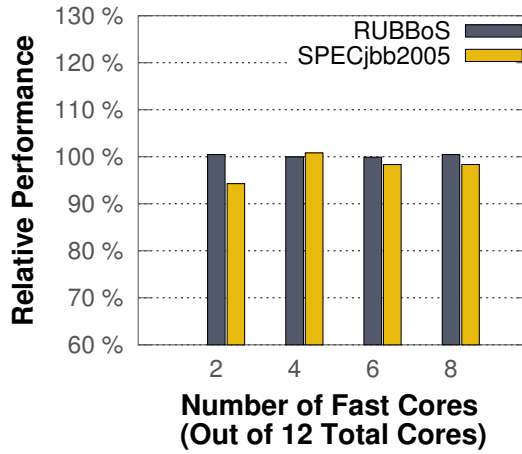


Figure 6.7: The performance of the proposed energy-efficient algorithm and the default Credit Scheduler running web workloads (RUBBoS and SPECjbb).

6.5.2 Integration with an Application-level Performance Management Algorithm

In this experiment, we test the example of integrating the proposed energy-efficient algorithm with an application-level performance management algorithm presented in Section 6.3.3. We build 4 VMs, each allocated with 9 VCPUs, and run SPECjbb in 2 of them and run RUBBoS in the other 2 VMs.

We test our system in a scenario where every application needs to finish a set of tasks and each task contains 1 million transactions. This experiment

represents a typical scenario in scientific computing applications, where several related computation tasks are submitted to a server and the performance of the system is determined by the time when all the tasks are completed.

Figure 6.8 compares the application-level performance (in throughput) and energy consumption of the proposed algorithm against the default Credit Scheduler. In all these experiments, our algorithm shows a higher throughput (4.1% on average) and lower energy consumption (6.3% on average). During the entire execution time of the applications, the scheduling algorithm moves the VCPUs among the cores to maximize the overall energy efficiency in every load balancing operation, hence minimizing the overall energy consumption required to execute the set of VCPUs with a predetermined number of instructions. In contrast, the Credit Scheduler moves VCPUs among VMs without energy efficiency considerations.

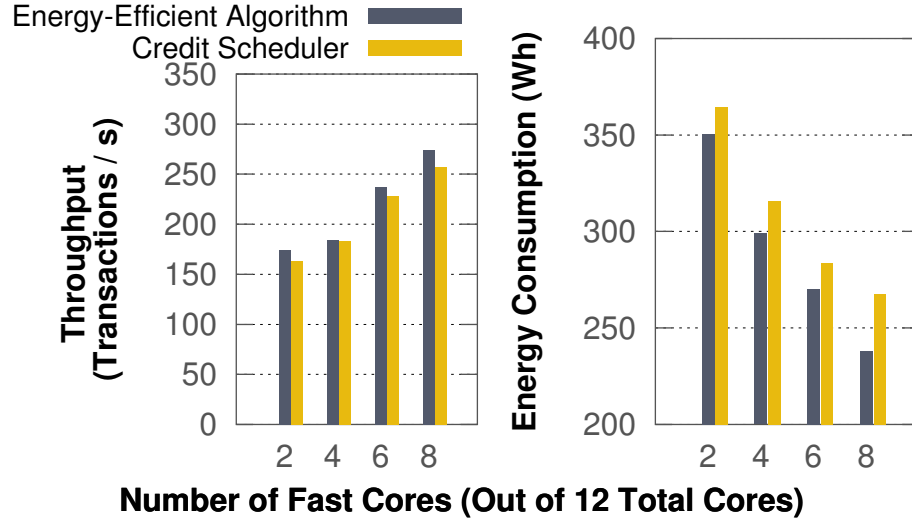


Figure 6.8: Throughput (average and standard deviation) and energy consumption for 1 million request per application.

