8-2010

Adaptive Performance and Power Management in Distributed Computing Systems

Ming Chen
mchen11@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Ming Chen entitled "Adaptive Performance and Power Management in Distributed Computing Systems." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Xiaorui Wang, Major Professor

We have read this dissertation and recommend its acceptance:

Seddik M. Djouadi, Gregory D. Peterson, Xueping Li

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:
I am submitting herewith a dissertation written by Ming Chen entitled "Adaptive Performance and Power Management in Distributed Computing Systems." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Xiaorui Wang, Major Professor

We have read this dissertation
and recommend its acceptance:

Gregory D. Peterson

Seddik M. Djouadi

Xueping Li

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Adaptive Performance and Power Management in Distributed Computing Systems

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Ming Chen

August 2010

Acknowledgment

Foremost, I would like to express my sincere gratitude to my advisor Dr. Xiaorui Wang for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and a tremendous amount of support during my four-year Ph.D study and research at the University of Tennessee. His guidance helped me in all the time of research. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I also would like to give my sincere thanks to Dr. Hui Zhang at NEC Laboratory America for his very patient guidance on the projects during my total five-month internship at NEC.

My sincere thanks also go to the rest of thesis committee: Dr. Gregory Peterson, Dr. Seddik M. Djouadi, Dr. Xueping Li, for their patience and insightful suggestion for my thesis.

It is a honor for me to have worked with my fellow labmates in the research group: Yefu Wang, Nitish Jha, Xiaodong Wang, Kai Ma, Xing Fu, Xue Li, and Ben Taylor. Thank all of you for the stimulating discussions in many projects.

Finally, I would like to thank my parents, for providing a strong foundation and a desire to improve and grow everyday. I would not be where I am today without their love.

Abstract

The complexity of distributed computing systems has raised two unprecedented challenges for system management. First, various customers need to be assured by meeting their required service-level agreements such as response time and throughput. Second, system power consumption must be controlled in order to avoid system failures caused by power capacity overload or system overheating due to increasingly high server density. However, most existing work, unfortunately, either relies on open-loop estimations based on off-line profiled system models, or evolves in a more ad hoc fashion, which requires exhaustive iterations of tuning and testing, or oversimplifies the problem by ignoring the coupling between different system characteristics (*i.e.*, response time and throughput, power consumption of different servers). As a result, the majority of previous work lacks rigorous guarantees on the performance and power consumption for computing systems, and may result in degraded overall system performance. In this thesis, we extensively study adaptive performance/power management and power-efficient performance management for distributed computing systems such as information dissemination systems, power grid management systems, and data centers, by proposing Multiple-Input-Multiple-Output (MIMO) control and hierarchical designs based on feedback control theory. For adaptive performance management, we design an integrated solution that controls both the average response time and CPU utilization in information dissemination systems to achieve bounded response time for high-priority information and maximized system throughput in an example information dissemination system. In addition, we design a hierarchical control solution to guarantee the deadlines of real-time tasks in power grid computing by grouping them based on their characteristics, respectively. For adaptive power management, we design

MIMO optimal control solutions for power control at the cluster and server level and a hierarchical solution for large-scale data centers. Our MIMO control design can capture the coupling among different system characteristics, while our hierarchical design can coordinate controllers at different levels. For power-efficient performance management, we discuss a two-layer coordinated management solution for virtualized data centers. Experimental results in both physical testbeds and simulations demonstrate that all the solutions outperform state-of-the-art management schemes by significantly improving overall system performance.

Contents

List of Figures

Chapter 1

Introduction

Recent years have seen a rapid growth of distributed computing systems that provide outsourced business-critical IT services. Two key challenges exist for effectively operating those computing systems. First, service owners need to be assured by meeting their required Service-Level Objectives (SLOs), such as response time and throughput. For example, buyers and sellers need to be matched based on their interests in e-commerce systems and notified immediately when new business opportunities are identified. Likewise, in surveillance applications, threats detected by various sensors must be reported to appropriate authorities within certain time constraints. Second, power consumption has to be capped within the capacity of the power supplies and cooling facilities in order to reduce operating costs and avoid system failures caused by power capacity overload or system overheating. For example, if the physical power limits are violated, the overloading of electrical circuits may cause circuit breakers to trip, resulting in undesired outages. Even though computing systems commonly rely on power provisioning, the actual power consumption of the IT equipment may still exceed the power distribution capacity of the facility. A real scenario that many computing facilities face is that business needs require deploying new servers rapidly while upgrades of the power and cooling systems lag far behind. In some regions, it is either impossible or cost-prohibitive to provide additional power from the utility company. Therefore, besides performance guarantees, it is important to control the power consumption of computing systems.

However, both performance and power management in distributed computing systems face several major challenges. First, multiple system characteristics need to be simultaneously considered to achieve the performance requirements and control the power consumption. For example, to meet the requirement of average response time is not the only performance requirement. Most applications are also expected to achieve maximized system throughput as well, so that more customer services can be provided. Similarly, power management in computing systems usually involves multiple components (*i.e.*, processor and main memory) at the server level and multiple servers at the rack level. As a result, we need Multi-Input-Multi-Output (MIMO) strategies to coordinate the multiple system characteristics to maximize overall system performance. Second, different system characteristics are usually heterogeneous. For example, in power grid computing systems, real-time tasks usually have significantly different periods and execution times. Therefore, it is difficult to apply a single control algorithm to guarantee the deadline of all the tasks simultaneously. Similarly, power is distributed level-by-level in data centers that hold thousands of computing systems. Power budgets at different levels of computing systems need to be allocated based on different power demands. Instead of using a single-level power management scheme, the power allocation needs to be handled by multi-level management schemes. As a result, more complicated strategies, featuring a hierarchical design, are required to address the heterogeneity. Third, different system characteristics are usually coupled. For example, increasing the throughput of an information dissemination system may result in requirement violation of average response time. Increasing the task rates in one server in a distributed computing system may increase the rates of other subtasks in other servers, so that the schedulibility may be impaired. As a result, the coupling among different system characteristics should be carefully addressed for both performance and power management. Fourth and most importantly, most distributed computing systems work in open and unpredictable environments, with workloads that may vary significantly at runtime. The worst-case scenario in such environments is not known a priori or is very pessimistic, rendering traditional worst-case analysis impractical. As a result, both performance and

power management algorithms cannot rely on open-loop estimations. They must be self-adaptive to workload variations for optimal overall system performance.

In recent years, various performance and power management solutions have been proposed for distributed computing systems. Most existing work, unfortunately, falls into the following three categories:

1. Open-loop estimation. Open-loop estimation does not explicitly measure system performance metrics (*i.e.*, response time) or system power consumption, but relies instead on estimated control input based on off-line profiled system models. It can guarantee the desired performance requirement or the desired power budget when the system characteristics can be accurately captured by the system models. However, when the system varies from the model, it may result in performance violation and power overloading.

2. Ad Hoc fashion. Ad Hoc is a heuristic-based adaptive controller that has been widely used in performance and power management for distributed computing systems. To design an effective Ad Hoc control scheme requires exhaustive iterations of tuning and testing by using nominal system workload. When the system workload differs from the nominal workload, the experiential parameters in the Ad Hoc controllers need to be re-tuned and re-tested.

3. Single-Input-Single-Output (SISO) strategies. SISO strategies usually oversimplify the problems in performance and power management by ignoring the coupling between different system characteristics (*i.e.*, response time and throughput, power consumption of different servers). Even though multiple SISO controllers may be designed, the SISO strategy still lacks coordination among multiple controllers.

As a result, the above strategies either lack rigorous guarantees on performance and power consumption for computing systems, or degrade the overall system performance significantly.

Recently, control theory has been identified as a promising theoretical foundation for both performance and power control in complex computing systems, in that it provides (1) standard approaches to choosing the right control parameters so that exhaustive iterations

of tuning and testing are avoided; (2) theoretically guaranteed control performance such as accuracy, stability, short settling time, and small overshoot; and (3) quantitative control analysis when the system is suffering unpredictable workload variations. This rigorous design methodology is in sharp contrast to heuristic-based adaptive solutions that rely on extensive empirical evaluation and manual tuning. More importantly, the control-based design can tolerate significant system model variations, so that even when the system model at runtime varies from the nominal model at the design time within a theoretically derivable range, the controlled variables can still be guaranteed to converge to the desired set points. Especially, the Multiple Input Multiple Output (MIMO) optimal control design and the hierarchical control design can significantly improve the effectiveness of those control-based method in performance and power management for computing systems. For example, the MIMO design can capture the coupling among different system characteristics so that optimal control inputs can be found by solving optimization problems. The hierarchical design can organize multiple controllers in distributed computing systems into two or more levels so that controllers in different levels can be coordinated by sending control signals to the controllers at the level below and feedback or sensing signals to the controllers at the level above.

In this thesis, I investigate the application of control-based methodology in computing systems for adaptive performance and power management, respectively. I propose management strategies that feature both MIMO optimal control design to address the coupling among heterogeneous system characteristics involved in performance and power management, and hierarchical control design to coordinate controllers at different levels. Experimental results in either physical testbeds or simulations demonstrate that our solutions outperform state-of-the-art performance and power management solutions for distributed computing systems by significantly improving overall system performance. Specifically, the following work has been done:

- Adaptive performance management. I have verified an integrated solution that controls both the average response time and CPU utilization in information dissemination systems to achieve bounded response time for high-priority information and maximized system throughput in an example information dissemination system [15][16], and a

hierarchical control solution to guarantee the deadlines of real-time tasks in power grid computing by grouping them based on their characteristics [14].

- Adaptive power management. I have verified both MIMO optimal control solutions at the cluster level [84] and the server level, and a hierarchical design for large-scale data centers [85] that hold thousands of computing systems. The hierarchical design for power management is consistent with the power distribution hierarchy of computing systems in large-scale data centers (*i.e.*, the data center-level, PDU-level, and rack-level). Our proposed design can capture the coupling among different servers, ranks, and PDUs at each level so that the power budget can be optimally shifted at each individual level.

- Power-efficient performance management. We also address the power-efficient performance management for virtualized data centers. The coordinated management that features a hierarchical design to minimize power consumption while guaranteeing performance requirements for virtualized data centers.

The rest of this thesis is organized as follows. Chapter 2 discusses the related work. Chapter 3 and 4 present different control-based methods in adaptive performance management and adaptive power management, respectively. Chapter 5 proposes the power-efficient performance management. Chapter 6 concludes the dissertation.

Chapter 2

Related Work

In this chapter, I first investigate the existing work that has been done in the field of performance and power management for computing systems and compare them with our work. We then introduce the application of feedback control theory in computing systems.

Extensive work has been done on performance management for distributed computing systems. We now briefly summarize them by putting them into the following major categories:

1. Internet services. Performance management schemes developed for Internet services is mainly to achieve performance guarantees, capacity reservation, and prioritization on requests, and delivered bandwidth for different service classes. For example, Horvath et al. address End-to-End response time control in multi-tier web servers by using dynamic voltage scaling [37]. Wang et al. present a load balancing controller to control the relative response time among virtualized servers by manipulating the CPU resource allocation [87]. Diao et al. develop MIMO control algorithms to control the processor and memory utilization for Apache web servers [18].

2. Real-time scheduling. Performance management schemes proposed for real-time scheduling is mainly to coordinate the CPU resources allocated to different tasks that fulfill different services, so that either the CPU utilization of the targeted server can meet with the schedulability bound or the upper-bound deadline miss ratio can

be guaranteed. For example, [76] and [58] introduced a scheduling algorithm called Deferrable Server to reserve CPU resources for aperiodic and sporadic jobs so that deadlines can be guaranteed. Abdelzaher et al. [2] and Liu et al. [59] derived a schedulibility test at the admission time for any fixed-priority scheduling algorithms to guarantee deadlines. A Feedback Control real-time Scheduling (FCS) framework [60] was proposed to guarantee schedulability bound for real-time systems with unknown task execution times by dynamically adjusting the task rates.

3. Real-time databases. In real-time databases, there is usually a competition for CPU resources between data updating and the real-time user transactions. A lot of solutions have been studied to guarantee the timeliness of user transactions. For example, some on-demand updating algorithms have been developed to skip unnecessary updates and allow more CPU resources for user transactions [70] [31] [32]. Kang et al. have presented feedback controllers to manage the deadline miss ratio of user transactions and sensor data freshness [45]. Amirijoo et al. have presented feedback controllers for QoS management using imprecise computations [5]. Haritsa et al. have presented value-based scheduling algorithms in real-time databases by assigning different values for user transactions [33].

The solutions we propose is different from all the existing work. First, although many existing works also have applied feedback-control solutions in their performance management, the applications we aim for are very different from theirs. For example, the first application example we investigate is to address another important problem: real-time metadata matching, which is the bottleneck of many existing real-time information dissemination systems. Real-time metadata matching is crucial to information dissemination because information can be disseminated from publishers to matched consumers only when the matching results are generated from the metadata matching process. Our second application example is to guarantee deadlines for real-time power grid computing tasks, which is different from those real-time systems addressed by most existing work. Due to the heterogeneous properties of real-time tasks in our application, such as the execution time and period, most

existing solutions in real-time system cannot be applied in real-time power grid computing system. Second, while most existing work adopts SISO controllers, our proposed solutions feature MIMO control and hierarchical design, which can capture the coupling among different system characteristics and coordinate multiple SISO controllers to achieve improved overall system performance.

Power consumption is one of the most important design constraints for modern data centers that hold thousands of computing systems. Most existing solutions can be generalized into two classes.

1. Minimizing the power consumption within a specified performance guarantee. At the cluster level, Verma et al. propose a correlation-based scheme of consolidating servers to minimize power consumption while guaranteeing the application performance [80]. Horvath et al. minimize the power consumption while guaranteeing the end-to-end delay in multi-tier web servers by using dynamic voltage scaling (DVS) [37]. At the server level, Li et al. develop an optimal slack allocation algorithm for power adaptation between processor and memory so that energy can be minimized [53]. At the component level, energy conserving algorithms have been presented for processors [7][90], DRAM systems [93][38], disks [54][94], and cache [46]. For all the work in this category, the goal of power management is to decrease energy bills by minimizing power consumption while achieving a specified performance guarantee. Therefore, they cannot provide any explicit guarantees for the power consumption to stay below the desired power budget and avoid potential thermal emergencies, though the performance is guaranteed to meet the desired metrics.

2. Power control. Power control, or power capping, is a different but equally important problem in power management for computing servers. The goal of power control is to control the power consumption of computing systems (*i.e.*, a data center, cluster) to the power budget of the power facilities so that power overloading or thermal emergencies can be avoided. Extensive work has been done to control power for computing systems. For example, several power provisioning strategies have been proposed by [22] to host

the maximized number of servers allowed by the limited power supply in data centers. Different power capping solutions have been proposed at different levels in data centers [72] to maximize application performance. At the cluster level, Ranganathan et al. propose a negotiation-based algorithm to allocate power budget to different servers in a chassis for better performance [73]. Femal et al. present another algorithm based on linear programming [25]. At the server level, Lefurgy et al. propose a SISO feedback control solution for server power capping by assuming that CPU is the major power contributor in servers [51]. Felter et al. use open-loop control to shift power between processor and memory to maintain a server power budget [23]. Brooks et al. [10] use ad-hoc control to limit processor temperature so cheaper heat-sinks can be used. At the component level, Diniz et al. propose several policies to limit power consumption for DRAM systems [19]. Lin et al. develop an Ad-Hoc thermal management (DTM) techniques for FB-DIMM memory systems by DVFS [55]. Isci et al. present several policies that address the power capping for CMPs [43].

Our proposed power management solutions cover both of the categories above. For example, our work in Chapter 5 (*i.e.*, power-efficient performance management for virtualized data centers) falls into the first category, while our other power management solutions focus on power control at different levels (*i.e.*, cluster level, data center level, and server level) in Chapter 4 fall into the second category. However, our work is different from the existing solution in that 1) in sharp contrast to their work, which relies on heuristic-based control schemes, we adopt a rigorous design methodology that features a *control-theoretic* framework for systematically developing control strategies with analytic assurance of control accuracy and system stability [35]. 2) While most existing solution uses a single management scheme, our proposed solution features a multi-level architecture which may coordinate management schemes at different levels. For example, the architecture of our power control solution for large-scale data centers is consistent with the hierarchy of power distribution systems in data centers. At each level, we apply different power control schemes for different goals while control schemes at different level are coordinated to achieve overall system stability and improved performance.

Control theoretic approaches have been applied to a number of computing systems, for example, web services [37][18], real-time scheduling [60], network routers [36], storage management [63][62], and power and thermal management [51][73]. A survey of feedback performance control for software is presented in [3]. Our proposed work is also a part of application of control theoretic approaches in computing systems. However, our proposed solutions have significantly improvement over those existing solutions. Instead of using simple control-based solutions such as SISO controllers, our solutions feature a more complicate design, *i.e.*, MIMO optimal control and hierarchical design. Experiments on both physical testbeds and simulations demonstrate that our design can significantly improve the performance of control-based solutions in performance and power management for computing systems.

Chapter 3

Adaptive Performance Management

The need for performance guarantees in computing systems is motivated, in part, by the increasing proliferation of data services such as online banking, trading, and information dissemination. Such services may suffer from unpredictable loads, potential bottlenecks, and security breaches such as denial of service attacks. Failure to meet acceptable performance specifications may result in loss of customers, financial damage, or liability violations. Existing approaches for designing performance-guaranteed computing systems which rely on a priori workload and resource knowledge are no longer applicable.

In this chapter, we propose a novel application of feedback control-based methodology, which features MIMO control and hierarchical design, in the performance management of two important services in distributed computing systems: response time guarantees in information dissemination systems and real-time deadline guarantees for power grid computing, respectively.

3.1 Response Time Guarantees for Information Dissemination Systems

In recent years, real-time information dissemination services have become increasingly important. For example, alert messages from stock tickers should be disseminated in

a timely manner to subscribing customers. Similarly, buyers and sellers need to be matched based on their interests in e-commerce systems, and notified immediately when new business opportunities are identified. In these applications, data flows from numerous (*e.g.*, thousands of) sources to numerous sinks have to be channeled flexibly, efficiently and more importantly, in a real-time manner. This requirement has been generally described as *Valuable Information at the Right Time (VIRT)* [34], which emphasizes that consumers of information should receive the information that is of interest to them as soon as it is available or whenever it is requested.

3.1.1 Background

INFOD (INFOrmation Dissemination) [40] is an example of information dissemination system that aims to support timely delivery of valuable information for a wide range of applications. As shown in Figure 3.1, information sources and sinks are defined as *publishers* and *consumers*, respectively. *Subscriptions* are prescribed requests of information and are submitted by *subscribers* on behalf of consumers. Publishers, consumers, and subscribers advertise their attributes and constraints, which are generally referred to as *metadata*, in a database called *registry*. For example, a consumer may have its location as an attribute and have a constraint on desired publishers: they must be located within 10 miles. An example of subscription is: all sensors (publishers) send their traffic jam information to all drivers (consumers) within 10 miles no later than 5 seconds after a jam occurs. Subscriptions may have different priorities. For example, subscriptions of traffic information for police should have higher priority than those for ordinary drivers. Metadata can be updated periodically and aperiodically. INFOD needs to find new matches between the metadata of publishers and consumers based on the subscriptions within a time constraint when a metadata update arrives, which we refer to as *subscription reevaluation* or *metadata matching*. Meanwhile, the registry server should be efficiently utilized so that the maximal number of subscriptions are reevaluated to find more matched results. Based on the matched results, publishers are informed where to send filtered information. The information is then sent to the matched consumers without going through INFOD.

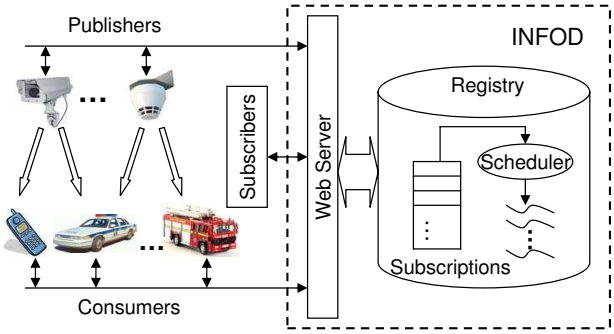


Figure 3.1: INFOD: an example information dissemination system

To guarantee both bounded matching delay and efficient system utilization in information dissemination systems faces two major challenges. *First*, when an update arrives, the system may need to reevaluate all related subscriptions to find new matching results between publishers and consumers. For example, a driver may constantly update its location attribute. As a result, all related subscriptions in the registry need to be continuously reevaluated by rerunning metadata matching to ensure that the driver receives information from the right locations. However, given the large number of publishers and consumers, reevaluating all related subscriptions may cause severe system overload and unacceptably long delays. Therefore, to guarantee bounded matching delay and avoid system overutilization, only some of subscriptions can be picked up to be reevaluated. *Second*, metadata updates may arrive at unpredictable intervals. Given a constant number of subscription reevaluations, the registry may suffer either overutilization or underutilization depending on the interval is short or long. Overutilization may lead to unbounded matching delay while underutilization may lead to unnecessarily low system throughput, which may result in incomplete matched results. Both unbounded matching delays and unnecessarily underutilized system are highly undesired. For example, a driver may fail in finding the fastest route, either due to the late notification of traffic information, or because the registry fails in finding the existing matched traffic information.

In this chapter, we firstly propose a novel Single Input Single Output (SISO) feedback controller to adaptively control the *average* response time of metadata matching by manipulating the number of low-priority subscriptions to be reevaluated. We then propose

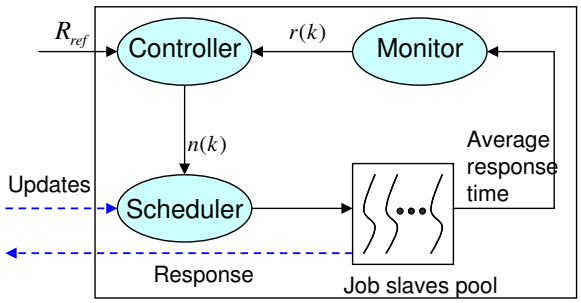


Figure 3.2: Feedback control architecture

an optimal Multiple Input Multiple Output (MIMO) controller to control both the matching delay of the high-priority subscriptions and the CPU utilization of the registry server in an integrated manner, based on a batching mechanism. Under the second control solution, the average matching delay of all the high-priority subscriptions can converge to the specified constraint, while the CPU utilization is controlled to a desired set point so that the system can reevaluate the maximal number of subscriptions.

3.1.2 Average Response Time Controller

In this subsection, we introduce the architecture, design, and analysis of a novel SISO feedback controller, to adaptively control the *average* response time of metadata matching by manipulating the number of low-priority subscriptions to be reevaluated.

Feedback Control Loop

A high-level description of the SISO control loop that controls the average response time of metadata matching by dynamically adjusting the job budget is shown in Figure 3.2. The key components in the control loop include a centralized *controller*, an average response time *monitor* and an application-level *scheduler*. The feedback control loop is invoked periodically at the end of every control period as follows: 1) The monitor measures the average response time of *all* metadata matching processes executed in the last control period and sends the value to the controller. The average response time is the *controlled variable* of the control loop. 2) The controller calculates the appropriate job budget in the next control period

based on the difference between the set point and the measured average response time. The job budget is the *manipulated variable* of the control loop. 3) Based on the calculated job budget, the scheduler schedules all the high-priority and the desired number of low-priority subscriptions to be reevaluated in the registry.

Controller Design

The core of the SISO feedback control loop is the controller. Now we introduce the problem formulation, system modeling, and the controller design.

Problem Formulation. We first introduce the following notation. T is the control period of the feedback control loop, which is selected in such a way that each control period can include multiple metadata updates. R_{ref} is the set point for average response time, which is selected to be slightly shorter than the desired time constraint to have some leeway because we are controlling the average response time. $r(k)$ is the measured average response time in the k^{th} control period. (kT sec after the system starts). $e(k)$ is the control error $e(k) = r(k) - R_{ref}$. $n(k)$ is the job budget in the k^{th} control period.

The control objective (*i.e.*, primary goal) is to guarantee that the average response time $r(k)$ converges to the set point R_{ref} within a limited *settling time*. In the meantime, we want to achieve the maximum possible job budget, $n(k)$, which is our secondary goal.

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 variable and the manipulated variable. However, a well-established physical equation is usually unavailable for computer systems. Therefore, we use a standard approach called *system identification* [26] to this problem.

Based on standard qualitative analysis [78] of the controlled system, we choose to use the following difference equation to model the controlled system:

$$r(k) = \sum_{i=1}^{n_a} a_i r(k-i) + \sum_{i=1}^{n_b} b_i n(k-i) \tag{3.1}$$

Table 3.1: Comparison of different models

Model	n_a	n_b	R^2(estimation)	R^2(validation)
1	0	1	0.8416	0.70314
2	0	2	0.85653	0.70346
3	1	1	0.8529	0.7034
4	1	2	0.85726	0.70364

where n_a and n_b are the orders of the control output and control input, respectively. a_i and 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 n_a, n_b in the difference equation (3.1). The order values are normally a compromise between model simplicity and modeling accuracy. In this work, we test different system orders as listed in the second and third columns in Table 3.1. We generate a sequence of pseudo-random digital white noise [26] as control input to stimulate the system and then measure the average response time in each control period. Based on the collected data, we use the *Least Squares Method (LSM)* to iteratively estimate the values of parameters a_i and b_i. The fourth column in Table 3.1 lists the estimated fits of the models in R^2, which is a standard metric in system identification to show how close the predicted data by the model is to the measured data from the real system. To verify the accuracy of the four models in different orders, we change the seed of the white noise to generate a different sequence of control input, and then compare the actual control output to those predicted by the estimated models in R^2. The results are shown in the last column of Table 3.1. Figure 3.3 shows the comparison results between the actual system output and the predicted outputs of two models. Model 1 has the lowest orders with $n_a = 0$ and $n_b = 1$ while Model 4 has the highest orders with $n_a = 1$ and $n_b = 2$. Figure 3.3 demonstrates that Models 1 and 4 both are sufficiently close to the actual system. We choose to use Model 1 in this work to simplify the controller design. Therefore, our system model is:

$$r(k) = b_1 n(k-1). \tag{3.2}$$

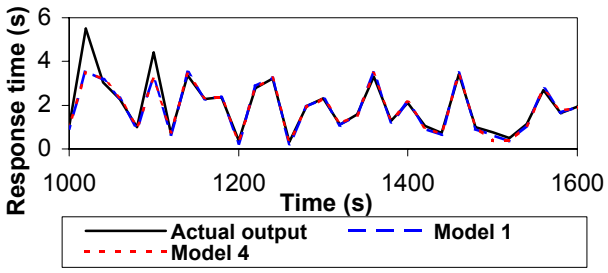


Figure 3.3: Comparison between predicted output and actual output

Root-Locus Design. The goal of the controller design is to meet the following requirements:

- Stability: The average response time should settle into a bounded range around the set point, R_{ref}, in response to a bounded reference input.
- Zero steady state error: The average response time should precisely settle to the set point.
- Short settling time: The system should settle to the set point within a limited time period.

Proportional-Integral (PI) control [26] has been widely adopted in industry control systems. We choose to use a PI controller because the integral part can eliminate the steady-state error. A more sophisticated PID (Proportional-Integral-Derivative) controller is not used because the derivative term may amplify the noise in response time. Following standard control theory, we design the PI controller in the Z-domain as:

$$F(z) = \frac{K_1(z - K_2)}{z - 1} \tag{3.3}$$

where K_1 and K_2 are control parameters that can be analytically chosen to guarantee control performance, using standard control design methods [26]. The time-domain form of the controller (3.3) is:

$$n(k) = n(k-1) + K_1 e(k) - K_1 K_2 e(k-1) \tag{3.4}$$

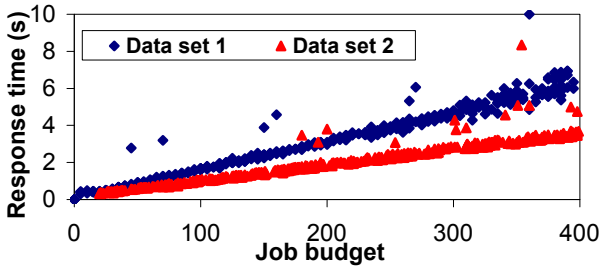


Figure 3.4: Linear relationship between average response time and job budgets

Using the Root-Locus method [26], we can choose our control parameters as $K_1 = 1/b_1$ and $K_2 = 0$ such that our closed-loop transfer function is:

$$G(z) = z^{-1} \tag{3.5}$$

It is easy to prove that our controller is stable and can achieve the set point in one control period. The detailed proofs can be found in a standard control textbook [26] and are skipped here.

Performance Analysis

The SISO controller is designed to achieve the control performance when the system model is accurate. However, in a real system, the actual system model could be different from the nominal model (3.2) we used to design the controller for several reasons. First, the execution time of reevaluating each subscription may be different due to different numbers of involved attributes and constraints. Second, the execution time of reevaluating the same subscription may vary at runtime as metadata updates could constantly change the attributes and constraints of publishers and consumers. Finally, the execution time may also be influenced by the hardware and software configurations of a particular system. Since developing a different controller for every different system with different sets of subscriptions is infeasible, it is important to analyze the impact of model variations on control performance.

As shown in Figure 3.4, an important observation from our experiments is that there exhibits an approximately linear relationship between the average response time of subscription reevaluation and the job budget despite different sets of subscriptions or systems.

Based on this observation, it is valid to assume that a real system model is similar to the nominal model but with a different parameter b_1'. Without loss of generality, we model the real system as:

$$r(k) = gb_1 n(k-1) \tag{3.6}$$

where $g = b_1'/b_1$ is the *execution time factor* and is used to model the variation between the real system model (3.6) and the nominal model (3.2). Therefore, the closed-loop transfer function for a real system (3.5) is:

$$G(z) = \frac{g}{z-(1-g)}. \tag{3.7}$$

Now we analyze each control performance metric with the real system model.

Stability. Based on the control theory, if the system is stable, all the poles should be within the unit circle on the z-plane [26]. Based on (3.7), we get $|1-g| < 1$. Hence the system remains stable as long as $0 < g < 2$. This stability range serves as an important reference when we apply our controller to different systems and subscriptions, *i.e.*, the average execution time of reevaluating a subscription in the real system should *not* be more than twice greater than the nominal value used in the controller design.

Steady State Error. The steady-state error of the real system can be derived as:

$$\lim_{z\to 1}(z-1)G(z)R_{ref}\frac{z}{z-1} = \lim_{z\to 1}(\frac{gz}{z-(1-g)}R_{ref}) = R_{ref}. \tag{3.8}$$

Equation (3.8) means we are guaranteed to achieve the desired average response time as long as the system is stable.

Settling Time. We transform (3.7) to the time-domain:

$$r(k) = (1-g)r(k-1) + gR_{ref} \tag{3.9}$$

Using the common definition of the settling time (the system settles when the output converges into the ± 0.05 range of the reference), we derive the required number of control

periods, k, for the system to settle as:

$$k \geq \frac{\ln 0.05}{\ln |1 - g|} \tag{3.10}$$

Our above analysis gives us theoretical confidence in the performance of our controller and provides a guideline to choose the parameters in the nominal system model (3.2). For example, given the possible minimum and maximum values of b_1' for typical sets of subscriptions, we can choose the nominal b_1 to be a value slightly larger than $\frac{b'_{1,min}+b'_{1,max}}{2}$ such that the system is guaranteed to be stable even when the real model is unknown at design time. Similarly, given our analysis regarding settling time, we can use a smaller b_1 for faster reaction to variations or a greater b_1 to reduce the system sensitivity. This kind of theoretical guidance is in sharp contrast to commonly used heuristic solutions that heavily rely on extensive empirical evaluation and manual tuning for desired system response.

3.1.3 Integrated Control of Matching Delay and CPU Utilizaiton

The SISO feedback controller successfully address the first challenge mentioned in Section 3.1.1 by manipulating the number of low-priority subscriptions to be reevaluated to control the *average* response time of metadata matching. It assumes that the interarrival intervals of incoming metadata updates are within a reasonable range of $(2s, 5s)$. However, metadata updates may arrive at unpredictable intervals in some extreme cases in real systems. To address all the challenges, we propose a mechanism by applying a batching window to group those updates at small interarrival intervals and release them all together. After each release, all high-priority subscriptions are reevaluated first, and then a certain number of low-priority subscriptions are selected for reevaluation. The low-priority subscriptions are reevaluated in a round-robin way. Clearly, both the batching window size and number of subscriptions chosen to be reevaluated affect the matching delay and utilization of the registry server. Therefore, it is important to determine the batching window size and number of subscriptions in an integrated way such that the average matching delay of all high-priority subscriptions are

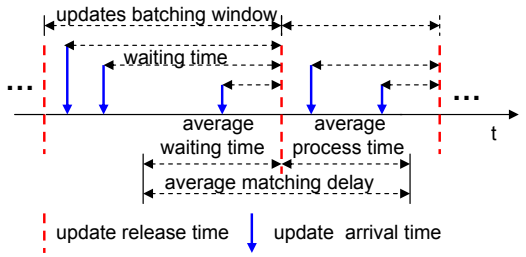


Figure 3.5: Illustration of batching window

guaranteed to meet the specified constraint, and the registry server can be efficiently utilized to achieve maximal system throughput.

In this subsection, we present an integrated control solution that controls both the matching delay of the high-priority subscriptions and CPU utilization of the registry server in an integrated manner. Under the new control solution, the average matching delay of all the high-priority subscriptions can converge to the specified constraint, while the CPU utilization is controlled to a desired set point so that the system can reevaluate the maximized number of subscriptions.

System Overview

Since the integrated control solution is based on the batching mechanism, we first introduce the batching mechanism, and then give a high-level description of the control architecture.

Batching Mechanism. As shown in Figure 3.5, we employ a *batching window* to accumulate all updates that come within the window and release them in a batch. At the end of each batching window, defined as the *releasing time*, all high-priority subscriptions are chosen for reevaluation first, and then some of low-priority subscriptions are reevaluated. All the low-priority subscriptions are picked up in a round-robin way. Specifically, we refer to the length of time between two adjacent releasing times as the *batching window size*. We define the difference between the arrival time and the releasing time of an updates as the *waiting time* of that update. The difference between the releasing time and the time when a subscription is completed is defined as the *processing time* of that subscription. The *average matching delay* of all high-priority subscriptions is calculated as the sum of the average

waiting time of all updates in a batching window and the following average processing time of all reevaluated high-priority subscriptions after the batching window.

The number of all reevaluated subscriptions each second is defined as the ***overall throughput*** of the system. The number of all subscriptions that are reevaluated after each batching window is defined as the ***job budget***, which should be maximized so that more low-priority subscriptions can be reevaluated after each batching window. The number of reevaluated low-priority subscriptions each second is defined as the ***throughput of low-priority subscriptions***. Hereinafter, we refer to both the overall throughput and the throughput of low-priority subscriptions as ***system throughputs***. In addition, the average time that all low-priority subscriptions take to be reevaluated at least once is defined as the ***average matching interval***.

Control Architecture. As shown in Figure 3.6, the key components in the integrated control loop include the integrated controller, delay and CPU utilization monitor, updates batcher, and scheduler. The control loop is invoked periodically at the end of every control period as follows: 1) The monitor measures the average matching delay and CPU utilization in the last control period, and sends the values to the controller. The average matching delay and CPU utilization are the *controlled variables* in the control loop. 2) The controller calculates the appropriate job budget and batching window size for the next control period based on the differences between the set points and measured controlled variables. The job budget and batching window size are the *manipulated variables* in the control loop. 3) Based on the calculated batching window size, all updates that arrive within the window are released in a batch. At the releasing time, the batcher calculates the average waiting time of all updates in the window and notifies the scheduler in the database. 4) After receiving the notification from the batcher, the scheduler, based on the calculated job budget, schedules all of the related high-priority subscriptions and the desired number of low-priority subscriptions to be reevaluated in the system.

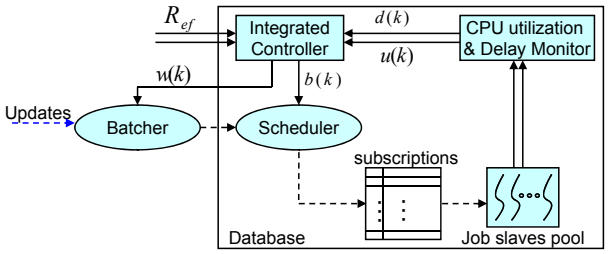


Figure 3.6: Integrated control architecture

Integrated Controller

Now we present the problem formulation, modeling, design, and analysis of the integrated controller.

Problem Formulation. We first introduce the following notation.

- $\mathbf{R_{ef}}$: The reference vector, $\mathbf{R_{ef}} = \begin{bmatrix} R_d & R_u \end{bmatrix}^T$, where R_d and R_u are the reference values of the average matching delay and CPU utilization, respectively.
- T: The control period of the integrated controller.
- $w(k)$, $p(k)$, and $d(k)$: The average waiting time of each update, the average processing time of the high-priority subscriptions, and the average matching delay in the k^{th} control period, respectively. Specifically, $d(k) = p(k) + w(k)$.
- $u(k)$, $s(k)$, and $b(k)$: The average CPU utilization, the batching window size, and job budget, respectively, in the k^{th} control period.
- $\overline{d}$, $\overline{u}$, $\overline{s}$ and $\overline{b}$: The operating points of $d(k)$, $u(k)$, $s(k)$, and $b(k)$.

The goal of the integrated control is to simultaneously control both the average matching delay of the high-priority subscriptions and CPU utilization to their respective set points, R_d and R_u. The purpose of controlling the average matching delay is to guarantee that all matched results of high-priority subscriptions can be disseminated within a time constraint after a metadata update arrives. The purpose of controlling the CPU utilization is to achieve maximized system throughputs (*i.e.*, the overall throughput and throughput for low-priority subscriptions) so that more subscriptions can be reevaluated to find more matched results between data sources and sinks. The selection of the set point of the CPU utilization

is a compromise between the control accuracy of the average matching delay and system throughputs. The higher the CPU utilization, the larger the system throughputs, but the more difficult it is to control the average matching delay accurately, which may result in undesired large oscillations.

The integrated controller is formulated as an optimal control problem to find the optimal batching window size and maximize the job budget while controlling matching delay and CPU utilization to their set points.

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.*, $d(k)$ and $u(k)$) and the manipulated variables (*i.e.*, $b(k)$ and $s(k)$). However, a well-established physical equation is usually unavailable for computing systems. Therefore, we apply a standard approach called system identification [26] to this problem.

We use the following difference equation to model the controlled system [78]:

$$\mathbf{y(k)} = \sum_{i=1}^{n_a} \mathbf{A_i y(k-i)} + \sum_{i=1}^{n_b} \mathbf{B_i v(k-i)}, \tag{3.11}$$

where $\mathbf{y(k)} = \begin{bmatrix} d(k) - \overline{d} & u(k) - \overline{u} \end{bmatrix}^T$ and $\mathbf{v(k)} = \begin{bmatrix} b(k) - \overline{b} & s(k) - \overline{s} \end{bmatrix}^\mathbf{T}$, the input and output vector, respectively. n_a and n_b are the orders of the control outputs and control inputs. $\mathbf{A_i}$ and $\mathbf{B_i}$ are model parameters whose values need to be determined by system identification.

To have an accurate model, we need to identify the operating points of the controlled system. For regulatory control, the operating points of the outputs (*i.e.*, $\overline{d}$ and $\overline{u}$) are typically chosen to lie close to the reference values (*i.e.*, R_d and R_u) [35]. Meanwhile, the operating points of the inputs are identified (*i.e.*, $\overline{b}$ and $\overline{s}$) by preliminary experiments such that the operating points of the outputs can be reached [78].

For system identification, we need to first determine the right orders for the system, *i.e.*, n_a and n_b, in the difference equation (3.11). The order values are normally a compromise between model simplicity and model accuracy. In this work, we test different system orders by using pseudo-random digital white noise [78] to stimulate the system and then measure the

control outputs (*i.e.*, $d(k)$ and $u(k)$) in each control period. Our experiments are conducted on the testbed introduced in detail in Section 3.1.4. Based on the collected data, we use the *Least Squares Method (LSM)* to iteratively estimate the values of parameters $\mathbf{A_i}$ and $\mathbf{B_i}$. In this work, we choose $n_a = 1$ and $n_b = 1$ for a trade-off between model complexity and model accuracy.

We then generate another sequence of white noise to validate the results of system identification. Figure 3.7 shows the measured outputs from the open-loop system and the predicted outputs from the model. The errors measured in *Root Mean Squared Error (RMSE)* are sufficiently small (*i.e.*, 0.113 and 0.031 for the average matching delay and the CPU utilization, respectively). We can see that the predicted outputs of the selected model are sufficiently close to the actual system outputs. Therefore, the resultant system model from our system identification is:

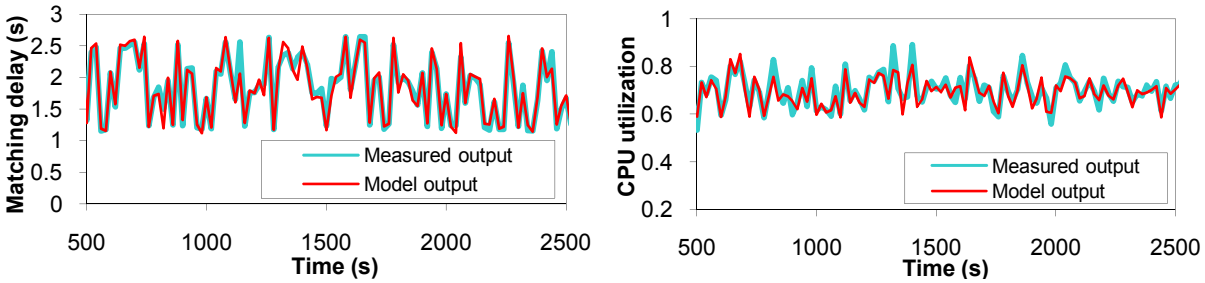
$$\mathbf{y(k)} = \mathbf{A_1 y(k-1)} + \mathbf{B_1 v(k-1)}, \tag{3.12}$$

where A_1 and B_1 are 2×2 constant matrices whose values are determined by system identification.

Controller Design. We apply the *Linear Quadratic Regulator (LQR)* control theory to design the controller based on the system model (3.12). LQR is an optimal control technique that can deal with coupled MIMO control problems and has small computational overhead at runtime. To design the controller, we first convert our system model to a state space model. The state variables are defined as follows:

$$\mathbf{x(k)} = \begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{bmatrix}, \tag{3.13}$$

where $\mathbf{e}(k) = \mathbf{R_{ef}} - \begin{bmatrix} d(k) & u(k) \end{bmatrix}^T$ and $\mathbf{e_I}(k) = \mathbf{e_I}(k-1) + \mathbf{e}(k)$ are the control error vector and the accumulated control error vector, respectively. We then design the LQR controller



(a) Matching delay **(b)** CPU utilization

Figure 3.7: System model validation

by choosing gains to minimize the following quadratic cost function:

$$\mathbf{J} = \sum_{k=1}^{\infty} \begin{bmatrix} \mathbf{e(k)^T} & \mathbf{e_I(k)^T} \end{bmatrix} \mathbf{Q} \begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{bmatrix} + \sum_{k=1}^{\infty} \mathbf{v^T(k) R v(k)}, \tag{3.14}$$

where $\mathbf{Q}$ and $\mathbf{R}$ are weighting matrices that determine the trade-off between the control errors and control efforts. The first item in (3.14) represents the control errors and accumulated control errors. By minimizing the first item, the closed-loop system can converge to the desired set points. The second item in (3.14) represents the control efforts. Minimizing the second item ensures that the controller will minimize the changes in the control inputs, *i.e.*, the changes of the job budget and batching window size. The LQR controller is designed by using the Matlab command *dlqry* to solve the optimization problem (3.14). The controller gain matrix $\mathbf{K}$ is as follows:

$$\mathbf{K} = \begin{bmatrix} \mathbf{K_P} & \mathbf{K_I} \end{bmatrix}, \tag{3.15}$$

where $\mathbf{K_P}$, $\mathbf{K_I}$ are constant controller parameters for the error vector $\mathbf{e}(k)$ and the accumulating error vector $\mathbf{e_I}(k)$, respectively, so that the cost function (3.14) is minimized. Consequently, the control inputs in the k^{th} control period, *i.e.*, the job budget and the batching window size, are computed as:

$$\begin{bmatrix} b(k) - \overline{b} \\ s(k) - \overline{s} \end{bmatrix} = -\mathbf{K} \begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{bmatrix} \tag{3.16}$$

Control Analysis for Model Variations. A fundamental benefit of the control-theoretic approach is that it gives us theoretical confidence in system stability, even when the controller is used under different working conditions. Now we analyze the system stability when the integrated controller is applied to a system whose model is different from the nominal model described by (3.12). One of the major model variations is due to the varying arrival time of updates within the batching window. As an example, we give the stability analysis for varying arrival time of updates.