6.5.3 Homogeneous Workloads

As discussed in Sections 6.3, the improvement in energy efficiency in our algorithm is due to its ability to move VCPUs between different types of cores based on the

differences in the energy-efficiency factors of the VCPUs. Thus, if all VCPUs in the system are similar, our algorithm is less effective in improving the energy efficiency. To test this, we run homogeneous workloads in the 24 VMs in our system and plot the results in Figure 6.9. As expected, the differences between our algorithm and the default Credit Scheduler is much smaller than the results presented in Section 6.5.1. In these experiments, our algorithm only improves the energy efficiency by 3.2%. Note that when all the VMs run the same workload, the VCPUs are not strictly homogeneous due to the phase changes in the applications running inside the VCPUs.

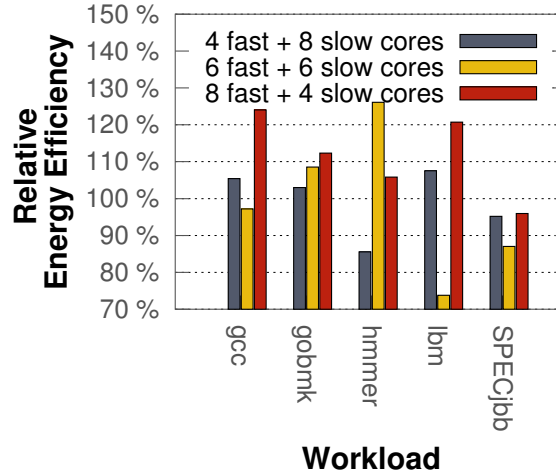


Figure 6.9: Relative energy efficiency of the energy-efficient algorithm and the default Credit Scheduler running homogeneous workloads.

6.6 Summary

The heterogeneity in multi-core architectures can be beneficial to today’s high-performance computer servers that commonly host virtual machines (VMs) on multi-core architectures, as applications running in different VMs often have different performance and power characteristics. Therefore, it is important to exploit the

performance heterogeneity of different cores to schedule VMs in a more energy-efficient way. However, existing VM managers, such as Xen, schedule VMs regardless of the heterogeneity and thus often result in low energy efficiencies.

In this dissertation, we have presented a novel algorithm that schedules VMs on multi-core architectures with performance heterogeneity to optimize the overall system energy efficiency. We first introduce a metric called energy-efficiency factor to characterize the power and performance behaviors of the application of each VM on different cores. Our scheduling algorithm dynamically estimates the VMs' energy-efficiency factors based on available hardware performance counters and then maps the VMs to the cores, such that the energy efficiency of the entire system is maximized in a light-weight way. We implement the proposed scheduling algorithm in Xen and evaluate it by testing various standard benchmarks on a hardware testbed. Extensive results with various standard benchmarks, such as SPEC CPU2006, SPECjbb, and RUBBoS show that our solution can improve the energy efficiency, *e.g.*, by 13.5% on average and up to 55% for some benchmarks, compared to the default Credit Scheduler of Xen. We also demonstrate that the proposed scheduling algorithm can be integrated with an application-level performance management algorithm and to achieve 6.3% average energy savings.

Chapter 7

Conclusion

Modern data centers must provide performance assurance for complex system software such as web applications. In addition, the power consumption of data centers needs to be minimized to reduce operating costs and avoid system overheating. In recent years, more and more data centers start to adopt server virtualization strategies for resource sharing to reduce hardware and operating costs by consolidating applications previously running on multiple physical servers onto a single physical server. In this dissertation proposal, four power efficient algorithms are proposed to effectively reduce server power consumption while achieving required application-level performance for virtualized enterprise servers.

First, this dissertation proposes a two-layer control architecture based on dynamic voltage and frequency scaling (DVFS). The primary control loop adopts a multi-input-multi-output control approach to maintain load balancing among all virtual machines so that they can have approximately the same performance level relative to their allowed peak values. The secondary performance control loop then manipulates CPU frequency for power efficiency based on the uniform performance level achieved by the primary loop. Empirical results demonstrate that our control solution can effectively reduce server power consumption while achieving required application-level performance for virtualized enterprise servers.