Since most updates arrive within a certain batching window, we have $w(k) \leq s(k)$. Therefore, we assume $w(k) = h \cdot s(k)$, where $0 \leq h \leq 1$. We define h as the *waiting-time factor*. In system identification, we assume that the average waiting time of updates is half of the batching window size, *i.e.*, $h = 0.5$. However, in real systems, the average waiting time of updates is unknown a priori. To verify that the closed-loop system is stable when the average waiting time is any proportion of the batching window size, here we outline the general steps to analyze the stability:

1. We model the variation of the average arrival time as h in the matrix B_1 in the system model (3.12) as follows:

$$\mathbf{B_1'} = \begin{bmatrix} b_{11} & b_{12} - 0.5 + h \\ b_{21} & b_{22} \end{bmatrix}, \tag{3.17}$$

   where $\mathbf{B_1'}$ is the varied matrix of $\mathbf{B_1}$ at runtime. b_{12} models the relationship between the window size and matching delay;

2. Derive the closed-loop system model by substituting the derived control inputs $\mathbf{v}(k)$ into the system model (3.12) by replacing $\mathbf{B_1}$ with $\mathbf{B_1'}$. The closed-loop system model is in the following form:

$$\begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{bmatrix} = (\mathbf{A} - \mathbf{B'K}) \begin{bmatrix} \mathbf{e}(k-1) \\ \mathbf{e_I}(k-1) \end{bmatrix} + \begin{bmatrix} \mathbf{I} - \mathbf{A_1} \\ \mathbf{0} \end{bmatrix} \mathbf{r}, \tag{3.18}$$

   where $\mathbf{B'} = \begin{bmatrix} -\mathbf{B_1'} & \mathbf{0} \end{bmatrix}$, and $\mathbf{r} = \mathbf{R_{ef}} - \begin{bmatrix} \overline{d} & \overline{u} \end{bmatrix}^T$.

3. Derive the stability condition of the closed-loop system described by (3.18). According to control theory, the closed-loop system is stable if all eigenvalues of the matrix $(\mathbf{A} - \mathbf{B}'\mathbf{K})$ are located inside the unit circle.

Following the steps above, we have proven that the closed-loop system is stable when g varies from 0 to 1 at runtime, which means that the system can be guaranteed to be stable no matter when the updates arrive. Similarly, we can also analyze the stability with other model variations by following the steps above.

3.1.4 Experimental Results

In this subsection, we first introduce the testbed. We then present the experimental results for the SISO controller and integrated controller, respectively.

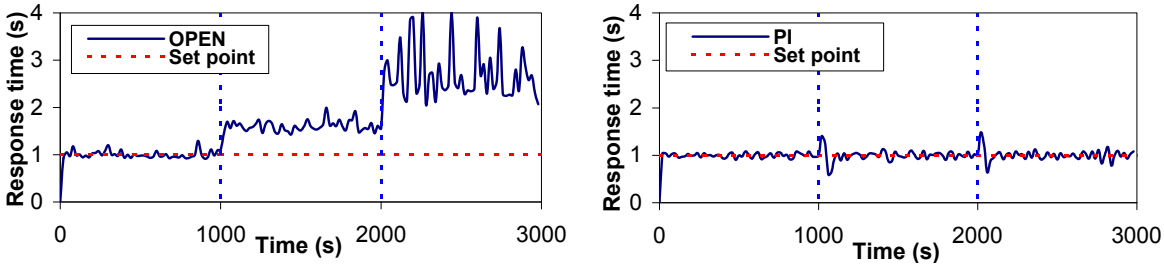
Testbed

Our testbed includes two servers connected via a network switch. Server 1 is equipped with Intel Core 2 Duo Xeon 5160 3.0GHz and Server 2 is equipped with AMD Athlon 64x2 4200+ 1.0GHz. Both of them run openSUSE 10.3 (2.6.22) and JDK 1.6.0. We use the INFOD system introduced in Section 3.1.1 as a representative information dissemination system to test our control loops. The INFOD registry is implemented in Oracle Database 11.1g on Server 1 to run the metadata matching processes. Since our objective is to evaluate the performance of the control loops running on Server 1, all INFOD publishers, consumers and subscribers are implemented on Server 2 to simplify the experimental setup.

The process of metadata matching is implemented as an Oracle PL/SQL procedure and configured as an event-based task with a priority of 3 out of the five priorities (1 to 5, with 1 as the highest) provided by the Oracle database scheduler to let the controllers have the highest priority to run.

SISO Response Time Controller

In all the experiments for the SISO response time controller, we use $1s$ as the set point for the response time of metadata matching (*i.e.*, subscription reevaluation). The same set of



(a) OPEN (b) PI

Figure 3.8: Control accuracy comparison between open-loop system and PI controller

metadata updates are used with interarrival intervals randomly distributed within $[2s, 5s]$ if not otherwise indicated.

Baselines. We use two algorithms: OPEN and Ad Hoc, as our baselines. OPEN is an open-loop algorithm that uses fixed job budget based on the estimated execution time of metadata matching. Although OPEN can result in desired average response time when estimated execution time is accurate, it may violate the timing constraint of metadata matching when execution time is underestimated. The resultant violation of time constraint may cause highly undesired accidents in real systems. At the same time, it may also cause an average response time unnecessarily shorter than the desired value when execution time is overestimated. Consequently, the job budget includes fewer low-priority subscriptions, which may lead to poor system quality of service.

Ad Hoc is a heuristic-based adaptive controller that represents a typical industry solution to response time control. At each control invocation, it simply raises or lowers the job budget by a certain step, depending on whether the measured average response time is lower or higher than the set point. However, it is commonly difficult for Ad Hoc to decide the best step size in a real system. Small steps may cause undesired slow response while large steps may cause the system to oscillate dramatically. In our experiments, Ad Hoc has a control period of 20s just as the PI controller.

Comparison with OPEN. In this experiment, we test both OPEN and PI in a scenario common to many real-time information dissemination systems. In this scenario, the average reevaluation of subscriptions has sharp increase at run time. This is common to real-time

information dissemination systems and could be caused by several reasons such as metadata updates adding new publishers or consumers to the system. We first adopt the same set of subscriptions that are used to design OPEN and PI. As a result, the estimated reevaluation time of a subscription is accurate at the beginning for both of the two controllers, i.e., g = 1. At time 1000s, the reevaluation time has a sharp increase and is larger than the estimation, i.e., g = 1.4. At time 2000s, more updates further increase the real execution time, i.e., g = 1.8.

As discussed before, OPEN is designed to have a fixed job budget to achieve the desired response time based on accurate knowledge of the subscription reevaluation time. Figures 3.8a shows that OPEN indeed achieves the desired response time at the beginning. However, OPEN violates the response time constraint after the execution time increase because OPEN cannot adapt the system for desired response time by reducing the job budget. Consequently, the average response time of metadata matching increases to $1.6s$. This long delay is highly undesired in real-time systems because the consumers of high-priority subscriptions (*e.g.*, firefighters) may therefore get into dangerous situations when they receive important information (*e.g.*, fire condition reports) with constant delays. With a further increase of execution time at time $2000s$, OPEN has even longer delay and also causes the system to have large oscillations. This is because the execution time of reevaluating those subscriptions becomes longer than the interarrival interval of metadata updates. As a result, the system cannot complete the subscription reevaluation within an interval, which leads to jitters in different control periods.

In contrast to OPEN, PI achieves the desired response time despite the variations in the subscription reevaluation time, as shown in Figure 3.8b. This is because PI monitors the real response time and dynamically adapts the job budget based on control theory. This experiment demonstrates that OPEN only works well when we have accurate knowledge of the average execution time of subscription reevaluation, and the average execution time never changes. However, our PI controller can guarantee the average response time converges to the set point in spite of significant average execution time variations.

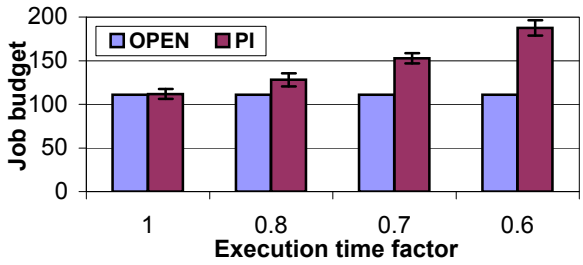
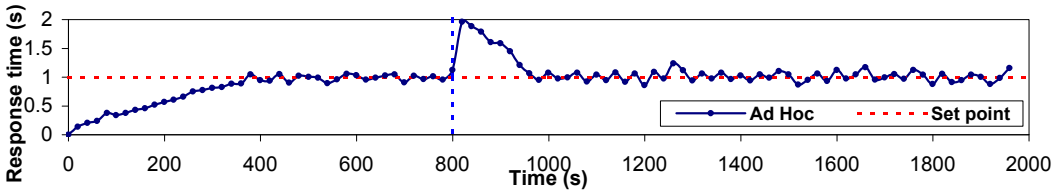


Figure 3.9: System quality of service comparison between OPEN and PI under different execution time factors

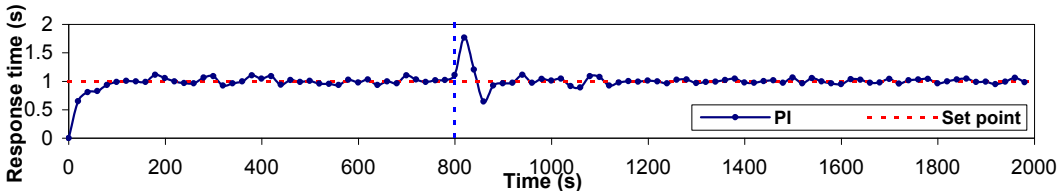
The primary goal of our feedback control architecture is to guarantee that the average response time of metadata matching (*i.e.*, subscription reevaluation) is shorter than a real time constraint. The secondary goal is to have a job budget (*i.e.*, total number of reevaluated subscriptions) as large as possible in each control period to achieve the best system quality of service. Because OPEN violates the response time constraint when the execution time factor, g, is larger than 1, as shown in Figure 3.8a, we only consider the situations when $g \leq 1$ in the comparison between PI and OPEN. Specifically, we use four sets of subscriptions with the execution time factor as 1, 0.8, 0.7 and 0.6, respectively. The resultant job budget for each subscription set is shown in Figure 3.9. Each data point is the average of 50 job budgets in the steady state for each execution time factor. Since OPEN uses a fixed job budget of 111, it has an unnecessarily short average response time when $g < 1$. In contrast, PI can control the average response time to converge to the desired set point by dynamically increasing the job budget from 111.9 to 128.1, 152.7 and 187.6. Although OPEN and PI both meet the constraint, PI can reevaluate more low-priority subscriptions thus provides better system quality of service.

Comparison with Ad Hoc. In this experiment, Ad Hoc is designed to achieve the best performance with a certain set of subscriptions whose real reevaluation time is smaller than the estimation ($g = 0.6$). Based on extensive tuning and testing, we find that a step size of 10 gives Ad Hoc performance as good as PI in the steady state.

Figures 3.10a and 3.10b show the control results of Ad Hoc and PI, respectively. At the beginning, Ad Hoc increases the job budget by 10 in each control period until the average response time reaches the set point. Since Ad Hoc relies on a fixed step to reach its steady



(a) Ad Hoc



(b) PI

Figure 3.10: Control accuracy comparison between Ad Hoc and PI

state, it takes Ad Hoc 19 control periods ($380s$) to settle down to the set point. In contrast, PI can effectively adapt its step size based on control theory. As a result, PI only takes 5 control periods ($100s$) to enter its steady state. Although it is possible to configure Ad Hoc with a greater step, doing so may cause Ad Hoc to have large oscillation in steady state, which is undesired for system performance. In the steady state until $800s$, the two controllers work similarly with almost the same deviations (0.055 for Ad Hoc and 0.051 for PI), as Ad Hoc is carefully tuned to do so.

At time $800s$, we assume the execution time of subscription reevaluation increases (with $g = 1$) due to the reasons given before. Similarly, Ad Hoc takes 10 control periods ($200s$) to return to the steady state. However, PI only takes 5 control periods. Since the step size of Ad Hoc is designed to achieve the best performance when the execution time factor is 0.6, the response time under Ad Hoc oscillates around the set point and thus frequently violates the time constraint. In contrast to Ad Hoc, PI converges to the set point smoothly with a small deviation of 0.044. This experiment demonstrates that Ad Hoc cannot effectively adapt to varying average execution time. It is commonly difficult for Ad Hoc to tune its step size at runtime, due to the lack of established adaptation methods. In contrast, PI relies on control theory for adaptation and thus can provide response time guarantees in spite of execution time variations.

In this experiment, we show PI also has better system quality of service than Ad Hoc. We use four sets of subscriptions with different execution time factors (0.6, 1.0, 1.4 and 1.8).

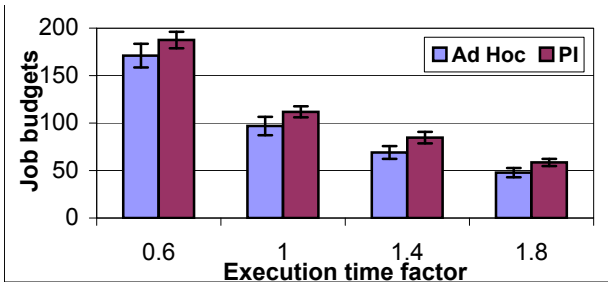


Figure 3.11: System quality of service comparison between Ad Hoc and PI under different execution time factors

As discussed before, it is commonly difficult for Ad Hoc to have a step size that is good for all workloads, because there exists a trade-off between system oscillation and settling time. Finer step size usually leads to smaller oscillation in steady state, but may cause longer settling time and so slower response to workload variations. Long settling time could be dangerous to the consumers of high-priority subscriptions (*e.g.*, firefighters), when the system needs to quickly converge back to the desired response time from an unexpected workload increase. Therefore, to have an acceptable settling time, we tune the step size of Ad Hoc such that Ad Hoc has a settling time within 10 control periods ($200s$). We do this for every subscription set to have a fair comparison with PI. However, this shorter settling time causes Ad Hoc to have larger system oscillation and thus frequent violations of time constraint. Hence, to achieve similar performance with PI for a fair comparison, a safe margin needs to be used for Ad Hoc to lower its set point a little for fewer violations of constraint. We calculate the safe margin as follows. First, we tune a step in the way we describe above for each set of subscriptions. We then calculate the standard deviation of a typical run for each set of subscriptions with its individual step. Finally, we use the maximum deviation of the four sets of subscriptions as the safe margin for Ad Hoc. The safe margin is calculated as $0.08s$ in our experiments. We then compare Ad Hoc and PI for the average job budget of 50 control periods in steady state. The results shown in Figure 3.11 demonstrate that PI can reevaluate more low-priority subscriptions, thus outperforms Ad Hoc in term of system quality of service.

Control Performance under Different Execution Time Factors. As discussed in Section 3.1.2, the stability of our controller is related to the execution time factor g. In this

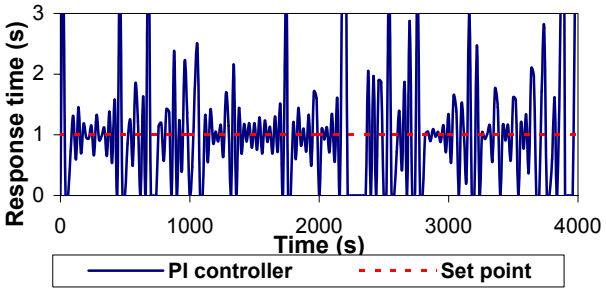


Figure 3.12: Controller outputs when g=2.6

experiment, we verify that theoretical result by using experiments to test the system stability with different g. In Figure 3.8b, the deviation of response time after the controller settles down to the steady state slightly increases from 0.047 to 0.064 and 0.078 as the execution time factor increases from 1 to 1.4 and 1.8 at time $1000s$ and $2000s$, respectively. However, when the execution time factor changes to 2.6, as shown in Figure 3.12, the controller becomes unstable as the response time oscillates significantly between 0 and a large value. The result is consistent with the stability range ($0 < g < 2$) derived in our stability analysis in Section 3.1.2.

To further investigate the relationship between system stability and g, we plot the mean and deviation of the average response time with different execution factors in Figure 3.13. In the figure, each data point is the mean of 50 outputs when the controller is in its steady state. The standard deviation of the average response time indicates the intensity of oscillation. As the execution time factor increases from 0.6 to 1.8, the standard deviation remains below 0.1. The small deviation is caused by uncertainties in computer systems (*e.g.*, cache, pipelining). When the execution time factor increases to 1.95 that is close to the theoretical stability bound, the standard deviation starts to increase noticeably. When the execution time factor locates outside the theoretical stability range, the system becomes unstable as it has significant oscillation and the average response time deviates from the set point. Figure 3.13 shows that the empirical results on a physical test-bed confirm our theoretical analysis. This experiment demonstrates that, in contrast to OPEN and Ad Hoc which rely on exhaustive iterations of tuning and testing, PI can provide verified theoretical guarantee that system is stable within a certain range of g.

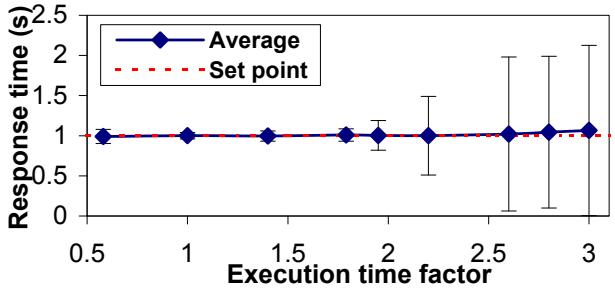


Figure 3.13: Average response time and deviation under different execution time factors

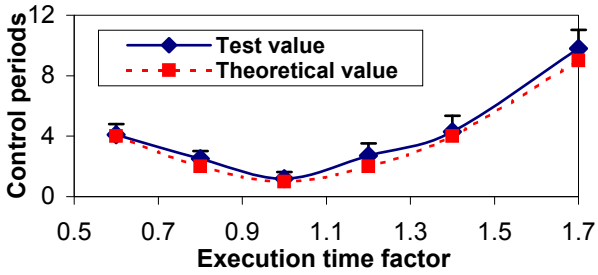


Figure 3.14: Settling time under different execution time factors

Our theoretical analysis (Equation (3.9)) reveals that the settling time of the PI controller is also related to the execution time factor g. In this experiment, we verify this result with different sets of subscriptions with the execution time factor as 0.6, 0.8, 1.0, 1.2, 1.4 and 1.7, respectively. For each set of subscriptions, we measure the average settling time as the number of needed control periods. Figure 3.14 shows the average and deviation of settling time in 10 repeated runs for different execution time factors. The theoretical values are rounded integers from the calculated results using Equation (3.9) because the number of control periods cannot be fractional. As shown in the figure, the closer the execution time factor approaches to 1, the shorter the settling time is. The largest deviation is 1.23 when $g = 1.7$. We can see that the experimental results are very close to the theoretical results. This experiment confirms our theoretical analysis.

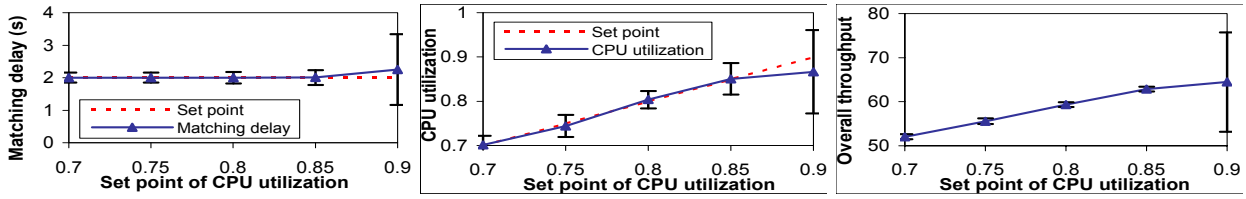
Integrated Controller

Now we present the experimental results for the integrated controller. We first test different set points of CPU utilization to justify the selection of the set point for CPU utilization employed in our experiments. We then evaluate the performance of the integrated controller. Finally, we compare it with two baselines.

According to queuing theory, the arrival of requests at a server can often be realistically characterized as a Poisson process [6]. Without loss of generality, we use updates whose arrival pattern follows the Poisson process with specific values of λ (*i.e.*, the average number of arriving updates every second) in this work. We refer to λ in Poisson process as the *update arrival rate*. The updates in our testbed have some typical arrival rates, ranging from 2 to 10. Note that our control algorithm is not limited to Poisson process. In all of the following experiments, we use $2s$ as the set point of the average matching delay and 0.8 as the set point of the CPU utilization. The steady states of both the average matching delay and CPU utilization are defined as $\pm 10\%$ of the set point.

Set Point Selection for CPU Utilization Control. As explained in Section 3.1.3, the selection of the set point of CPU utilization is a trade-off between the control accuracy of the matching delay and overall throughput. In this subsection, we test the effect of set points of CPU utilization on the control accuracy and overall throughput (*i.e.*, the number of reevaluated subscriptions per second) with experiments.

We first run the integrated controller with different set points of CPU utilization, from 0.70, 0.75, 0.80, 0.85 to 0.90, while keeping the set point of the average matching delay constantly at $2s$. The reason why we do not vary the set point of the matching delay is that the constraint of matching delay is usually specified by customers. We then plot the mean and standard deviation of 50 outputs in the steady state for the average matching delay, CPU utilization, and the overall throughput in Figure 3.15. As shown in Figures 3.15(a) and (b), both the average matching delay and the CPU utilization have mean values approximately equal to their set points with only small deviations when the set point of the CPU utilization is smaller than 0.9. However, as the set point increases to 0.9, both the average matching delay and CPU utilization have mean values deviating from the set point with significant oscillations. This is because the system becomes overloaded and the system model is nonlinear when the CPU utilization approaches 0.9. As shown in Figure 3.15(c), the overall throughput steadily improves as the set point increases from 0.7 to 0.85 and reaches the maximal value at the CPU utilization of 0.85 with small oscillations. However, as the



(a) Matching delay (b) CPU utilization (c) Overall throughput

Figure 3.15: Control accuracy and overall throughput with different set points of CPU utilization

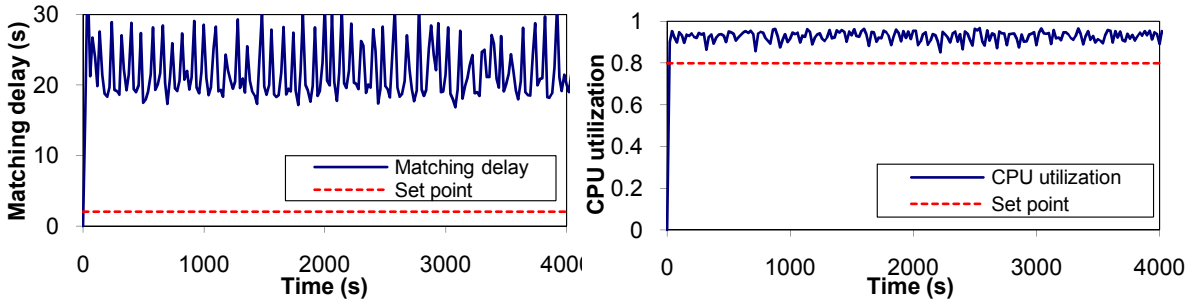
set point increases to 0.90, the overall throughput begins to oscillate significantly due to the large oscillation of the CPU utilization.

To ensure that the controller works safely in the operating region, we choose 0.80 as the set point to allow some leeway for the nonlinear region. This experiment gives us a good reference to choose the set point for CPU utilization.

Control Performance of the Integrated Controller. We first demonstrate the performance of the open-loop system to show the importance of controlling the matching delay and CPU utilization. We then evaluate the performance of the integrated controller with different update arrival rates.

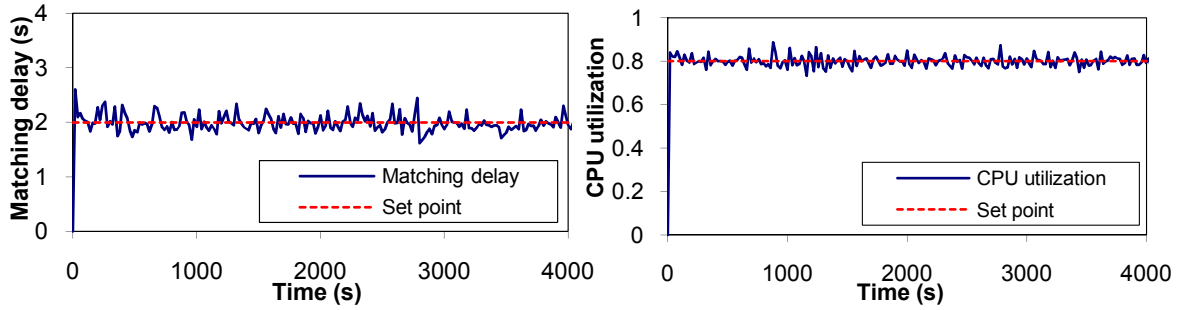
The open-loop system is the system without any matching delay or CPU utilization management, in which all the subscriptions are reevaluated upon the arrival of each update. In this experiment, we use an update arrival rate of 4 (*i.e.*, $\lambda = 4$), which is a typical arrival rate in real systems. Figure 3.16 shows the average matching delay and CPU utilization of the open-loop system with 400 subscriptions. We can see that reevaluating all subscriptions upon each update causes severe system overload (*i.e.*, the CPU utilization is almost 0.93 (there is some CPU time for I/O waiting.)) and unacceptably long delays (*i.e.*, the delay is greater than $30s$ at times). This experiment shows that unbounded matching delay and system overutilization will occur without matching delay or CPU utilization management in information dissemination systems. Figure 3.17 shows that the integrated controller successfully achieves the desired matching delay and CPU utilization with only small oscillation.

In real information dissemination systems, the update arrival rate may vary due to different factors, such as working hours. To evaluate the integrated controller under varying



(a) Matching delay (b) CPU utilization

Figure 3.16: A typical run of the open-loop system



(a) Matching delay (b) CPU utilization

Figure 3.17: A typical run of the integrated controller

arrival rates, we run experiments and plot the mean and standard deviation of 50 outputs in the steady state when the arrival rates vary from 2, 4, 6, 8 to 10. The standard deviation indicates the intensity of oscillation of the outputs in the steady state. From Figure 3.18, we can see that the mean of the average matching delay and CPU utilization are all around the set points with only small standard deviations (the largest is $0.18s$ and 0.03, respectively). This experiment shows that the integrated controller can, precisely, achieve the desired average matching delay and CPU utilization under different update arrival rates.

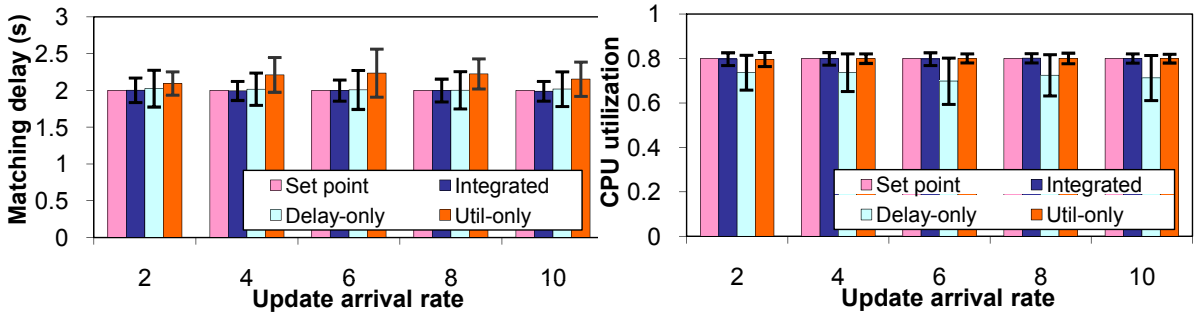
One of the advantages of the integrated controller is that it can simultaneously control both the matching delay and CPU utilization to achieve maximized system throughputs so that more valuable information can be timely disseminated. To highlight this advantage, we compare our integrated controller with two baselines: *Delay-only* and *Util-only*, in terms of control accuracy, overall throughput, and throughput for low-priority subscriptions.

Comparison with Delay-only: Delay-only is a SISO controller that is proposed in Section 3.1.2. It has a fixed batching window size and controls only the average matching delay of the high-priority subscriptions by manipulating the job budget. We tune it with

different batching window sizes to make it have the best overall throughput while controlling the average matching delay with an accuracy close to that of the integrated controller. We find that a window size of $1.95s$ produces the best results. We test Delay-only with different update arrival rates and plot the mean and standard deviation of 50 outputs in the steady state for average matching delay, CPU utilization, overall throughput, and throughput for low-priority subscriptions. As shown in Figures 3.18(a), the average matching delay of Delay-only with different arrival rates is all approximately equal to the set point with small oscillations (with the largest standard deviation as $0.25s$). However, as shown in Figure 3.18(b), Delay-only has a poor CPU utilization because it only controls matching delay. As a result, Delay-only has a poor overall throughput and a poor throughput for low-priority subscriptions, as shown in Figures 3.19(a) and (b).

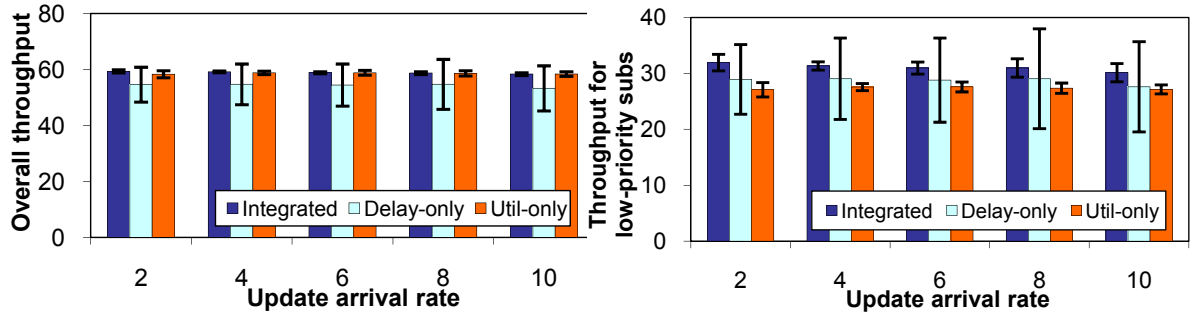
Comparison with Util-only: Util-only is another SISO controller that only controls CPU utilization by manipulating the job budget with a fixed batching window size. We first fix the batching window size for Util-only at the mean of the batching window size of the integrated controller in the steady state (*i.e.*, $1.8s$). We plot the mean and standard deviation of 50 outputs in the steady state for average matching delay and CPU utilization under different update arrival rates, as shown in Figures 3.18. We can see that Util-only violates the specified delay constraint though its CPU utilization converges to the set point with small oscillations. To have a fair comparison, we then tune Util-only with different batching window sizes so that it has an average matching delay close to the set point of $2s$. By iterative tuning and testing, we find that a batching window size of $1.6s$ gives us the best results. We then compare Util-only with the integrated controller in terms of overall throughput and throughput for low-priority subscriptions. As shown in Figures 3.19(a), since both Util-only and the integrated controller control the CPU utilization to the set point of 0.8, they have almost the same overall throughput. However, due to a smaller batching window size, Util-only has a smaller job budget, which results in a lower throughput for low-priority subscriptions, as shown in Figure 3.19(b).

The two experiments show that only controlling either matching delay or CPU utilization results in poor system throughputs or violates the specified delay requirement. In contrast,



(a) Matching delay (b) CPU utilization

Figure 3.18: Control accuracy comparison among Delay-only, Util-only, and the proposed integrated controller



(a) Overall throughput (b) Throughput for low-priority subscriptions

Figure 3.19: Throughput comparison among Delay-only, Util-only, and the proposed integrated controller

the integrated controller, by controlling both matching delay and CPU utilization, can meet the specified constraint for matching delay and achieve maximized possible system throughputs.

3.2 Real-Time Deadline Guarantees for Power Grid Computing

The power grid is a vital part of today's industrialized society, which relies upon the constant availability of high quality electrical power for proper functioning in industry, infrastructure, and domestic life. A power outage is likely to cause significant economic loss by disrupting the normal operating environment, which can be catastrophic. For example, the blackout that happened on August 14th, 2003 affected most of the northeastern United States and parts of Canada, and cost the United States between 4 billion USD and 10 billion USD [57].

3.2.1 Background

Blackouts are typically caused by two or more contingencies that occur in a short time period. A contingency is an unplanned event that a power device is destroyed or degraded, *e.g.*, a transmission line is tripped or a generator is unexpectedly halted. A single contingency typically does not necessarily cause the system to become unstable because careful contingency screening and corrective actions (*i.e.*, a load shedding or a transmission loading relief) in EMS can dynamically re-adjust the system so that it is able to accommodate the single contingency. If the system stability can be ensured even after the worst single contingency occurs, the system is said to be in the *N-1 secure state* [28], where "N-1" means the normal system minus one key element. However, if a second contingency occurs before corrective actions are taken, the system may become unstable and result in a local blackout. Even worse, once the system has become unstable, more and more contingencies could occur as the system encounters conditions outside its operational range, which may result in a cascade blackout.

Clearly, it is critical to take corrective actions before a second contingency occurs to avoid blackouts. The North American Electric Reliability Council (NERC) has specified that once a contingency occurs, corrective actions must be taken within 30 minutes [1]. In a typical EMS, a group of algorithms should be completed in a timely manner to determine corrective actions: State Estimation, Contingency Screening, and Optimal Power Flow, which can be collectively referred to as *Real-Time Operation (RTO)* [89]. Its primary goal is to determine the proper corrective actions so that the power grid can stay in the N-1 secure state. First, *State Estimation (SE)* receives real-time data collected from sensors in the field by a Supervisory Control And Data Acquisition (SCADA) system. It then ensures that the data is correct and estimates any missing measurements, so that a complete and accurate model of the system is available. Second, if a contingency is occurring, *Contingency Screening (CS)* takes the estimated model and determines the constraints required so that no single contingency may lead to system instability. Finally, *Optimal Power Flow (OPF)* (also known as Economic Dispatch) assesses the security constraints and determines the most economically efficient way to set the states of all the devices. These settings are then sent

to the SCADA, which forwards them to the field to fulfill the necessary corrective actions. Besides RTO, there are many other essential real-time tasks running in EMS. For example, *Automated Generation Control (AGC)* supervises and coordinates the power generators in real time [89]. *Short Term Operation (STO)* forecasts the customer power demand in the next hour and then optimally schedules the generation units. *Alarm Propagation (AP)* processes alarms to inform human operators of contingencies, so that corresponding measures can be taken manually to avoid system instability. Although these real-time tasks are not directly related to blackouts, continuously missing deadlines is detrimental to system stability. Most of the tasks introduced above are required to run on regular intervals in EMS by the NERC standard [1]. A list of typical real-time tasks, with their associated subtasks, in the EMS is presented in Table 3.2.

At present, many EMS run the power grid computing algorithms in an open-loop manner, so there is no guarantee that they will be completed by a given deadline. To guarantee the deadlines of those real-time tasks at runtime, several challenges must be faced. First, those real-time tasks are divided into multiple subtasks that are distributed among different processors in EMS and the execution of a task involves the execution of multiple subtasks under precedence constraints. To meet the end-to-end deadline, we need to meet the subdeadline of each subtask. Second, the computing system in EMS is an open and unpredictable environment. The workload in the system is system-dependent and time-varying, which cannot be accurately characterized a priori. Third, those tasks in the system have significantly different periods and execution times. For example, the period of AGC is usually in tens of seconds while the period of STO can be as long as thousands of seconds. Therefore, it is difficult to apply a single control algorithm to guarantee the deadline of all the tasks together.

In this subsection, we propose a control-based control solution which features a hierarchical control architecture by differentiating the tasks with relatively long periods from the ones with relatively short periods. We explore the different properties of the collection of real-time tasks in power grid computing and apply different control algorithms to guarantee their deadlines.

Table 3.2: Power grid computing tasks

	tasks	subtasks	execution time	period	type
1	Real-Time Operation (T_1)	State Estimation (T_{11})	$180s$	$500s$	long
		Contingency Screening (T_{12})	$100s$		
		Optimal Power Flow (T_{13})	$80s$		
2	Short Term Operation (T_2)	Load Forecasting (T_{21})	$120s$	$2000s$	long
		Unit Commitment (T_{22})	$120s$		
3	Automated Generation Control (T_3)	AGC Signal Request (T_{31})	$2s$	$[10s, 30s]$	short
		Automated Generation Control (T_{32})	$8s$		
4	Alarm Propagation (T_4)	Information Request (T_{41})	$2s$	$[4s, 20s]$	short
		Alarm Propagation (T_{42})	$2.5s$		
5	Voltage Security Assessment (T_5)	Voltage Security Assessment (T_{51})	$3s$	$[8s, 30s]$	short
6	Topology Analysis (T_6)	Topology Analysis (T_{61})	$4s$	$[8s, 30s]$	short

3.2.2 Proposed Solution

We first present an end-to-end task model commonly used for DRE systems such as power grid computing. We then describe our proposed solution.

As introduced in Section 3.2.1, the power grid computing tasks can be divided into multiple subtasks and each of them can run on different processors. The execution of a task involves the execution of multiple subtasks under precedence constraints (*i.e.*, CS runs after SE while OPF runs after CS.). Each subtask also runs periodically and shares the same period as the task that it belongs to (*i.e.*, CS and OPF should have the same period as SE). Based on these facts, we can model those tasks as follows. A system is comprised of m end-to-end periodic tasks, $\{T_i|1 \leq i \leq m\}$, executed on n processors, $\{P_i|1 \leq i \leq n\}$. Task T_i is composed of a chain of subtasks, $\{T_{ij}|1 \leq j \leq m_i\}$, that may be allocated to multiple processors. A subtask, T_{ij} $(1 < j \leq m_i)$, cannot be released for execution until its predecessor, T_{ij-1}, is completed. We assume that a non-greedy synchronization protocol (*e.g.*, release guard [58]) is used to enforce the precedence constraints between subsequent subtasks. Each subtask, T_{ij}, has an estimated execution time c_{ij} at the time of design. However, the actual execution time of a task may be significantly different from its estimation and varies at runtime.

Based on the task model above, the problem of meeting the deadline can be transformed into the problem of meeting the subdeadline of each subtask. A well-known approach for meeting the subdeadlines on a processor is by enforcing the *schedulable utilization bound* [58]. The subdeadlines of all the subtasks on a processor are guaranteed to be met if the

utilization of the processor remains below its schedulable utilization bound. Previous work [61] has proposed an End-to-end Utilization CONtrol (EUCON) algorithm to guarantee the deadline of end-to-end tasks in a DRE system. EUCON is a Multiple-Input-Multiple-Output (MIMO) controller designed based on the Model Predicative Control (MPC) theory. It guarantees the deadlines of all tasks by controlling the CPU utilization of each processor to be within a certain schedulability bound by dynamically adjusting the rates of all the end-to-end tasks in the system. EUCON can provide robust utilization guarantees simultaneously on multiple processors when task execution times deviate from estimation or vary significantly at runtime. However, it cannot be directly adopted in power grid computing since that algorithm is designed with the assumption that the periods of all the tasks have the same timescale.

As shown in Table 3.2, the periods of the tasks in power grid computing can have significantly different timescales, which range from tens of seconds to thousands of seconds. In this work, we refer to the tasks with relatively long periods as *long tasks* and the ones with relatively short periods as *short tasks*. Their subtasks are called *long subtasks* and *short subtasks*, respectively. The long subtasks can run iterative or non-iterative algorithms. A long iterative subtask uses an algorithm that achieves the specified computation accuracy after a certain number of iterations. We now discuss the characteristics of the tasks in power grid computing as follows:

- For short tasks, we assume that their rate can be dynamically adjusted within a range $[R_{min,i}, R_{max,i}]$. A short task running at a higher rate contributes a higher value to the application at the cost of higher CPU utilization. This is a reasonable assumption in power grid computing. For example, the more frequently AGC runs, the finer control it provides for power generation. Therefore, the adaptation of the rate adjustment can be incorporated into the CPU utilization control framework. In this work, we use EUCON [61] to control the CPU utilization of short subtasks to a certain budget on multiple processors simultaneously by adjusting the rates of each short task. Hereinafter, we refer to it as the ***short task utilization controller***.

- For long iterative tasks, the number of iterations in the algorithm can be adjusted. The higher the accuracy it needs, the more iterations it must run at the cost of higher CPU utilization. In power grid computing, computation accuracy can be normally compromised for task timeliness. This is a valid assumption in that a slightly less accurate result is better than a result that misses its deadline. Therefore, the adaptation of the number of iterations of long iterative subtasks can be incorporated into the CPU utilization control framework. In this work, we present a Single-Input-Single-Output (SISO) controller to control the CPU utilization of each long subtask to be within a certain budget. Since we only explore the property of the long iterative task in this work, *the long iterative task/subtask is referred to as the long task/subtask for simplicity* if not otherwise stated. Hereinafter, we refer to the controller of a long subtask as the ***long task utilization controller***.

The two utilization controllers (*i.e.*, the short task utilization controller and the long task utilization controller) control the CPU utilizations of each long subtask and all short subtasks to their respective set points while the total schedulability utilization bound is enforced. Clearly, the more budget allocated to the long subtask from the total schedulibility utilization bound, the better computation accuracy the long subtask can achieve. However, this will lead to lower rates for the short subtasks and hence the less value they contribute to the system. Therefore, it is important to determine how the total schedulability bound should be optimally partitioned between short subtasks and long subtasks so that the long subtasks can achieve the specified computation accuracy while the short subtasks can run as frequently as possible to contribute more value to the system. In order to achieve the desired budget allocation, we propose an upper-level controller that controls the computation accuracy of each long subtask to the specified level by allocating the CPU utilization budget between the short tasks and long tasks. The total allocable budget is calculated by subtracting the utilization of the long non-iterative tasks from the total utilization bound. The variation of CPU utilization of the long non-iterative tasks is regarded as system noise in this work. We refer to the upper-level controller as the ***accuracy controller***. In this work, the ***computation accuracy*** is defined as the absolute value of the common logarithm of the

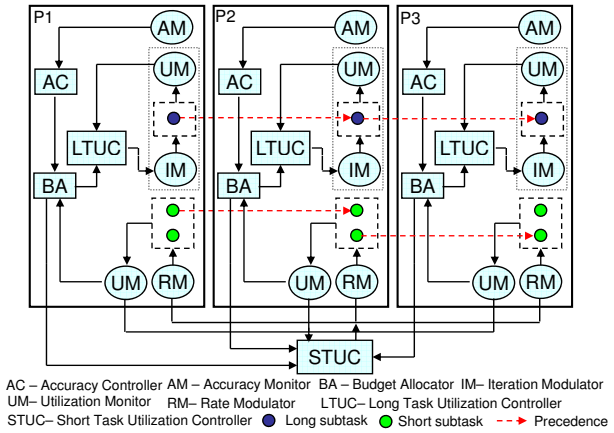


Figure 3.20: Hierarchical control architecture

computation error after each iteration. Hereinafter, we refer to the two utilization controllers as the lower-level controllers. The detailed system architecture is introduced in Section 3.2.3.

3.2.3 Hierarchical Control Architecture

As shown in Figure 3.20, the two lower-level control loops, *i.e.*, the short task utilization control loop and the long task utilization control loop, control the CPU utilizations of short subtasks and long subtasks to their respective utilization budgets. The upper-level control loop, *i.e.*, the accuracy control loop, controls the computation accuracy of the long subtask by dynamically adjusting the CPU utilization budgets of the two lower-level control loops so that the total CPU utilization is guaranteed to approach the schedulability bound.