Second, DVFS-based solutions cannot further reduce the energy consumption of a server when the server processor is already at the lowest DVFS level and the server utilization is still low (*e.g.*, 5% or lower). This dissertation proposes Virtual Batching, a novel request batching solution for virtualized servers with primarily light workloads. Based on the performance of the VMs, Virtual Batching determines the time length for periodically batching incoming requests and putting the processor into sleep. When the workload intensity changes from light to moderate, request batching is automatically switched to DVFS to increase processor frequency for performance guarantees.

Third, this dissertation proposes a performance-controlled power optimization solution for virtualized server clusters with multi-tier applications. This solution utilizes both DVFS and server consolidation strategies for maximized power savings by integrating feedback control with optimization strategies. At the application level, a multi-input-multi-output controller is designed to achieve the desired performance for applications spanning multiple VMs, on a short time scale, by reallocating the CPU resources and DVFS. At the cluster level, a power optimizer is proposed to incrementally consolidate VMs onto the most power-efficient servers on a longer time scale.

Finally, this dissertation proposes a VM scheduling algorithm that exploits core performance heterogeneity to optimize the overall system energy efficiency.

These four algorithms are proposed to be demonstrated in empirical results on hardware testbeds and trace-driven simulations and compared against state-of-the-art baselines.

Bibliography

Bibliography

- [1] About ideas relative performance estimate 2 (RPE2).
<http://www.ideasinternational.com/performance/>. 83
- [2] Credit Scheduler. <http://wiki.xensource.com/xenwiki/CreditScheduler>. 29, 64, 108
- [3] Traces available in the Internet Traffic Archive.
<http://ita.ee.lbl.gov/html/traces.html>. 68
- [4] Advanced Micro Devices. AMD Family 10h Server and Workstation Processor Power and Thermal Data Sheet, 2010. 62
- [5] Amazon. Amazon elastic mapreduce. <http://aws.amazon.com/elasticmapreduce/>, 2010. 106
- [6] AMD. Quad-core AMD opteron processors for servers and workstations. Technical report, AMD, 2008. 7
- [7] AMD. *White Paper Publication 31116: BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*. Advanced Micro Devices, Inc., Apr. 2010. 110
- [8] L. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40:33–37, 2007. 3

- [9] L. Bertini, J. Leite, and D. Mosse. SISO PIDF controller in an energy-efficient multi-tier web server cluster for e-commerce. In *FeBED*, 2007. [13](#)
- [10] A. Caprara and U. Pferschy. Worst-case analysis of the subset sum algorithm for bin packing. *Operations Research Letters*, 32(2), 2004. [14](#)
- [11] J. Chen and L. K. John. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 927–930, New York, NY, USA, 2009. ACM. [7](#), [16](#)
- [12] J.-J. Chen, H.-R. Hsu, K.-H. Chuang, C.-L. Yang, A.-C. Pang, and T.-W. Kuo. Multiprocessor energy-efficient scheduling with task migration considerations. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, 2004. [10](#)
- [13] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing server energy and operational costs in hosting centers. In *ACM SIGMETRICS*, 2005. [11](#)
- [14] Y. Chen, S. Iyer, X. Liu, D. Milojevic, and A. Sahai. SLA decomposition: Translating service level objectives to system level thresholds. In *ICAC*, 2007. [12](#), [13](#)
- [15] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007. [6](#), [15](#)
- [16] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005. [4](#), [75](#)
- [17] DOE. Electric power monthly, October 2009. [3](#)
- [18] P. Dorato, C. T. Abdallah, and V. Cerone. *Linear Quadratic Control: An Introduction*. Prentice Hall, 1998. [21](#), [23](#)