Long Task Utilization Control Loop. As shown in Figure 3.20, the key components in the long task utilization control loop include a utilization monitor, a long task utilization controller and an iteration modulator. The control loop is invoked periodically with a period selected to include multiple instances of the long subtask under control. At the end of every control period, the monitor first measures the CPU utilization contributed by the long subtask and sends the value to the long task utilization controller. The CPU utilization taken by the long subtask is the *controlled variable* in the control loop. The controller then calculates the maximum number of iterations that the long subtask can run for the next control period, based on the difference between the set point and the measured utilization. The number of iterations is the *manipulated variable*. Finally the iteration modulator notifies

the long subtask of the maximum number of iterations when it creates new instances in the next control period. There is a long task utilization controller for each long subtask.

Short Task Utilization Control Loop. The short task utilization control loop maintains the desired CPU utilization of multiple processors simultaneously, despite the variation of execution time of the short subtasks at runtime. The key components in the short task utilization control loop include a centralized short task controller, a utilization monitor, and a rate modulator on each processor. The period of the control loop is selected so as to include multiple instances of the short task with the longest period. At the end of each control period, the utilization monitor on each processor measures the CPU utilization and sends the value to the centralized short task utilization controller. The CPU utilization of the short subtasks on each processor is the *controlled variable* in this control loop. The controller calculates the optimal change in the rate of each short task for the next control period and sends the value to each processor. The rate of each short task is the *manipulated variable* in this control loop. The rate modulator on each processor modifies the rates of the short tasks running on the processor, correspondingly. In this work, we use EUCON [61] as the short task utilization controller. There is one centralized short task utilization controller in the whole system.

Accuracy Control Loop. The key components in the accuracy control loop include an accuracy monitor, an accuracy controller, and a budget arbitrator. The control loop is also invoked periodically. The period is selected to be longer than the maximum settling time of the two lower-level controllers. This guarantees that the two lower-level controllers can always settle to their respective utilization budget before the new budget is set, so that the control loops are decoupled and can be designed independently. At the end of every control period:

1. The accuracy monitor measures the average computation accuracy (*i.e.*, the absolute value of the common logarithm of the error) of all instances of the long subtask and sends the value to the accuracy controller. The computation accuracy of the long subtask is the *controlled variable* in the control loop.

2. The controller calculates the appropriate CPU budget for the long task utilization controller based on the difference between the measured accuracy and the set point (*i.e.*, the specified accuracy). The budget for the short task controller is calculated by subtracting the budget of the long task utilization controller and the measured utilization of the long non-iterative subtasks from the total schedulability bound. The two budgets are sent to the budget arbitrator. At the same time, the average utilization of the short subtasks in the last control period is also sent to the budget arbitrator. The budget of the long task utilization controller is the *manipulated variable* in this control loop.

3. The budget arbitrator compares the average utilization of the short subtasks with its budget to decide whether the short task utilization controller is saturated, which means the utilization of the short subtasks cannot settle to the set point due to the range of their rates. If the short task controller is saturated, the budget of the short task controller is set as the current average utilization and the budget of the long task controller is calculated by subtracting the budget of the short task controller and the measured utilization of long non-iterative subtasks from the total schedulability bound. If it is not saturated, the arbitrator does not modify the two budgets. The two budgets are then sent to the two lower-level controllers. The goal of the budget arbitrator is to prevent the accuracy controller from allocating too much or too little budget to the long subtask so that the short task utilization controller saturates, which may result in system over-utilization (*i.e.*, violation of the schedulability utilization bound) or system under-utilization.

There is an accuracy controller for each long subtask. The hierarchical control architecture can be extended to handle multiple long subtasks on the same processor by sharing the same budget arbitrator. In the case of saturation of the short task utilization controller, the budget arbitrator allocates the remaining budget proportionally to the requested budget of each accuracy controller.

3.2.4 Controllers Design

Now we present the system modeling, design and analysis of the long task utilization controller and the accuracy controller. The design and analysis of the short task utilization controller can be found in [61].

Long Task Utilization Controller

We first introduce some notations. T_l is the control period. U_l is the set point. $p(k)$ is the period of the tasks/subtasks. For long tasks, $p(k)$ is fixed at runtime. $it(k)$ is the number of iterations that a long subtask runs. $u_l(k)$ is the CPU utilization of the long subtask, and $exec(k)$ is the average execution time of the subtask in the k^{th} control period.

The goal of the long task utilization controller is to control the CPU utilization of the long subtask to approach the set point. In the meantime, the long subtask should run as many iterations as possible to achieve better accuracy.

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.*, $u_l(k)$) and the manipulated variables (*i.e.*, $it(k)$). If the execution time of the long subtask is known, the CPU utilization that it takes is $u_l(k) = \frac{exec(k)}{p(k)}$. Since $exec(k)$ is proportional to the number of iterations that the long subtask runs, *i.e.*, $it(k)$, we have:

$$u_l(k) = \frac{exec\_per\_it(k)}{p(k)} it(k), \tag{3.19}$$

where $exec\_per\_it(k)$ is the execution time per iteration. Since the algorithm for the long subtask is known beforehand and we do not change the period of the long subtask at runtime, we can make the assumption that $exec\_per\_it(k) = exec\_per\_it$ and $p(k) = p$, where $exec\_per\_it$ and p are constants. As a result, we get the system model of the long task utilization controller as follows:

$$u_l(k) = k_u it(k-1), \tag{3.20}$$

where $k_u = \frac{exec\_per\_it}{p}$.

Controller Design and Analysis. Proportional-Integral (PI) control [26] can provide robust control performance despite considerable modeling errors. Based on the system model (3.20), we design a PI controller as follows:

$$it(k) = it(k-1) + K_1 e(k) - K_1 K_2 e(k-1), \tag{3.21}$$

where $e(k) = U_l - u_l(k)$ is the control error. Using the Root-Locus method [26], we can choose our control parameters as $K_1 = 1/k_u$ and $K_2 = 0$ such that our closed-loop transfer function is:

$$G(z) = z^{-1}. \tag{3.22}$$

Next, we reevaluate the control performance when the system (3.20) changes due to the variation of $exec\_per\_it$. Without loss of generality, we model the overall variation as g_u and get a real model at runtime:

$$u_l(k) = k'_u it(k), \tag{3.23}$$

where $k'_u = g_u \cdot k_u$. We apply the PI controller (3.21) on the real system model (3.23) to get the closed-loop transfer function at runtime as:

$$G(z) = \frac{g_u}{z - (1 - g_u)}. \tag{3.24}$$

Based on control theory, we can prove that the closed-loop system at runtime (3.24) is stable and has zero steady-state error only if $0 < g_u < 2$. This analysis shows that the long task utilization controller (3.21) can effectively control the CPU utilization of the long subtask despite significant variations of $exec\_per\_it$, as long as those variations do not result in a k'_u that is twice the nominal value of k_u.

Our above analysis gives us theoretical confidence in the performance of our controller and provides a guideline to choose the parameters in the nominal system model (3.20). For example, given the possible minimum and maximum values of k'_u for a certain algorithm, we can choose the nominal k_u to be a value slightly larger than $\frac{k'_{u,min}+k'_{u,max}}{2}$, such that the system is guaranteed to be stable even when the real model is unknown at the design time.

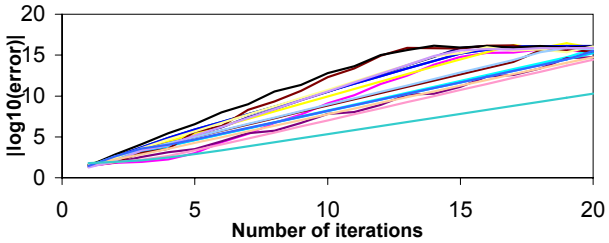


Figure 3.21: Linear relationship between the computation accuracy and the number of iterations

Accuracy Controller

We first introduce some notations. T_a is the control period. R_a is the accuracy set point. $error(k)$ is the computation error of the long iterative task, and $a(k)$ is the computation accuracy of the long subtask. Specifically, $a(k) = |log10(error(k))|$. $b_s(k)$ and $b_l(k)$ are the budgets for the short task utilization controller and the long task utilization controller, respectively.

The goal of the accuracy controller is to control the computation accuracy of the long subtask to the set point, *i.e.*, the required accuracy, by dynamically adjusting the allocated CPU utilization budget to the long subtask on this processor. The more budget is allocated to the long subtask, the higher computation accuracy it achieves.

System Modeling. Similar to the long task utilization controller, we need to model the dynamics of the controlled system, namely the relationship between the controlled variables (*i.e.*, $a(k)$) and the manipulated variables (*i.e.*, $b_l(k)$). Unlike the long task utilization controller, the system model cannot be derived analytically. However, though the algorithms that the long subtasks run are highly system dependent (*e.g.*, [92] and [13] propose two different state estimation algorithms.), an important observation from our experiments is that there exhibits an approximately linear relationship between the computation accuracy and the number of iterations that the algorithm runs. For example, Figure 3.21 plots the relationship between the number of iterations and the computation accuracy based on different input information for one of the algorithms used in the subtask of SE. Although the curves in Figure 3.21 have different slopes, all are approximately linear. Based on this observation, it is valid to assume that there exists a model between the computation accuracy

and the number of iterations as follows:

$$a(k) = k_a it(k-1), \tag{3.25}$$

where k_a is the nominal slope of the curve which models the relationship between the computation accuracy and the number of iterations. Since we already have the model between the CPU utilization and the number of iterations, presented in (3.20), we get a model between the utilization budget $b_l(k)$ and the computation accuracy $a(k)$ as follows:

$$a(k) = \frac{k_a}{k_u} b_l(k-1). \tag{3.26}$$

Controller Design and Analysis. Similar to the long task utilization controller, we design a PI controller as follows:

$$b_l(k) = b_l(k-1) + K_1 e(k) - K_1 K_2 e(k-1), \tag{3.27}$$

where $e(k) = R_a - a(k)$ is the control error, $K_1 = \frac{k_a}{k_u}$, and $K_2 = 0$.

If we model the variation of the parameter of k_a at runtime as g_a, we can depict the closed-loop system model at runtime as:

$$G(z) = \frac{g_a/g_u}{z - (1 - g_a/g_u)}. \tag{3.28}$$

Based on control theory, we can prove that the closed-loop system at runtime (3.28) is stable and has zero steady-state error only if $0 < g_a/g_u < 2$. Given an iterative algorithm, if we carefully choose the nominal values of k_a and k_u, we can make sure the closed-loop system is stable at runtime despite the variation of the computation accuracy after each iteration.

3.2.5 Simulation Results

In this subsection, we first introduce the simulation environment. We then demonstrate the performance of the hierarchical control solution, and then compare it with two baselines.

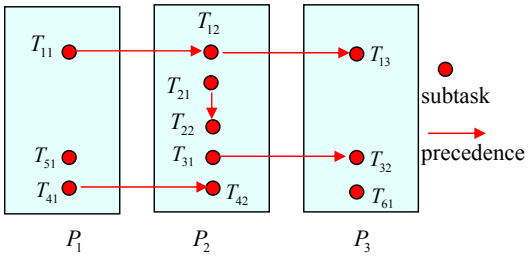


Figure 3.22: Workload configuration

In all the experiments, the simulation clock is equal to $10ms$ for higher precision. We only present the deadline miss ratio of the task of RTO. This is because its goal is to determine the corrective actions when a contingency occurs and continuous deadline misses of RTO may directly lead to blackouts.

Simulation Environment

Our simulation environment is developed based on the EUCON simulator [61] by adding the accuracy control loop and the long task utilization control loop. The simulator, the accuracy control loop, and the long task utilization control loop are implemented in C++, while the short task utilization control loop is implemented in MATLAB using the *lsqlin* least squares solver.

Now we introduce the workload configuration, deadline assignment and period assignment as follows.

Workload Configuration. The configuration of those subtasks among the processors in EMS is highly system dependent. As an example, we configure the tasks listed in Table 3.2 among three processors in the simulator, which is shown in Figure 3.22. However, the hierarchical control solution can be easily reconfigured to handle different configurations. The subtasks on each processor are scheduled by the Rate Monotonic Scheduling (RMS) algorithm [58]. The precedence constraints among subtasks are enforced by the release guard protocol [58]. Since the algorithms of the subtasks of STO vary among different systems, we assume that they are long non-iterative subtasks for a typical setup in this work. For example, [88] propose a non-iterative estimation scheme for the subtask of Load Forecasting. Since our objective is to highlight the hierarchical control solution, we also assume that there

is only one long iterative subtask on each processor to simplify the setup. This means that there is only one long task utilization controller and accuracy controller on each processor. However, the hierarchical control solution can be easily extended to support multiple long subtasks on a processor as introduced in Section 3.2.3.

Deadline Assignment. The deadline of RTO should be within 30 minutes, according to the NERC standard [1]. However, the deadline assignment for other tasks is system dependent according to the power grid under control. In this work, we use a typical assignment, which is shown in Table 3.2. The end-to-end deadline of each task is evenly divided into subdeadlines for each subtask based on the number of subtasks that the task has.

In the simulator, when a subtask is ready to run, it first checks whether the task to which it belongs has missed the deadline or not. If the deadline has already been missed, the current instances of all subtasks in this task will be terminated and new instances will be started. This is important in power grid computing. Since the algorithm in the subtask may encounter dead loops due to unpredictable errors, continuing the task which has missed its deadline may lead to over-utilization of the processor and continuous deadline misses.

Period Assignment. In this work, we decide the period of each task based on their deadline requirement. According to the theory of RMS, each task's period $p_i = d_i/n_i$, where n_i is the number of subtasks in task T_i. d_i is the deadline of T_i. Therefore, the resultant period of each task T_i equals the subdeadline of its subtasks. Hence, the schedulable utilization bound of RMS is the total utilization bound on each processor [58]:

$$B_i = m_i(2^{1/m_i} - 1), \tag{3.29}$$

where m_i is the number of subtasks on P_i. All (sub)tasks meet their (sub)deadlines if the utilization bound on every processor is enforced. For example, P_1 in the setup shown in Figure 3.22 accommodates 3 subtasks and thus, its utilization bound is 0.7798.

Based on the task periods, we select the periods of the control loops as follows. The control period of the short task utilization control loop is set as $100s$ so that at least 4

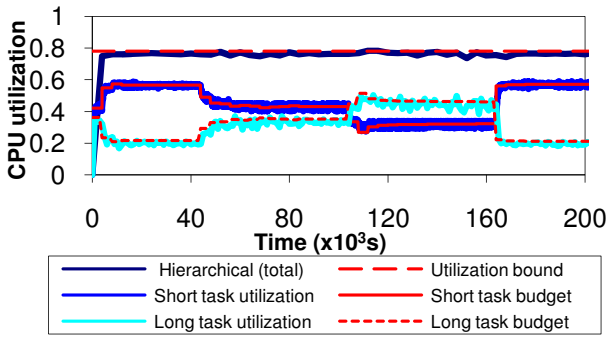


Figure 3.23: CPU utilization of the hierarchical control solution (P_1)

instances of the short task are included. The control period of the long task utilization control loop and the accuracy control loop are set as $1,000s$ and $4,000s$ to include at least 2 and 8 instances of the task of RTO, respectively.

Performance of the Hierarchical Control

In this experiment, one of the long subtasks increases the number of iterations required to achieve the specified accuracy at runtime. This is a common scenario to most power grid computing systems. For example, the computation accuracy of SE depends highly on the quality of the sensor data collected from the field. The greater the noise, the more number of iterations required to achieve the specified accuracy. Similarly, the computation accuracy of OPF relies on the complexity of the constraints estimated by CS. Without loss of generality, we choose SE to increase the required number of iterations at runtime in this experiment.

Figures 3.23 and 3.24 show the CPU utilization of the processor on which SE is running (*i.e.*, P_1) and the computation accuracy of SE, respectively. At the beginning, the required number of iterations is 6. We can see that the budget of the long task utilization controller decreases since the initial budget is allocated by assuming that the required number of iterations is 10. At time $40,000s$, the required number of iterations increases from 6 to 10, due to the degraded quality of the sensor data. The budget allocation between the long task utilization controller and the short task utilization controller is adjusted by the accuracy controller correspondingly so that the specified accuracy of 6 is still achieved (as shown in Figure 3.24). At time $100,000s$, the required number of iterations further increases to 20 due to the continuously degraded quality of the data. To achieve the specified accuracy, the

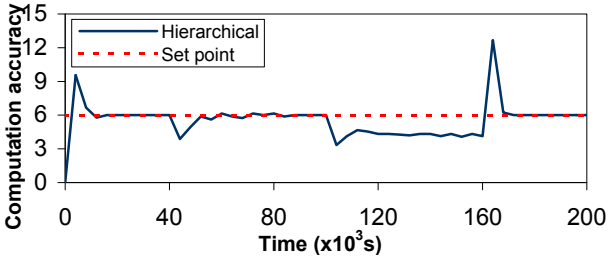


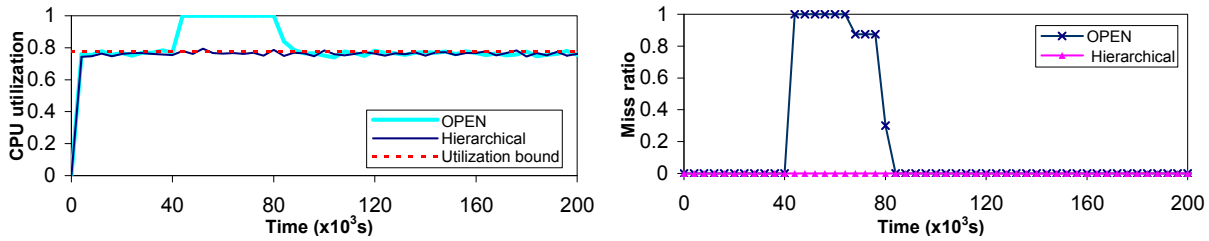
Figure 3.24: Computation accuracy of the hierarchical control solution (Subtask SE)

accuracy controller further increases the budget of the long subtask and decreases the budget of the short subtasks, making the short utilization controller reach saturate. However, the budget arbitrator detects the saturation and throttles the budget of the short task utilization controller with the average utilization of the short subtasks in the last control period. Hence, the long subtask cannot get adequate budget and has to terminate the iteration before it achieves the specified accuracy. This results in a computation accuracy of 4.3 as shown in Figure 3.24. At time $160,000s$, the quality of data returns to the normal level and the required number of iterations decreases to 6. The budget allocation is adjusted and the long subtask achieves the specified accuracy again. During the whole run, the two lower-level utilization controllers effectively control the utilizations of the long subtask (*i.e.*, SE) and the short subtasks to approach their respective set points. As a result, the total schedulability utilization bound is guaranteed so that deadline misses are avoided.

Based on the results, we can conclude that the hierarchical control solution can dynamically adjust budget allocation between the long subtask and the short subtasks in response to variations, and effectively guarantee the total utilization bound to help avoid blackouts.

Comparison with OPEN

OPEN is an open-loop algorithm that employs fixed rates for the short tasks, and runs long subtasks until the specified computation accuracy is achieved. OPEN first estimates the number of iterations that the long subtask needs to run to achieve the specified accuracy. It then allocates the utilization budgets between the long subtask and the short subtasks in a static way. Based on the allocated budgets, it calculates the periods of the short tasks a



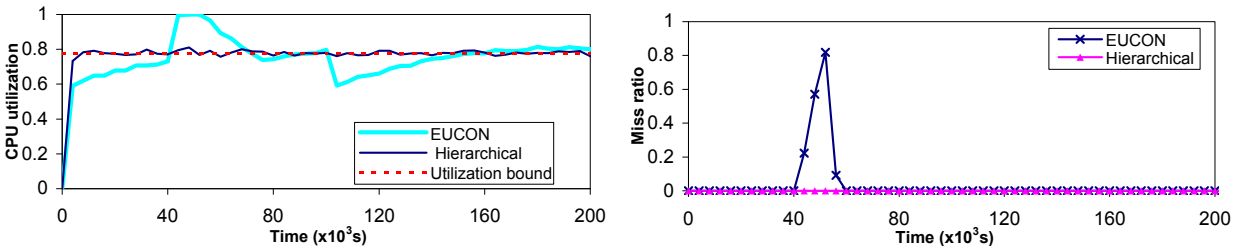
(a) Total utilization (P_1) **(b)** Miss ratio (Task RTO)

Figure 3.25: Comparison with OPEN

priori based on the estimated execution time. OPEN can achieve the utilization bound and maximize the system performance (*e.g.*, running the short tasks as frequently as possible) if all of the estimated information is accurate.

Similar to the previous scenario, the required number of iterations to achieve the specified accuracy of 6 for the subtask of SE has a sharp increase from 10 to 20 at time $40,000s$. Figure 3.25(a) shows the total utilization of the processor on which SE is running (*i.e.*, P_1) for both OPEN and the hierarchical control. Figure 3.25(b) shows the miss ratio for the task of RTO which SE is a subtask of. As shown in Figure 3.25(a), OPEN indeed achieves the utilization bound at the beginning. However, OPEN keeps violating the total utilization bound after the abrupt variation occurs until the quality of the sensor data returns to the normal level at time $80,000s$. This is because OPEN has no feedback information from this abrupt variation and the long subtask of SE still runs until the specified accuracy is achieved. As a result, the task of RTO continuously misses its deadline due to the violation of the total utilization bound as shown in Figure 3.25(b). This is a very dangerous situation for the power grid. If a contingency occurs during this time, the power grid is vulnerable to more contingencies since corrective actions cannot be taken timely. As a result, there will be a high probability of a blackout (*e.g.*, the blackout on Aug 14th, 2003).

In contrast, the CPU utilization of the hierarchical control solution is controlled to stay around the schedulability bound, as shown in Figure 3.25(a). This is because the accuracy controller and the budget arbitrator can allocate the total utilization budget between the long subtask and the short subtasks in response to variations. In addition, both the long task controller and the short task controller can effectively control the CPU utilization to approach



(a) Total utilization (P_1) **(b)** Miss ratio (Task RTO)

Figure 3.26: Comparison with EUCON

their respective allocated budgets as shown in Figure 3.23, so that the total utilization bound is enforced. Accordingly, the power grid computing system misses no deadlines and blackouts can be avoided.

Comparison with EUCON

EUCON relies on a single control algorithm to control all end-to-end tasks in the system. The control period of EUCON is selected so as to include multiple instances of the task with the longest period. However, the real-time tasks in the power grid computing system have periods with very different timescales. As a result, EUCON takes unacceptably long time to respond to any variations of the short subtasks. One advantage of the hierarchical control solution over EUCON is that the hierarchical control solution uses different controllers for tasks with periods in different timescales (*i.e.*, the long utilization controller and the short task utilization controller) so that it can quickly respond to any variations of the short subtasks.