- [19] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003. [1](#)
- [20] Electronic Educational Devices Inc. Watts Up Pro Power Meter. <http://www.wattsupmeters.com>. [92](#)
- [21] E. M. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *PACS*, 2002. [86](#)
- [22] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *USENIX*, 2003. [11](#), [63](#)
- [23] K. Fleszar and K. S. Hindi. New heuristics for one-dimensional bin-packing. *Comput. Oper. Res.*, 29(7), 2002. [14](#), [87](#), [88](#)
- [24] G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems*, 3rd edition. Addition-Wesley, 1997. [21](#), [27](#), [79](#)
- [25] GoDaddy.com, Inc. Virtual Dedicated Servers - Highly Configurable Plans Low Prices. ”<http://www.godaddy.com/Hosting/virtual-dedicated-servers.aspx>”, 2010. [55](#)
- [26] T. F. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics*. Chapman, 2008. [14](#)
- [27] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE ’07, pages 126–136, New York, NY, USA, 2007. ACM. [6](#), [15](#)
- [28] V. Gupta and R. Nathuji. Analyzing performance asymmetric multicore processors for latency sensitive datacenter applications. In *Proceedings of the 2010 international conference on Power aware computing and systems*,

- HotPower'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association. 6, 15
- [29] M. Haouari and M. Serairi. Heuristics for the variable sized bin-packing problem. *Comput. Oper. Res.*, 36(10), 2009. 14
 - [30] J. Heo, D. Henriksson, X. Liu, and T. Abdelzaher. Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study. In *RTSS*, 2007. 13
 - [31] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE*, 2009. 4, 75
 - [32] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. Dynamic voltage scaling in multi-tier web servers with end-to-end delay control. *IEEE Transactions on Computers*, 56(4), 2007. 11
 - [33] C. Hyser, B. McKee, R. Gardner, and B. J. Watson. Autonomic virtual machine placement in the data center, Tech Report, HP Labs. <http://www.hpl.hp.com/techreports/2007/HPL-2007-189.pdf>, 2008. 13
 - [34] IDEAS. About IDEAS relative performance estimate 2 (RPE2). "<http://www.ideasinternational.com/performance/>". 58
 - [35] Intel. Intel 5500 series datasheet vol. 1. 2009. 3
 - [36] Intel Labs. *The SCC Platform Overview Revision 0.7*. Intel Labs, 2010. 7
 - [37] C. Isci, A. Buyuktosunoglu, and M. Martonosi. Long-term workload phases: Duration predictions and applications to dvfs. volume 25, pages 39–51, Los Alamitos, CA, USA, September 2005. IEEE Computer Society Press. 117, 120
 - [38] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management.

- In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society. [120](#)
- [39] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society. [110](#)
- [40] Julie Marguerite and Emmanuel Cecchet. RUBBoS: Rice University Bulletin Board System. Rice University, <http://www.cs.rice.edu/CS/Systems/DynaServer/RUBBoS/>. [90](#), [122](#)
- [41] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management in virtualized server environments. In *NOMS*, 2006. [12](#), [13](#), [14](#), [86](#)
- [42] D. Kim, H. Kim, M. Jeon, E. Seo, and J. Lee. Guest-aware priority-based virtual machine scheduling for highly consolidated server. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 285–294, Berlin, Heidelberg, 2008. Springer-Verlag. [6](#), [15](#)
- [43] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM. [6](#), [16](#)
- [44] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. *Microarchitecture, IEEE/ACM International Symposium on*, 0:81, 2003. [16](#)