To highlight the advantage, we run experiments to simulate a scenario common to typical power grid computing systems, in which the execution time of one of the short subtasks

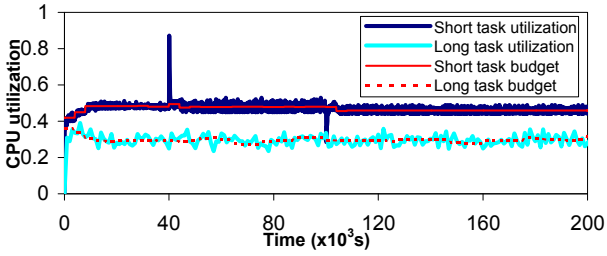


Figure 3.27: Utilization for short tasks and long tasks in the hierarchical control solution

increases dramatically at runtime due to accidental reasons. For example, in the blackout on Aug 14th, 2003, the execution time of AP abruptly increased. In this experiment, the period of EUCON is set to the same as the period of the accuracy controller (*i.e.*, $4,000s$) to include at least 2 instances of the task with the longest period (*i.e.*, STO). We also assume that the periods of the two long tasks (*i.e.*, RTO and STO) can be adjusted within a certain range. Figures 3.26(a) and (b) show the total utilization of the processor which accommodates one of the subtasks of AP (*i.e.*, P_1) and the miss ratio of the task of RTO, respectively. The execution time of both of the subtasks of AP increases to 2.5 times of their estimated execution time from $4,000s$ to $100,000s$. As shown in Figure 3.26(a), the abrupt increase leads to over-utilization of the system controlled by EUCON. EUCON takes approximately $20,000s$ (5 control periods which are around 5.5 hours) to respond to the abrupt variation by adjusting the periods of all tasks due to its long control period. Subsequently, the long duration of over-utilization leads to continuous deadline misses for the task of RTO as shown in Figure 3.26(b). If there is a contingency during this time, the power grid will be vulnerable to additional contingencies and hence, there is a high probability of a blackout due to delayed corrective actions.

In contrast to the slow response of EUCON, the short task utilization controller in the hierarchical control responds to the sudden variation quickly within approximately $500s$. This is because the short task utilization controller has a period much shorter than EUCON and quickly responds to the variation by adjusting the periods of all short tasks in the system. In addition, the total CPU utilization sampled with the same period of EUCON is only minimally affected by this abrupt increase as shown in Figure 3.27. As a result, the task of RTO has no deadline misses shown in Figure 3.26(b). Note that this abrupt variation may temporarily lead to several deadline misses for the short tasks which have subtasks on the same processor that AP is has subtasks running on, due to a short duration violation of the total utilization bound. However, these few deadline misses can be tolerated by most power grid computing systems in that effective corrective actions are taken timely when a contingency occurs. Therefore, no blackouts should happen due to this variation.

Chapter 4

Adaptive Power Management

In recent years, power has become the most important concern for enterprise data centers that host thousands of computing servers and provide outsourced commercial IT services. For example, running a single high-performance 300 W server for one year could consume 2628 KWh of energy, with an additional 748 KWh in cooling this server [8]. The total energy cost for this single server would be $338 a year without counting the costs of air conditioning and power delivery subsystems [8]. On the other hand, as modern data centers continue to increase computing capabilities for growing business requirements, high-density servers become more and more desirable due to real-estate considerations and better system management features.

An effective way to reduce energy consumption of blade servers is to transition the hardware components from high-power states to low-power states whenever possible [8]. Most components in a modern blade server such as processors [69, 50], main memory [17, 24] and disks [30] have adjustable power states. Components are fully operational, but consume more power in high-power states while having degraded functionality in low-power states [8]. An energy-efficient server design is to have run-time measurement and control of power to adapt to a given *power budget* so that we reduce the power (then the performance) of the components when actual power consumption of the server exceeds the budget [50]. As a result of controlling power consumption, we can have the maximum server performance while not using more power than what power supplies can provide. Even though servers

in a data center are usually provisioned to have their peak power consumption lower than the capacity of power supplies, it is particularly important for a system with multiple power supplies to be able to reduce its power budget at runtime in the case of a partial failure of its supply subsystem.

Traditionally, adaptive power management solutions heavily rely on heuristics. Recently, feedback control theory has been successfully applied to power control for a single server [69, 74, 91]. For example, recent work [50] has shown that control-theoretic power management outperforms a commonly used heuristic-based solution by having more accurate power control and better application performance. In this chapter, we propose control-based adaptive power management that controls the power consumption to the desired power budget, at three different levels: cluster level, data center level, and server level.

4.1 Cluster-level Power Management

Currently, the widely used high-density servers from major vendors (*e.g.*, IBM and HP) are so-called *blade servers* which pack traditional multi-board server hardware into a single board. Multiple servers are then put into a *chassis* (also called an enclosure) which is equipped with common power supplies and various ports. The greatest immediate concerns about blade servers are their power and cooling requirements, imposed by limited space inside the server chassis. The increasing high-density may also lead to a greater probability of thermal failure and hence require additional energy cost for cooling [71].

In this section, we propose a novel MIMO control algorithm, based on the well-established Model Predictive Control (MPC) theory, to provide a control-theoretic solution for managing the power of *multiple* servers in a small-scale cluster (*e.g.*, a chassis) of a data center.

4.1.1 Cluster-level Power Control Loop

In this subsection, we give a high-level description of our power control loop that adaptively manages the power consumption of a server cluster by manipulating the *clock frequency* of the processor of each server in the cluster.

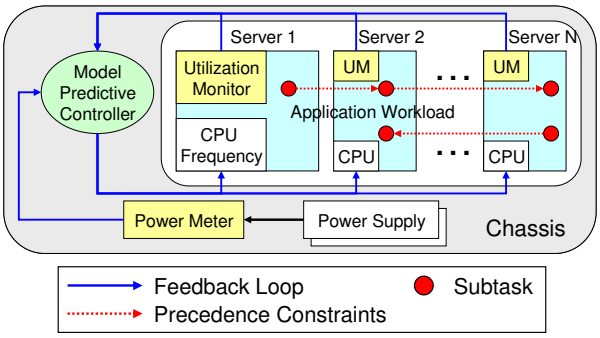


Figure 4.1: Power control loop for a small-scale cluster

There are two reasons for us to use processor throttling as our actuation method in this work. First, processors typically have well-documented interfaces to adjust frequency levels. Second, processors commonly contribute the majority of total power consumption of small form-factor servers [9]. As a result, the processor power difference between the highest and lowest power/performance states is large enough to compensate for the power variation of other components and thus to provide an effective way to support a power budget reduction in the event of power supply failure. In Chapter 5, we propose to use memory throttling as one of the actuation methods to capping the power consumption at the server level.

As shown in Figure 4.1, the key components in the control loop include a centralized *controller* and a *power monitor* at the cluster level, and a *CPU utilization monitor* and a *CPU frequency modulator* on each server. The control loop is invoked periodically and its period is chosen based on a trade-off between actuation overhead and system settling time [50]. The following feedback control loop is invoked at the end of every control period: 1) The power monitor (*e.g.*, a power meter) measures the average value of the total power consumption of all servers in the last control period and sends the value to the controller through its feedback lane. The total power consumption is the *controlled variable* of the control loop. 2) The utilization monitor on each processor sends its CPU utilization in the last control period to the controller. The utilization values can be used by the controller to optimize power allocation in the next control period. 3) The controller collects the power value and utilization vector, computes the new CPU frequency level for the processor of each server, and then sends the level to the CPU frequency modulator on each server through its

feedback lane. The *CPU frequency levels* are the manipulated variables of the control loop. 4) The CPU frequency modulator on each server changes the CPU frequency level of the processor accordingly.

4.1.2 System Modeling

Now we analytically model the power consumption of the server cluster. We first introduce several notations. T is the control period. $p_i(k)$ is the power consumption of Server i in the k^{th} control period. $f_i(k)$ is the frequency level of the processor of Server i in the k^{th} control period. $d_i(k)$ is the difference between $f_i(k+1)$ and $f_i(k)$, *i.e.*, $d_i(k) = f_i(k+1) - f_i(k)$. $u_i(k)$ is the CPU utilization of Server i in the k^{th} control period. N is the total number of servers in the cluster. $tp(k)$ is the total power consumption of the whole cluster, *i.e.*, $tp(k) = \sum_{i=1}^{N} p_i(k)$. P_s is the power set point, *i.e.*, the desired power constraint of the cluster. The control goal is to guarantee that $tp(k)$ converges to P_s within a given settling time.

Using the system identification approach, we have observed that the power consumption of a server changes immediately as the clock frequency changes. This is consistent with the observation presented in [50] that power consumption changes within a millisecond after a processor changes its performance state. Since a power sampling period is usually hundreds of milliseconds or even seconds, power consumption can be regarded to be determined exclusively by the current clock frequency and independent of the power consumption in the previous control periods. Figure 4.2 plots the average power consumption of the four servers used in our experiments at five available CPU frequency levels, which are represented as a fraction of the highest level. The workload used to do system identification is Linpack, which is introduced in detail in Section 4.1.5. Server 2 to 4 are almost identical while Server 1 has slightly different components. Two linear models fit well ($R^2 > 96\%$) for Server 1 and the other three servers, respectively. In general, our system model of power consumption is:

$$p_i(k) = A_i f_i(k) + C_i \tag{4.1}$$

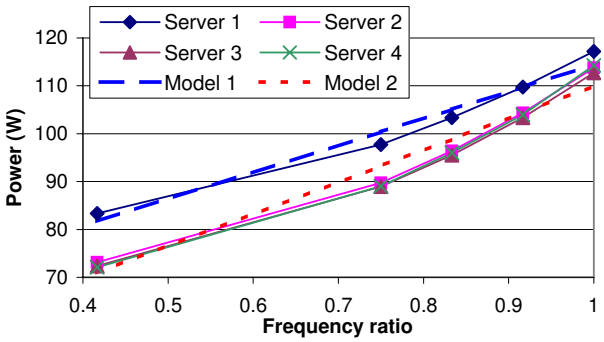


Figure 4.2: Power models of four servers

where A_i is a generalized parameter that may vary for different servers. The dynamic model of the system as a difference equation is:

$$p_i(k+1) = p_i(k) + A_i d_i(k) \tag{4.2}$$

To verify the accuracy of our system models, we stimulate the servers with pseudo-random digital white-noise inputs [26] to change the CPU frequency every five seconds in a random fashion. We then compare the actual power consumption with the values predicted by our model. Figure 4.3 shows that the predicted output by Model 1 is adequately close to the actual power output of Server 1. The predicted output of a second-order model is just slightly better than that of the first-order model by only having a 3.8% variation difference. We use the first-order model (4.2) in this work to simplify the controller design.

Based on (4.2), we now consider the total power consumption of all servers in a cluster. Their power consumptions can be modeled in the matrix form:

$$\mathbf{p(k+1)} = \mathbf{p(k)} + \mathbf{Ad(k)} \tag{4.3}$$

where, $\mathbf{p(k)} = \begin{bmatrix} p_1(k) \\ \vdots \\ p_N(k) \end{bmatrix}$,

$\mathbf{A} = \begin{bmatrix} A_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & A_N \end{bmatrix}$, $\mathbf{d(k)} = \begin{bmatrix} d_1(k) \\ \vdots \\ d_N(k) \end{bmatrix}$.

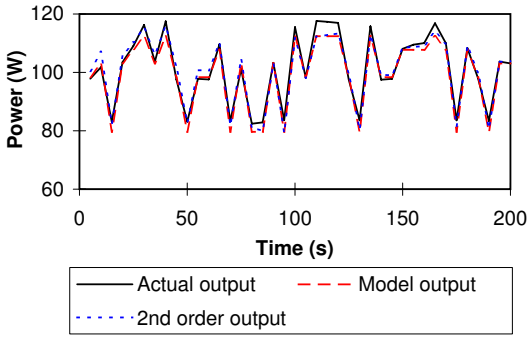


Figure 4.3: Comparison between predicted power output and actual power output

The total power consumption, $tp(k+1)$, is the summation of the power consumed by each individual server.

$$tp(k+1) = tp(k) + \begin{bmatrix} A_1 & \dots & A_N \end{bmatrix} \begin{bmatrix} d_1(k) \\ \vdots \\ d_N(k) \end{bmatrix} \tag{4.4}$$

Note that each A_i is the *estimated* system parameter resulted from system identification using a typical workload (*i.e.*, Linpack). The *actual* value of A_i in a real system may change for different workloads and is *unknown* at design time. However, in Section 4.1.3, we show that a system controlled by the controller designed with the estimated parameters can remain stable as long as the variation of A_i is within an allowed range.

4.1.3 Control Design and Analysis

We apply the *Model Predictive Control* (MPC) theory [64] to design the controller based on the system model (4.4). MPC is an advanced control technique that can deal with coupled MIMO control problems with constraints on the plant and the actuators. This characteristic makes MPC well suited for power control in server clusters.

MPC Controller Design

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 sampling periods, called the *prediction horizon*. The control objective is to select an input trajectory that minimizes the cost function while satisfying the constraints. An input trajectory includes the control inputs in the following M sampling periods, $\mathbf{d(k)}$, $\mathbf{d(k+1|k)}$, ... $\mathbf{d(k+M-1|k)}$, 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 $\mathbf{d(k)}$ is applied as the control input to the system. At the end of the next sampling period, the prediction horizon slides one sampling period and the input is computed again based on the feedback $tp(k)$ from the power 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 sampling period, the controller computes the control input $\mathbf{d(k)}$ that minimizes the following cost function under constraints.

$$V(k) = \sum_{i=1}^{P} \|tp(k+i|k) - ref(k+i|k)\|^2_{Q(i)} + \sum_{i=0}^{M-1} \|\mathbf{d(k+i|k)} + \mathbf{f(k+i|k)} - \mathbf{F_{max}}\|^2_{\mathbf{R(i)}} \tag{4.5}$$

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

$$ref(k+i|k) = P_s - e^{-\frac{T}{T_{ref}}i}(P_s - tp(k)) \tag{4.6}$$

where T_{ref} is the time constant that specifies the speed of system response. A smaller T_{ref} causes the system to converge faster to the set point but may lead to larger overshoot. By minimizing the tracking error, the closed-loop system will converge to the power set point P_s if the system is stable. The second term in the cost function (4.5) represents the *control penalty*. The control penalty term causes the controller to optimize system performance by minimizing the difference between the highest frequency levels, $\mathbf{F_{max}}$ and the new frequency levels, $\mathbf{f(k+i+1|k) = d(k+i|k) + f(k+i|k)}$ along the control horizon. The control weight vector, $\mathbf{R(i)}$, can be tuned to represent preference among servers. For example, a higher weight may be assigned to a server if it has heavier or more important workload so that the controller can give preference to increasing its frequency level. As a result, the overall system performance can be optimized.

This control problem is subject to three constraints. First, the CPU frequency of each server should be within an allowed range (*e.g.*, Intel Xeon processor only has eight states). Second, two or more servers that run the same application service may have the same frequency level. Third, the total power consumption should not be higher than the desired power constraint. The three constraints are modeled as:

$$
\begin{aligned}
& F_{min,j} \leq d_j(k) + f_j(k) \leq F_{max,j} \qquad (1 \leq j \leq N) \\
& d_i(k) + f_i(k) = d_j(k) + f_j(k) \\
& tp(k) \leq P_s
\end{aligned}
$$

Based on the above analysis, cluster-level power management has been modeled as a constrained MIMO optimal control problem. The controller must minimize the cost function (4.5) under the three constraints. This constrained optimization problem can be easily transformed to a standard constrained least-squares problem [64]. The controller uses a standard least-squares solver to solve the optimization problem on-line. In our system, we implement the controller based on the `lsqlin` solver in Matlab. `lsqlin` uses an active set method similar to that described in [6]. The computational complexity of `lsqlin` is polynomial in the number of servers and the control and prediction horizons.

Stability Analysis

A fundamental benefit of the control-theoretic approach is that it gives us theoretical confidence for system stability, even when the system model (*i.e.*, system parameter A_i) may change for different workloads. We say that a system is *stable* if the total power $tp(k)$ converges to the desired set point P_s, that is, $\lim_{k\to\infty} tp\,(k) = P_s$. Our MPC controller solves a finite horizon optimal tracking problem. Based on optimal control theory [52], the control decision is a linear function of the current power value, the power set point of the cluster, and the previous decisions for CPU frequency levels.

We now outline the general process for analyzing the stability of a server cluster when the actual system model is different from the model resulted from system identification (*i.e.*, with different A_i). First, given a specific system, we derive the control inputs $\mathbf{d}(\mathbf{k})$ that minimize the cost function based on the *estimated* system model (4.4) with estimated parameters $\mathbf{A}$. The control inputs represent the control decision based on the estimated system model. Second, we construct the *actual* system model by assuming the actual parameter $A_i' = g_i A_i$, where g_i represents the unknown system gain. The stability analysis of the actual system needs to consider a composite system consisting of the dynamics of the original system and the controller. Third, we then derive the closed-loop system model by substituting the control inputs derived in the first step into the actual system model. Finally, we analyze the stability of the closed-loop system by computing all the poles of the closed-loop system. According to control theory, if all poles locate inside the unit circle in the complex space and the DC gain matrix from the control to the state is the identity matrix, the state of the system, *i.e.*, the total power consumption, will converge to the set point. The allowed variation range of g_i can be established by computing the values of g_i that cause the poles to move across the unit circle.

The detailed steps and a complete stability proof for the example cluster used in our experiments can be found in the appendix. Our results show that the system can remain stable as long as the variation of A_i is within an allowed range. In addition, a Matlab program is developed by us to perform the above stability analysis procedure automatically. In our stability analysis, we assume the constrained optimization problem is feasible, *i.e.*, there

exists a set of CPU frequency levels within the acceptable ranges that can make the total power consumption equal to its set point. If the problem is infeasible, no control algorithm can guarantee the set point through CPU frequency adaptation. In that case, the system may need to integrate with other adaptation mechanisms (e.g., disk or memory throttling).

4.1.4 Implementation

Our testbed includes a cluster composed of 4 Linux servers to run workloads and a Linux desktop machine to run the controller. The four servers are equipped with 2.4GHz AMD Athlon 64 3800+ processors with 1GB RAM and 512KB L2 Cache. The controller machine is a Dell OptiPlex GX520 with 3.00GHz Intel Pentium D Processor and 1GB RAM. All the machines are connected via an internal switch. The 4 servers run openSUSE Linux 10.2 with kernel 2.6.18 while the controller machine runs SUSE Linux 10.1 with kernel 2.6.16.

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

Power Monitor: The power consumption of each server in the cluster is measured with a WattsUp Pro power meter [21] by plugging the server into the power meter and then connected to the a standard 120-volt AC wall outlet. The WattsUp power meter has an accuracy of $\pm 1.5\%$ of the measured value. To access power data, the data port of each power meter is connected to a serial port of the controller machine. A system file is then generated for power reading in Linux systems. The power meter samples power data every 1 second and responds to requests by writing all new readings after last request to the system file. The controller then reads the power data from the system file and conducts the control computation.

Utilization Monitor: The utilization monitor uses the */proc/stat* file in Linux to estimate the CPU utilization in each control period. The */proc/stat* file records the number of jiffies (usually 10ms in Linux) when the CPU is in user mode, user mode with low priority (nice), system mode, and when used by the idle task, since the system starts. At the end of each sampling period, the utilization monitor reads the counters, and estimates the CPU

utilization as 1 minus the number of jiffies used by the idle task in the last sampling period divided by the total number of jiffies in the same period.

Controller: The controller is implemented as a multi-thread process. The main thread uses a timer to periodically invoke the control algorithm presented in Section 4.1.3, while the child thread employs the *select* function to get CPU utilizations from all the servers in the cluster. Every time the periodic timer fires, the controller requests a new power reading and the utilizations of all the servers in the last control period, and then invokes a Matlab program to execute the control algorithm presented in Section 4.1.3. As the outputs of the control algorithm, new CPU frequency levels are calculated and sent to the CPU frequency modulator on each server to enforce in the next control period.

CPU Frequency Modulator: We use AMD's Cool'n'Quiet technology [4] to enforce the new CPU frequency. The AMD Athlon 64 3800+ microprocessor has 5 discrete CPU frequency levels and can be extended to have 8 levels. We use 5 levels in this work. 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*. A BIOS routine periodically checks this file and resets the CPU frequency accordingly. The average overhead (*i.e.*, transition latency) for the BIOS to change frequency in AMD Athlon processors is about $100\mu s$ according to the AMD white paper report [4].

Since the new CPU frequency level periodically received from the controller is a floating-point (fractional) value, the modulator code must locally resolve this to a series of discrete frequency values to approximate the fractional value. For example, to approximate 3.2 during a control period, the modulator would output the sequence 3, 3, 3, 3, 4, 3, 3, 3, 3, 4, etc on a smaller timescale. To do this, we implement a first-order delta-sigma modulator [50], which is commonly used in analog-to-digital signal conversion. The detailed algorithm of the first-order delta-sigma modulator can be found in [50]. Clearly, when the sequence has more numbers during a control period, the approximation will be better but the actuation overhead may become higher. In this work, we choose to use 50 discrete values to approximate the fractional frequency level, which leads to a subinterval of 100ms during an example control

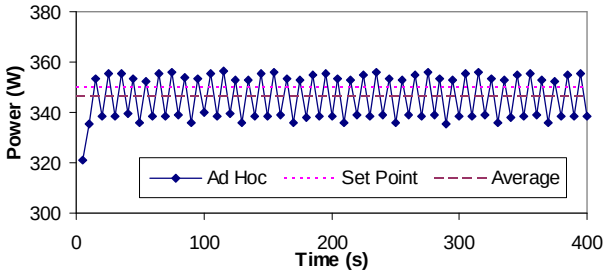


Figure 4.4: A typical run of Ad Hoc

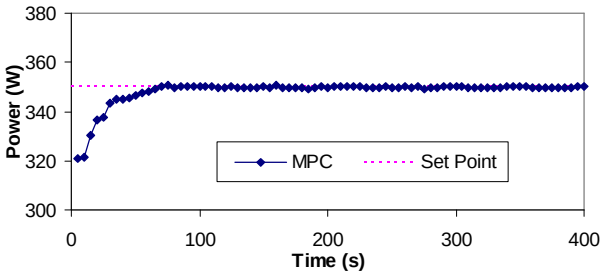


Figure 4.5: A typical run of the MPC controller

period of 5s. As a result, the effect of actuation overhead on system performance is no more than 0.1% ($100\mu s/100ms$) even in the worst case when the frequency needs to be changed in every subinterval. This amount of overhead is acceptable to most computer systems.

4.1.5 Empirical Results

In this subsection, we first introduce two state-of-the-art baselines: a MIMO ad hoc controller and a single-input-single-output (SISO) controller. We then discuss the benchmark used in our experiments and the experimental set-up. Finally, we compare our MPC MIMO controller against the two baselines, in terms of control accuracy and application performance.

Baselines, Benchmark and Set-up

We use two state-of-the-art controllers, referred to as Ad Hoc and SISO, as baselines in our experiments. Ad Hoc is a heuristic-based controller designed for cluster-level power control, which is adapted (with minor changes) from the preemptive control algorithm presented in a recent paper [73]. Ad Hoc represents a typical industry solution to power control of a server cluster. We compare our controller against Ad Hoc to show that a well-designed ad hoc controller may still fail to have accurate power control and thus lead to degraded application performance. The control scheme of Ad Hoc is briefly summarized as follows.

1. Start with the processors of all servers throttled to the lowest frequency level;
2. In each control period, (i) if the total power consumption is lower than the set point, choose the server with the highest CPU utilization to increase its frequency level by

one; or (ii) if the power reading is above the set point, choose the server with the lowest CPU utilization to decrease its frequency level by one; (iii) in steps i and ii, if all servers have the same CPU utilization, choose a server in a round-robin fashion.

3. Repeat step 2 until system stops.

A fundamental difference between Ad Hoc and our MPC controller is that Ad Hoc simply raises or lowers the clock frequency level by one step, depending on whether the measured power is lower or higher than the power set point. In contrast, MPC computes a fractional frequency level based on well-established control theory and uses the frequency modulator to approximate this output with a series of discrete frequency levels.

The second baseline, SISO, is a control-theoretic controller designed to control the power consumption of a *single* server, which is also presented in a recent paper [50]. In contrast to our MPC controller, SISO is a proportional (P) controller designed based on the system model of a single server (4.2). With SISO, a separate controller is used on each server to control its power *independently* from other servers. The power budget of each server is calculated by evenly dividing the total budget of the cluster by the number of servers in the cluster, because it is impossible to predict which server would have more workload and thus need more power at runtime. We use SISO as our baseline to show that a controller designed for a single server *cannot* be easily extended to control a server cluster where multiple servers are coupled together due to the common power budget. A fundamental advantage of our MPC controller is that MPC explicitly incorporates the interprocessor coupling in a cluster into its MIMO model and controller design, so that power can be shifted among servers to improve overall system performance.

In our experiments, we first use the High Performance Computing Linpack Benchmark (HPL) (V1.0a) [41] which is the workload used for system identification. To demonstrate that our control algorithm can effectively control a system running a different workload, we then run the experiments using SPEC CPU2006 (V1.0). HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic. The problem size of HPL is configured to be $10,000 \times 10,000$ and the block size is set as 64 in all experiments unless otherwise noted. SPEC CPU2006 is configured with one user thread and recorded as

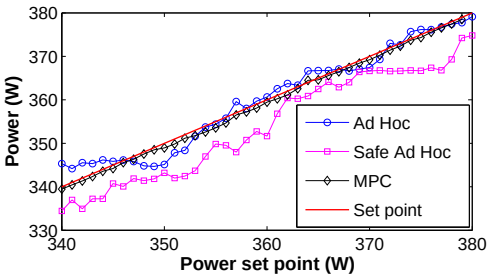


Figure 4.6: Comparison of steady state errors

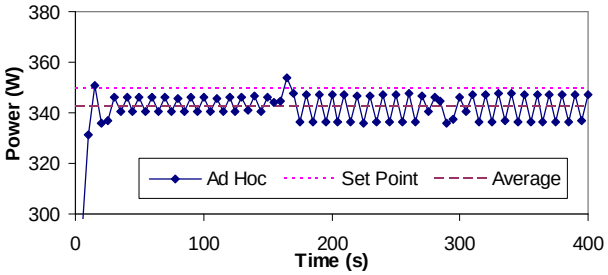


Figure 4.7: A typical run of Safe Ad Hoc

performance ratio, *i.e.*, the relative speed of the server to finish each benchmark (compared to a reference Sun UltraSparc II machine at 296MHz). CPU2006 is divided into CINT2006 and CFP2006 which consist of integer and floating-point benchmarks, respectively. The reported result is the average of all benchmarks in each category.

The MPC controller parameters used in all experiments include the prediction horizon as 8 and the control horizon as 2. The time constant T_{ref}/T_s used in (4.6) is set as 2 to avoid overshoot while having a relatively short settling time. Since the shortest period for our power meter to sample power is 1 second, the control period T for all controllers is set to 5 seconds to eliminate instantaneous reading errors by having an averaged value. While 5 seconds may seem to be a long time for a power controller to respond to power budget violation, our control algorithm can achieve much faster response time when it runs on high-end server clusters equipped with high-precision power monitor that can sample power in a much shorter period. In such a cluster where all servers share a common power supply, the control period should be derived to make sure that the settling time of the controller is shorter than the designed time interval that the power supply can sustain a power overload [50].

Comparison to Ad Hoc

Control Accuracy. In this experiment, we run the HPL benchmark on all the four servers. The power set point is 350 W. Since all servers have a CPU utilization of 100% when running HPL, Ad Hoc chooses servers to change frequency level in a round-robin fashion. Figure 4.4 shows that Ad Hoc starts with all processors throttled to the lowest frequency level. Since

the power is lower than the set point at the beginning of the run, Ad Hoc responds by stepping up the frequency level of one server at a time, until the power is higher than the set point at time 15s. Afterwards, Ad Hoc oscillates between two frequency levels once for each server in a round-robin way, because the set point power is between the two power consumption levels at two adjacent frequency levels for every server. As a result, the power consumption never settles to the set point and has a steady-state error of -3.8 W.

Figure 4.5 shows a typical run of our MPC controller. In contrast to Ad Hoc, MPC accurately achieves the desired power set point by having a floating-point (fractional) frequency level from the MPC controller, and then using the frequency modulator to generate a series of discrete frequency levels on a finer timescale to approximate the fractional level. One may think that Ad Hoc could be improved by also using a series of discrete levels to change the CPU frequency every 100ms. However, Ad Hoc would still have the same steady-state error because, without a fractional frequency level based on control theory, it can only oscillate between two frequency levels of each server. In addition, it is actually infeasible to run Ad Hoc every 100ms in practice, because Ad Hoc needs to measure the *actual* power consumption and then step up or down by one frequency level.

We acknowledge that MPC controller may have slightly higher actuation overhead than Ad Hoc by having a finer actuation timescale. However, as discussed in Section 4.1.4, the overhead is usually ignorable in most systems.

Figure 4.6 shows the result of running both MPC and Ad Hoc under a series of power set points from 340 W to 380 W. Each data point is the average of the steady-state power levels of three independent runs. The steady-state power level of each run is the averaged power level in the steady state of the controller, which is calculated by eliminating the transient power values at the beginning of the run. The MPC controller is able to meet the set point with a precision less than 1 W. However, Ad Hoc shows steady-state error that is often above the set point. For example, when set point is 340 W, Ad Hoc has the maximum positive steady-state error as 5.3 W above the set point.

Since Ad Hoc has steady state error, it is inappropriate to use Ad Hoc in a real system because a positive steady-state error (*i.e.*, average power is above the set point) may cause

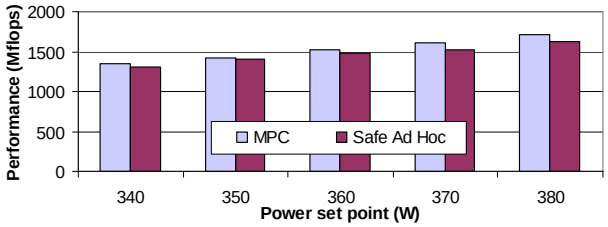


Figure 4.8: Application performance comparison

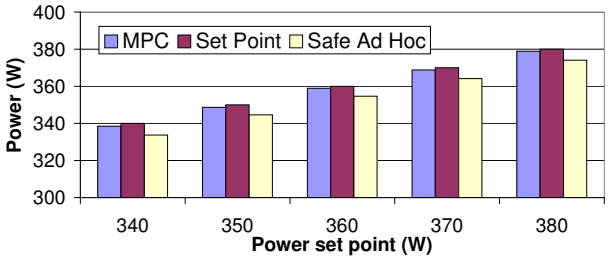


Figure 4.9: Power consumption using SPEC

the power supply to have overload and then likely failure. One may think that Ad Hoc could be easily modified to eliminate its positive steady-state error by having a safety margin. To do this, we can get the steady-state error of each single run of Ad Hoc. We then get the maximum positive steady-state errors of the three runs for each set point and then the maximum errors for all set points from 340 W to 380 W. By doing that, we get a safety margin of 5.545 W and we re-run the experiments for Ad Hoc with its power budget deducted this margin. This modified Ad Hoc policy is referred to as Safe Ad Hoc. We note that a similar baseline called *Improved Ad Hoc* has been used in [50]. Figure 4.7 shows Safe Ad Hoc runs at or below the set point most of the time. Note that at time 165s, Safe Ad Hoc still violates the power constraint once. This is because the safety margin is calculated based on the steady-state errors which are averaged values. Please note that Safe Ad Hoc is actually *infeasible* in practice because it is hard to have such *a priori* knowledge about the safety margin before spending a lot of time measuring this margin at runtime. However, we use Safe Ad Hoc as a baseline that can achieve the best possible performance in an ad hoc way and yet does not violate the power constraint.

Application Performance. In this subsection, we investigate the impact of cluster-level power control on the performance of the HPL benchmark. We use Safe Ad Hoc (with a safety margin of 5.545 W) instead of Ad Hoc because Ad Hoc would violate the power constraint and thus may not be suitable for a real system. Figure 4.8 plots the benchmark performance (aggregated value of all the four servers) of Safe Ad Hoc and MPC. MPC has better performance than Safe Ad Hoc for all five set points from 340 W to 380 W, with a maximum performance improvement of 6.1% at 370 W. This is because MPC can accurately achieve the set-point power while Safe Ad Hoc stays below the set point most of the time.

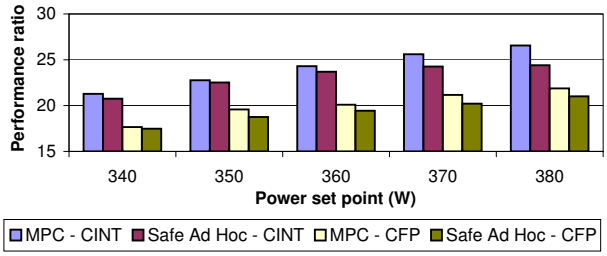


Figure 4.10: Application performance using SPEC

As a result, the performance of Safe Ad Hoc is worse than that of MPC. Please note again that it is actually *impossible* in practice for Safe Ad Hoc to have such a tight safety margin resulted from extensive experiments in this work. In a real system, Ad Hoc is commonly configured with a large safety margin and thus would result in much worse performance.

Results using SPEC CPU2006. To demonstrate the effectiveness of MPC with different workloads, we compare MPC with Safe Ad Hoc using SPEC CPU2006. All parameters for both MPC and Safe Ad Hoc remain the same. Figure 4.9 shows that the power consumption of the cluster under MPC is very close to the set point. The small gaps are caused by the short idle intervals between the runs of different benchmarks in CPU2006. In contrast, Safe Ad Hoc wastes the power budget because it uses the safety margin to ensure that power consumption always stays below the budget. As a result, MPC achieves better application performance (*i.e.*, the relative CPU speed compared to the reference machine used by SPEC) than Safe Ad Hoc for both CINT and CFP, as shown in Figure 4.10.

Comparison to SISO

In this set of experiments, we leave the first server idling and run the HPL benchmark only on the other three servers. The power set point is set to 360 W. Under the SISO control policy, each server evenly gets 90 W as its local power budget. Figure 4.12 shows that SISO

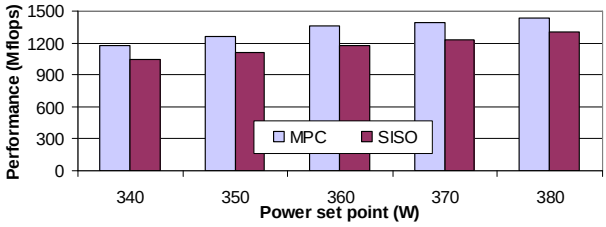


Figure 4.11: Application performance comparison with SISO

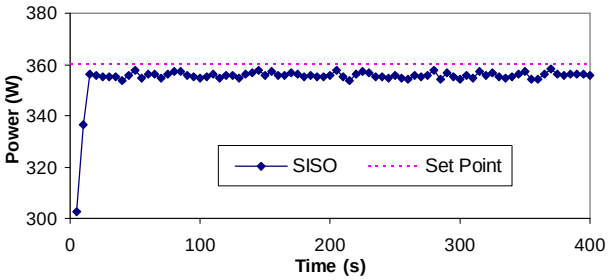


Figure 4.12: A typical run of SISO

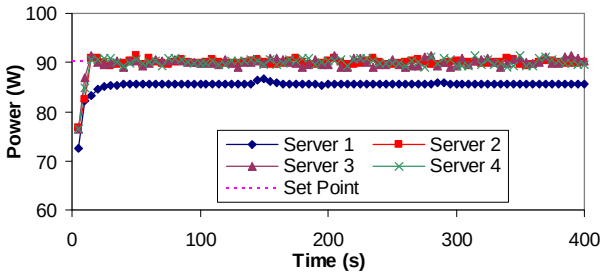


Figure 4.13: Power consumption of each server under SISO

has steady-state errors. This is because the idle server (Server 1) cannot use up its power budget of 90 W even when it is running at the highest CPU frequency level, as shown in Figure 4.13. On the other hand, the three busy servers (Server 2 to 4) cannot get enough power so that they can only run at degraded frequency levels.

Figure 4.11 shows the HPL performance data of MPC and SISO in this experiment. MPC has better performance because its total power consumption can exactly reach the desired power budget. In contrast, SISO policy has worse performance because some power budget is wasted by the idle server due to the even distribution of total budget. The maximum performance improvement of MPC is 15.3% at 360 W. Hence, it is clearly important for different servers in a cluster to share power resource because they commonly have non-uniform workloads. Even though it might be possible for the SISO control policy to have a cluster-level supervisor to dynamically adjust power budget among different SISO controllers at runtime, doing so in practice usually involves negotiation between the supervisor and the local controllers and thus cause undesired long response time. In addition, it may also cause each local controller to have constantly varying set point and thus may affect the stability of the whole cluster.

4.2 Data Center-level Power Management

In today's data centers that hold thousands of computing servers, power needs to be controlled at three levels: rack enclosure, Power Distribution Unit (PDU), and an entire data center, due to the physical and contractual power limits at each level [22]. However, to date, most existing work on server power control focuses exclusively on controlling the

power consumption of a single server. Only a few recently proposed control strategies are designed for the rack enclosure level [73, 72]. These centralized solutions cannot be easily extended to control an entire large-scale data center due to several reasons. First, the worst-case computational complexity of a centralized controller is commonly proportional to the system size and thus cannot scale well for large-scale systems [86]. Second, since every server in the data center may need to communicate with the centralized controller in every control period, the controller may become a communication bottleneck. Furthermore, a centralized controller may have long communication delays in large-scale systems. Therefore, highly scalable control solutions need to be developed.

There are several challenges in developing scalable power control algorithms. First, the global control problem (*i.e.*power control for an entire data center) needs to be decomposed into a set of control subproblems for scalability. The decomposition strategy must comply with the data centers' power distribution hierarchy. Second, the local controller designed for each decomposed subproblem needs to achieve local stability and control accuracy despite significantly varying workloads. Third, each local controller needs to coordinate with other controllers at different levels for global stability and control accuracy. Finally, the system performance of the data center needs to be optimized based on optimal control theory, subject to various system constraints.

In this section, we present a highly scalable hierarchical power control architecture for large-scale data centers composed of thousands of servers. Our control architecture is designed based on control theory for theoretically guaranteed control accuracy and system stability.

4.2.1 Hierarchical Power Control Architecture

In this subsection, we first introduce the power distribution hierarchy used in many data centers. We then introduce the design of our hierarchical power control architecture.

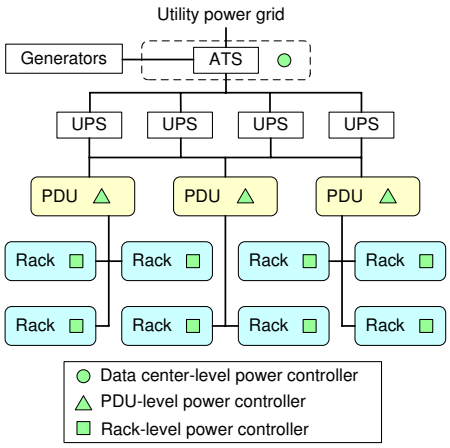


Figure 4.14: Simplified power distribution hierarchy in a typical Tier-2 data center.

Power Distribution Hierarchy

Today's data centers commonly have a three-level power distribution hierarchy to support hosted computer servers [22], though the exact distribution architecture may vary from site to site. Figure 4.14 shows a simplified illustration of the three-level hierarchy in a typical 1 MW data center. Power from the utility grid is fed to an Automatic Transfer Switch (ATS). The ATS connects to both the utility power grid and on-site power generators. From there, power is supplied to Uninterruptible Power Supplies (UPS) via multiple independent routes for fault tolerance. Each UPS supplies a series of Power Distribution Units (PDUs), which are rated on the order of 75 - 200 kW each. The PDUs further transform the voltage to support a group of server racks.

A typical data center may house ten or more PDUs. Each PDU can support approximately 20 to 60 racks while each rack can include about 10 to 80 computer servers. We assume that the power limit of the upper level (*e.g.*the data center) is lower than the sum of the maximum power limits of all the lower-level units (*e.g.*PDUs). This assumption is reasonable because many data centers are rapidly increasing their number of hosted servers to support new business in the short-term while infrastructure upgrades happen over much longer time scales.

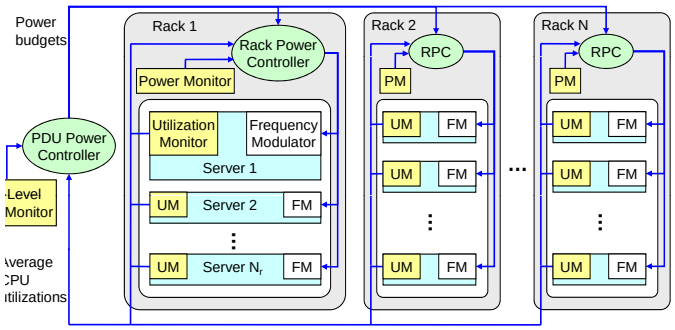


Figure 4.15: PDU-level and rack-level power control loops in the hierarchical power control architecture.

Control Architecture

Now we provide a high-level description of the our power control architecture, which features a three-level power control solution, as shown in Figure 4.14. First, the rack-level power controller adaptively manages the power consumption of a rack by manipulating the *CPU frequency* (e.g., via Dynamic Voltage and Frequency Scaling (DVFS)) of the processors of each server in the rack. Second, the PDU-level power controller manages the total power consumption of a PDU by manipulating the *power budget* of each rack in the PDU. Similar to the PDU-level controller, the data center-level controller manages the total power consumption of the entire data center by manipulating the *power budget* of each PDU. Our control architecture is directly applicable to data centers where applications (*e.g.*, scientific computing and background data processing) can allow degraded performance when power must be controlled to stay below a budget at runtime (*e.g.*, due to thermal emergency).

As shown in Figure 4.15, the key components in a rack-level control loop include a *power controller* and a *power monitor* at the rack level, as well as a *CPU utilization monitor* and a *CPU frequency modulator* on each server. The rack-level power controller is proposed in Section 4.1. The focus of this work is on the power control loops at the PDU and data center levels and the coordination among controllers at different levels.

The key components in a PDU-level power control loop include a *power controller* and a *power monitor* at the PDU level, as well as the rack-level power controllers and the utilization monitors of all the racks located within the PDU. The control loop is invoked periodically

to change the power budgets of the rack-level control loops of all the racks in the PDU. Therefore, to minimize the impact on the stability of a rack-level control loop, the control period of the PDU-level loop is selected to be longer than the settling time of the rack-level control loop. This guarantees that the rack-level control loop can always enter its steady state within one control period of the PDU-level loop, so that the two control loops are decoupled and can be designed independently. The following steps are invoked at the end of every control period of the PDU-level loop: 1) The PDU-level power controller receives the power consumption of the entire PDU in the last control period from the PDU-level power monitor. The power consumption is the *controlled variable* of this control loop. 2) The PDU-level controller also receives the average CPU utilization of each rack from the rack-level utilization monitor. The utilizations are used to optimize system performance by allocating higher power budgets to racks with higher utilizations. 3) The PDU-level controller then computes the power budget for each rack to have in the next control period based on control theory. The power budgets are the *manipulated variables* of the control loop. 4) The power budget of each rack is then sent to the rack-level power controller of that rack. Since the rack-level power controller is in its steady state at the end of each control period of the PDU-level controller, the desired power budget of each rack can be achieved by the rack-level controller by the end of the next control period of the PDU-level controller.

Similar to the PDU-level control loop, the data center-level power control loop controls the power consumption of the *entire* data center by manipulating the power budgets of the PDU-level power control loops of all the PDUs in the data center. The control period of the data center-level power control loop is selected in the same way to be longer than the settling time of each PDU-level control loop.

4.2.2 PDU-level Power Controller

In this section, we introduce the design and analysis of the PDU-level power controller. The data center-level controller is designed in the same way.

Problem Formulation

PDU-level power control can be formulated as a dynamic optimization problem. In this section, we analytically model the power consumption of a PDU. We first introduce the following notation. T_p is the control period. $pr_i(k)$ is the power consumption of Rack i in the k^{th} control period. $\Delta pr_i(k)$ is the power consumption change of Rack i, *i.e.*$\Delta pr_i(k) = pr_i(k+1) - pr_i(k)$. $br_i(k)$ is the power budget of Rack i in the k^{th} control period. $\Delta br_i(k)$ is the power budget change of Rack i, *i.e.*$\Delta br_i(k) = br_i(k+1) - br_i(k)$. $ur_i(k)$ is the average CPU utilization of all the servers in Rack i in the k^{th} control period. N is the total number of racks in the PDU. $pp(k)$ is the aggregated power consumption of the PDU. P_s is the power set point, *i.e.*the desired power constraint of the PDU.