- [45] V. Kumar and A. Fedorova. Towards better performance per watt in virtual environments on asymmetric single-isa multi-core systems. *SIGOPS Oper. Syst. Rev.*, 43:105–109, July 2009. [6](#), [16](#)
- [46] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. In *ICAC*, 2008. [13](#), [14](#)
- [47] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12), 2003. [13](#)
- [48] C. Lefurgy, X. Wang, and M. Ware. Server-level power control. In *ICAC*, 2007. [3](#), [11](#), [30](#), [83](#)
- [49] F. L. Lewis and V. L. Syrmos. *Optimal Control, Second Edition*. John Wiley & Sons, Inc., 1995. [82](#)
- [50] X. Liu, X. Zhu, S. Singhal, and M. Arlitt. Adaptive entitlement control of resource containers on shared servers. In *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management*, Nice, France, 2005. [10](#)
- [51] L. Ljung. *System Identification: Theory for the User (2nd Edition)*. Prentice Hall PTR, December 1998. [47](#), [52](#)
- [52] J. M. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, 2002. [80](#), [82](#)
- [53] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: Eliminating server idle power. In *ASPLOS*, 2009. [45](#), [60](#)
- [54] K. Meng, R. Joseph, R. P. Dick, and L. Shang. Multi-optimization power management for chip multiprocessors. In *Proceedings of the 17th international*

- conference on Parallel architectures and compilation techniques*, PACT '08, pages 177–186, New York, NY, USA, 2008. ACM. [7](#), [107](#)
- [55] R. J. Minerick, V. W. Freeh, and P. M. Kogge. Dynamic power management using feedback. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP)*, Sep. 2002. [11](#)
 - [56] R. Nathuji, P. England, P. Sharma, and A. Singh. Feedback driven qos-aware power budgeting for virtualized servers. In *FeBID*, 2009. [12](#), [13](#)
 - [57] R. Nathuji and K. Schwan. VirtualPower: coordinated power management in virtualized enterprise systems. *SIGOPS Oper. Syst. Rev.*, 41(6), 2007. [4](#), [12](#), [76](#)
 - [58] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 1–10, New York, NY, USA, 2008. ACM. [6](#), [15](#)
 - [59] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys*, 2009. [12](#), [13](#)
 - [60] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007. [2](#), [10](#), [12](#), [13](#)
 - [61] M. Powell, A. Biswas, J. Emer, S. Mukherjee, B. Sheikh, and S. Yardi. Camp: A technique to estimate per-structure power at run-time using a few simple parameters. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 289–300. IEEE, 2009. [110](#)
 - [62] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No power struggles: Coordinated multi-level power management for the data center. In *ASPLOS*, 2008. [11](#), [12](#), [14](#)

- [63] K. Rajamani and C. Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. In *ISPASS*, 2003. [13](#)
- [64] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. volume 37, pages 302–313, New York, NY, USA, June 2009. ACM. [7](#)
- [65] P. Ranganathan, P. Leech, D. Irwin, and J. S. Chase. Ensemble-level power management for dense blade servers. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2006. [11](#)
- [66] P. Rong and M. Pedram. Battery-aware power management based on markovian decision processes. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, ICCAD '02*, pages 707–713, New York, NY, USA, 2002. ACM. [15](#)
- [67] E. Rotem, A. Mendelson, R. Ginosar, and U. Weiser. Multiple clock and voltage domains for chip multi processors. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42*, pages 459–468, New York, NY, USA, 2009. ACM. [110](#)
- [68] W. J. Rugh. *Linear System Theory*. Prentice Hall, Aug 1995. [50](#)
- [69] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems, EuroSys '10*, pages 139–152, New York, NY, USA, 2010. ACM. [6](#), [16](#)
- [70] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu. Power-aware QoS management in web servers. In *IEEE Real-Time Systems Symposium (RTSS)*, 2003. [11](#)

- [71] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43:66–75, April 2009. [6](#), [16](#)
- [72] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2002. [11](#)
- [73] C. D. Spradling. Spec cpu2006 benchmark tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, 2007. [109](#), [120](#), [122](#)
- [74] Standard Performance Evaluation Corporation. SPECjbb2005, a new Java Server Benchmark. <http://www.spec.org/jbb2005/docs/WhitePaper.html>. [122](#)
- [75] Standard Performance Evaluation Corporation. All Published SPECpower ssj2008 Results. ”http://www.spec.org/power_ssj2008/results/power_ssj2008.html”, 2010. [62](#), [93](#)
- [76] R. T. Stefani. *Gain margin and phase margin*, pages 152–157. The control handbook. CRC Press, 1999. [51](#)
- [77] G. Tao. *Adaptive control design and analysis*. Adaptive and learning systems for signal processing, communications, and control. Wiley-Interscience, 2003. [52](#)
- [78] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. volume 36, pages 363–374, New York, NY, USA, June 2008. ACM. [7](#), [111](#)
- [79] A. Verma, P. Ahuja, and A. Neogi. pmapper: Power and migration cost aware application placement in virtualized systems. In *Middleware*, 2008. [9](#), [12](#), [14](#), [55](#), [58](#), [71](#), [93](#), [95](#)

- [80] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *USENIX*, 2009. [12](#), [14](#), [58](#), [84](#), [85](#)
- [81] VPSLink. Cheap VPS Hosting Plans. "http://vpslink.com/vps-hosting", 2010. [55](#), [56](#)
- [82] L. Wang and Y. Lu. Efficient power management of heterogeneous soft real-time clusters. In *RTSS*, 2008. [14](#)
- [83] X. Wang and M. Chen. Cluster-level feedback power control for performance optimization. In *14th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2008)*, 2008. [11](#), [13](#), [86](#)
- [84] X. Wang, M. Chen, C. Lefurgy, and T. W. Keller. SHIP: Scalable hierarchical power control for large-scale data centers. In *PACT*, 2009. [13](#), [62](#), [93](#), [107](#)
- [85] X. Wang and Y. Wang. Co-con: Coordinated control of power and application performance for virtualized server clusters. In *IWQoS*, 2009. [13](#), [47](#)
- [86] Y. Wang, K. Ma, and X. Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 314–324, New York, NY, USA, 2009. ACM. [7](#), [110](#)
- [87] Z. Wang, X. Zhu, and S. Singhal. Utilization and SLO-based control for dynamic sizing of resource partitions. In *16th IFIP/IEEE International Workshop on Distributed Systems (DSOM)*, 2005. [10](#)
- [88] G. Wei, J. Liu, J. Xu, G. Lu, K. Yu, and K. Tian. The on-going evolutions of power management in xen. In *Xen Summit*, 2009. [6](#), [15](#)
- [89] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. In *ICAC*, 2007. [12](#), [13](#)

- [90] W. Xu, X. Zhu, S. Singhal, and Z. Wang. Predictive control for dynamic resource allocation in enterprise data centers. In *IEEE/IFIP Network Operations and Management Symposium*, 2006. [10](#)
- [91] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proceedings of the 1st International Conference on Virtual Execution Environments*, 2005. [10](#)
- [92] X. Zhu, D. Young, B. J. Watson, J. Rolia, S. Singhal, B. Mckee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 islands: An integrated approach to resource management for virtualized data centers. *Cluster Computing*, 12:45–47, 2009. [12](#), [13](#)
- [93] Y. Zhu and F. Mueller. Feedback EDF scheduling exploiting dynamic voltage scaling. In *IEEE Real-Time and Embedded Technology and Applications Symposium(RTAS)*, 2004. [11](#)

Vita

Yefu Wang was born in Nong'an, Jilin Province, China. He received the M.S. degree in Control Science and Engineering in 2006, B.S. in Finance in 2004, B.S. in Automation in 2003, all from Harbin Institute of Technology, China. Before began his Ph.D study in University of Tennessee, Knoxville, he worked at Beijing Suwell Co. Ltd., China. He received the University of Tennessee Chancellor's Honor for Extraordinary Professional Promise in 2010. He Joined Siemens Healthcare in Aug. 2010.