Given a control error, $pp(k) - P_s$, the control goal at the k^{th} control point (*i.e.*time kT_p) is to dynamically choose a power budget change vector $\mathbf{\Delta br(k)} = [\Delta br_1(k) \dots \Delta br_N(k)]^T$ to minimize the difference between the power consumption of the PDU in the next control period and the desired power set point:

$$\min_{\{\Delta br_j(k) | 1 \le j \le N\}} (pp(k+1) - P_s)^2 \tag{4.7}$$

This optimization problem is subject to three constraints. First, the power budget of each rack should be within an allowed range, which is estimated based on the number of servers in that rack and the maximum and minimum possible power consumption of each server. This constraint is to prevent the controller from allocating a power budget that is infeasible for the rack-level power controller to achieve. Second, power differentiation can be enforced for two or more racks. For example, in some commercial data centers that host server racks for different clients, racks may have different priorities for power budget allocation. As power is directly related to application performance, the power budget allocated to one rack may be required to be n (*e.g.*1.2) times that allocated to another rack. This is referred to as *proportional power differentiation*. The differentiation is particularly important when the entire data center is experiencing temporary power budget reduction. In that case, with power differentiation, premium clients may have just slightly worse application performance

while ordinary clients may suffer significant performance degradation. Finally, the total power consumption should not be higher than the desired power constraint. The three constraints are modeled as:

$$P_{min,j} \leq \Delta br_j(k) + br_j(k) \leq P_{max,j} \ (1 \leq j \leq N)$$
$$\Delta br_i(k) + br_i(k) = n(\Delta br_j(k) + br_j(k)) \ (1 \leq i \neq j \leq N)$$
$$pp(k+1) \leq P_s$$

where $P_{min,j}$ and $P_{max,j}$ are the *estimated* minimum and maximum power consumption of a rack. The two values are estimated based on the number of servers in the rack and the estimated maximum and minimum power consumption of a server when it is running a nominal workload. The two values may be different in a real system due to different server configurations and workloads, which could cause the controller to allocate a power budget that is infeasible (*e.g.*too high or too low) for a rack-level controller to achieve. This uncertainty is modeled in the system model described in the next subsection. Therefore, PDU-level power management has been formulated as a constrained MIMO optimal control problem.

System Modeling

We now consider the total power consumption of a PDU. The total power consumption in the $(k+1)^{th}$ control period, $pp(k+1)$, is the result of the power consumption of the PDU in the previous control period, $pp(k)$, plus the sum of the power consumption changes of all the racks in the PDU.

$$pp(k+1) = pp(k) + \sum_{i=1}^{N} \Delta pr_i(k) \tag{4.8}$$

As introduced in Section 4.2.1, the control period of the PDU-level controller is longer than the settling time of the rack-level controller. As a result, at the end of each control period of the PDU-level controller, the desired power budget of each rack should have already been achieved by the corresponding rack-level controller, *i.e.*the power consumption change $\Delta pr_i(k)$ should be equal to the power budget change $\Delta br_i(k)$. However, there could be situations that a rack may fail to achieve a given power budget because it is infeasible to

do so. For example, a rack may fail to reach a given high power budget because its current workload is not as power-intensive as the nominal workload used to estimate the maximum power consumption of a rack used in constraint (4.8). As a result, the current workload may not be enough for the rack to achieve the given power budget even when all the servers in the rack are running at their highest frequencies. In that case, the power consumption change of the rack may become a function of the change of its assigned budget, *i.e.*$\Delta pr_i(k) = g_i \Delta br_i(k)$, where g_i is the system gain, which is also called the *power change ratio*. Note that g_i is used to model the uncertainties of the PDU-level power controller and its value is unknown at design time. Our model is not limited to constant g_i. When g_i varies along time, we can identify a range of g_i for which there exists a common Lyapunov function [26] for all g_is. As a result, our model can handle time varying g_i without any change. The literature has discussion for time-varying systems in more detail [86].

In general, the relationship between the power consumption of all the servers in a PDU and the power budget change of each rack in the PDU can be modeled as follows.

$$pp(k+1) = pp(k) + \mathbf{G}\boldsymbol{\Delta}\mathbf{br}(\mathbf{k}) \tag{4.9}$$

where $\mathbf{G} = \begin{bmatrix} g_1 & \dots & g_N \end{bmatrix}$, and $\boldsymbol{\Delta}\mathbf{br}(\mathbf{k}) = \begin{bmatrix} \Delta br_1(k) & \dots & \Delta br_N(k) \end{bmatrix}^T$.

4.2.3 Controller Design and Analysis

We apply *Model Predictive Control* (MPC) theory [64] to design the controller. Similar to the cluster-level controller, the MPC controller at the PDU level also 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 $\boldsymbol{\Delta}\mathbf{br}(\mathbf{k})$ that minimizes the following cost function under constraints.

$$\begin{aligned} V(k) = & \sum_{i=1}^{P} \|pp(k+i|k) - ref(k+i|k)\|^2_{Q(i)} \\ & + \sum_{i=0}^{M-1} \|\boldsymbol{\Delta}\mathbf{br}(\mathbf{k}+\mathbf{i}|\mathbf{k}) + \mathbf{br}(\mathbf{k}+\mathbf{i}|\mathbf{k}) - \mathbf{P_{max}}\|^2_{\mathbf{R(i)}} \end{aligned} \tag{4.10}$$

The reference trajectory defines an ideal trajectory along which the total power $pp(k+i|k)$ should change from the current value $pp(k)$ to the set point P_s (*i.e.*power budget of the PDU). Our controller is designed to track the following exponential reference trajectory so that the closed-loop system behaves like a linear system.

$$ref(k+i|k) = P_s - e^{-\frac{T_p}{T_{ref}}i}(P_s - pp(k)) \tag{4.11}$$

By minimizing the tracking error, the closed-loop system will converge to the power set point P_s if the system is stable.

The control penalty term causes the controller to optimize system performance by minimizing the difference between the estimated maximum power consumptions, $\mathbf{P_{max}} = [P_{max,1} \dots P_{max,N}]^T$ and the new power budgets, $\mathbf{br(k+i+1|k)} = \mathbf{\Delta br(k+i|k)} + \mathbf{br(k+i|k)}$ along the control horizon. The control weight vector, $\mathbf{R(i)}$, can be tuned to represent preference among servers. For example, a higher weight may be assigned to a rack if it has heavier or more important workloads, so that the controller can give preference to increasing its power budget. As a result, the overall system performance can be optimized. In our experiments, we use the average CPU utilization of all the servers in each rack as an example weight to optimize system performance.

We have established a system model (4.9) for the PDU-level power consumption in Section 4.2.2. However, the model cannot be directly used by the controller because the system gains $\mathbf{G}$ are unknown at design time. In our controller design, we assume that $g_i = 1, (1 < i < N)$, *i.e.*all the racks can achieve their desired power budget changes in the next control period. Hence, our controller solves the constrained optimization based on the following *estimated* system model:

$$pp(k+1) = pp(k) + [1 \dots 1]\,\mathbf{\Delta br(k)}. \tag{4.12}$$

In a real system that has different server configurations or is running a different workload, the *actual* value of g_i may become different than 1. As a result, the closed-loop system may behave differently. A fundamental benefit of the control-theoretic approach is that it gives us theoretical confidence for system stability, even when the estimated system model (4.12) may change for different workloads. In MPC, we say that a system is *stable* if the total power

$pp(k)$ converges to the desired set point P_s, that is, $\lim_{k\to\infty} pp\,(k) = P_s$. We prove that a system controlled by the controller designed with the assumption $g_i = 1$ can remain stable as long as the actual system gain $0 < g_i < 14.8$. This range is established using stability analysis of the closed-loop system by considering the model variations.

This control problem is subject to the three constraints introduced in Section 4.2.2. The controller must minimize the cost function (4.10) under the three constraints. This constrained optimization problem can be transformed to a standard constrained least-squares problem. The transformation is similar to that in [61] and not shown due to space limitations. The controller can then use a standard least-squares solver to solve the optimization problem on-line. In our system, we implement the controller based on the `lsqlin` solver in Matlab. `lsqlin` uses an active set method similar to that described in [27]. The computational complexity of `lsqlin` is polynomial in the number of racks in the PDU and the control and prediction horizons. The overhead measurement of `lsqlin` can be found in [61].

4.2.4 Coordination with Rack-level Controller

As discussed in Section 4.2.1, to achieve global stability, the period of an upper-level (*e.g.*, PDU) control loop is preferred to be longer than the settling time of a lower-level (*e.g.*, rack) control loop. This guarantees that the lower-level loop can always enter its steady state within one control period of the upper-level control loop, so that the two control loops are decoupled and can be designed independently. As long as the two controllers are stable individually, the combined system is stable. Note that the configuration of settling time is a sufficient but *not* necessary condition for achieving global stability. In other words, global stability can be achieved in some cases even when the control period is shorter than the settling time of the lower-level control loop.

We now analyze the settling times of the PDU-level control loop and the rack-level control loop. The settling time analysis includes three general steps. First, we compute the feedback and feedforward matrices for the controller by solving the control input based on the system model (*e.g.*, (4.9)) of a specific system and the reference trajectory (*e.g.*, (4.11)). The analysis needs to consider the composite system consisting of the dynamics of the original

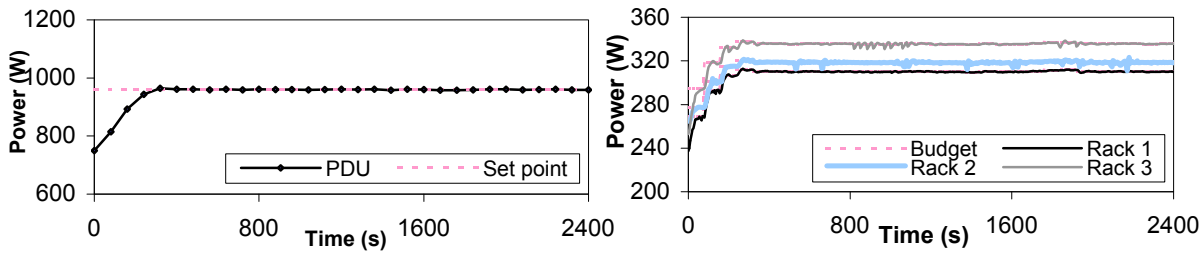
system and the controller. Second, we derive the closed-loop model of the composite system by substituting the control inputs derived in the first step into the actual system model. Finally, we calculate the dominant pole (*i.e.*the pole with the largest magnitude) of the closed-loop system. According to control theory, the dominant pole determines the system's transient response such as settling time.

As an example, we follow the above steps to analyze the settling times of the PDU-level controller and a rack-level controller used in our experiments. The PDU-level controller has a nominal gain vector $\mathbf{G} = [1, 1, 1]$. Our results show that the magnitude of the dominant pole of of the closed-loop system is 0.479. As a result, the number of control periods for the PDU-level loop to settle is 6. Similarly, the number of control periods for the rack-level loop to settle is 16.

Note that the selection of control periods is also related to the sampling period of the adopted power monitor. Since the shortest period for the power meters used in our testbed to sample power is 1 second, the control period of the rack-level control loops is set to 5 seconds to eliminate instantaneous reading errors by having averaged values. According to the settling time analysis, the control period of the PDU-level control loop is set to 80 seconds in our experiments. It is important to note that our control loops are not restricted to such long control periods. When equipped with high-precision power monitors that can sample power in a significantly shorter period (*e.g.*1 millisecond [50]), our control solution can quickly react to sudden power budget violations caused by unexpected demand spikes at different levels. In a real data center where such power monitors are equipped, the control periods should be derived to ensure that the settling times of the controllers are shorter than the manufacturer-specified time interval for the power supplies to sustain a power overload [50].

4.2.5 System Implementation

In this subsection, we first introduce the physical testbed and benchmark used in our experiments, as well as the implementation details of the control components. We then introduce our simulation environment.



(a) Power consumption of the PDU (b) Power consumptions of the three racks

Figure 4.16: A typical run of the hierarchical control solution on the physical testbed.

Testbed

Our testbed includes 9 Linux servers to run workloads and a Linux machine to run the controllers. The 9 servers are divided into 3 groups with 3 servers in each group. Each group emulates a rack while the whole testbed emulates a PDU. Server 1 to Server 4 are equipped with 2.4GHz AMD Athlon 64 3800+ processors and run openSUSE 11.0 with kernel 2.6.25. Server 5 to Server 8 are equipped with 2.2GHz AMD Athlon 64 X2 4200+ processors and run openSUSE 10.3 with kernel 2.6.22. Server 9 is equipped with 2.3GHz AMD Athlon 64 X2 4400+ processors and runs openSUSE 10.3. All the servers have 1GB RAM and 512KB L2 cache. Rack 1 includes Server 1 to Server 3. Rack 2 includes Server 4 to Server 6. Rack 3 includes Server 7 to Server 9. The controller machine is equipped with 3.00GHz Intel Xeon Processor 5160 and 8GB RAM, and runs openSUSE 10.3. All the machines are connected via an internal Ethernet switch.

In our experiments on the testbed, we run the High Performance Computing Linpack Benchmark (HPL) (V1.0a) on each server as our workload. HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic. We use HPL as our workload because it provides a standard way to quantify the performance improvement achieved by our control solution. We have tested our control architecture using other commercial benchmarks. The results are similar and can be found in the extended version [83].

Simulation Environment

To stress test the hierarchical control architecture in large-scale data centers, we have developed a C++ simulator that uses a trace file from real-world data centers to simulate

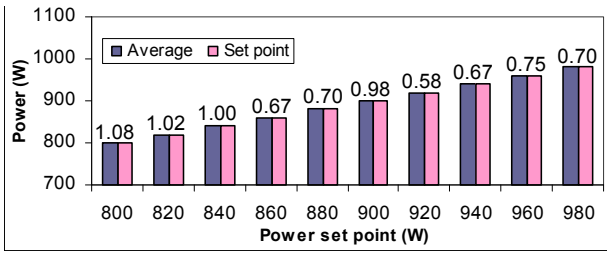


Figure 4.17: Average power consumption of the emulated PDU under different power set points (with standard deviations above the bars).

the CPU utilization variations. The trace file includes the utilization data of $5,415$ servers from ten large companies covering manufacturing, telecommunications, financial, and retail sectors. The trace file records the average CPU utilization of each server in every 15 minutes from 00:00 on July 14th (Monday) to 23:45 on July 20th (Sunday) in 2008. We generate several data center configurations. In each configuration, we group the servers into 6 to 8 PDUs with each PDU including 20 to 60 racks and each rack including 10 to 30 servers. Based on the specifications of the real servers used in our testbed, each server is randomly configured to have a minimum power consumption between 90W and 110W, a maximum power consumption between 150W and 170W, and a lowest relative frequency level between 0.3 and 0.5.

4.2.6 Experimentation

In this subsection, we first present our empirical results on the testbed. We then describe our simulation results in large-scale data centers based on the real trace file.

Empirical Results

Precise Power Control. In this experiment, we run the HPL benchmark on each of the 9 servers. The power set point of the PDU is 960W. Figure 4.16 shows a typical run of the hierarchical control solution. At the beginning of the run, the total power of the PDU is lower than the set point because all the servers are initially running at the lowest frequency levels. The PDU-level controller responds by giving more power budgets to all the three racks. The rack-level controllers then step up the servers' frequency levels to achieve the new power budgets within one control period of the PDU-level loop. After four control

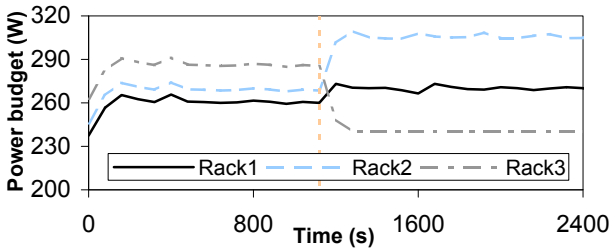


Figure 4.18: Power differentiation based on performance needs. Rack 3 has the lowest utilization.

periods, the power consumption of the PDU has been precisely controlled at the desired set point, without causing an undesired overshoot. After the transient state, as shown in Figure 4.16(b), the power budget allocated to each rack is kept at a stable value with only minor variations. The power consumption of each rack has also been precisely controlled at their respective allocated budgets. As discussed in Section 4.2.3, the PDU controller tries to minimize the difference between the estimated maximum power consumption ($i.e. P_{max,j}$) and the allocated power budget for each rack in its cost function. Specifically, the maximum power consumption for Racks 1 to 3 is 339W, 347.5W, and 364.5W, respectively. Since all the racks have the same weight (100% CPU utilization), their budgets are allocated to have the same distance with their maximum power consumptions.

In a data center, a PDU may be given different power set points at different times. For example, a data center may need to deploy a new PDU before an upgrade of its power distribution capacity can be done. As a result, the power set points of all other PDUs need to be reduced to accommodate the new PDU. Therefore, it is important to precisely control power for different power set points. We test our control solution for different set points (from 800W to 980W). Figure 4.17 plots the average power consumption of the emulated PDU with the standard deviation on the top of each bar. Each value is the average of 20 power measurements of the PDU after the PDU-level controller enters its steady state. The maximum standard deviation is only 1.08W around the desired set point. This experiment demonstrates that our solution can provide precise power control.

Power Differentiation. In data centers, it is commonly important for servers to have differentiated power budgets, especially when the available power resource is not enough for

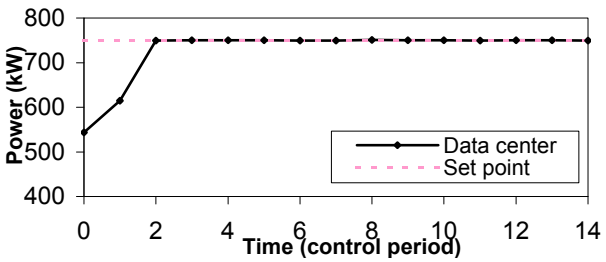


Figure 4.19: A typical run of the hierarchical solution in a simulated large-scale data center.

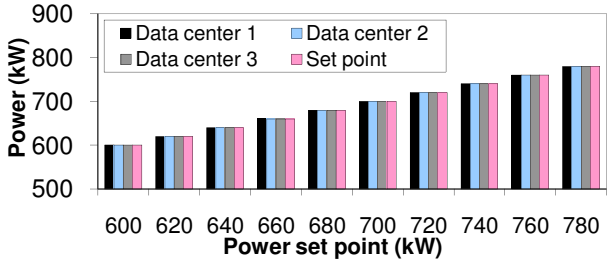
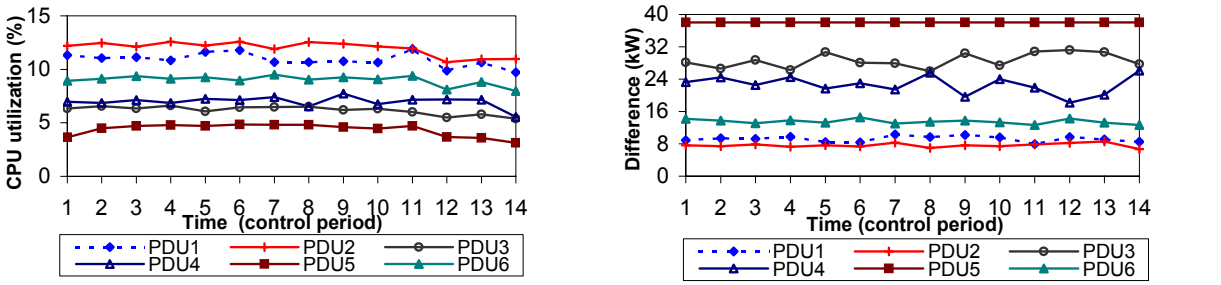


Figure 4.20: Average power consumptions under different data centers and power set points.

all the servers to run at their highest CPU frequency levels. For example, higher budgets can be given to servers running heavier workloads for improved overall system performance.

In this experiment, we differentiate server racks by giving higher weights (*i.e.*$\mathbf{R}(\mathbf{i})$) in the controller's cost function (4.10) to racks that have heavier workloads. Specifically, the weights are assigned proportionally to the racks' average CPU utilizations. Since running HPL on a server always leads to a 100% CPU utilization, we slightly modify the original HPL workload by inserting a sleep function at the end of each iteration in its computation loop, such that we can achieve different utilizations such as 80%, 50% for different servers. In the modified version of HPL, the problem size is configured to be $4,000 \times 4,000$ and the block size is set as 1. Note that the modified HPL benchmark is used *only* in this experiment. The power set point of the PDU is set to 810W. At the beginning of the run, we use the original HPL on all the servers so that all the racks have an average CPU utilization of 100%. As a result, all the racks are given the same weight. At time 1120s, we dynamically change the workload only on the servers in Racks 1 and 3 to run the modified HPL, so that the average CPU utilizations of Racks 1 and 3 become approximately 80% and 50%, respectively. Figure 4.18 shows that the controller responds to the workload variations by giving a higher power budget to a rack with a higher average CPU utilization. Rack 3 has the lowest budget because it has the lowest average utilization (*i.e.*50%). Note that application-level performance metrics such as response time and throughput can also be used to optimize power allocation in our solution. The results demonstrate that our solution can provide power differentiation for the consideration of overall system performance.



(a) Average CPU utilization (b) Difference between maximum power and budget

Figure 4.21: Power budget differentiation based on average CPU utilizations.

Simulation Results in Large-Scale Data Centers

In this section, we test our soluton in large-scale data centers using the trace file introduced in Section 4.2.5, which has the utilization data of 5,415 servers from real-world data centers.

Figure 4.19 is a typical run of our solution in a data center that is generated based on the method introduced in Section 4.1.4. This data center has 6 PDUs and 270 racks. The power set point of the data center is 750kW. As shown in Figure 4.19, the power of the data center precisely converges to the desired set point in two control periods of the data center-level control loop. Figure 4.20 plots the average power consumptions of three randomly generated data centers under a wide range of power set points from 600kW to 780kW. It is clear that our solution can achieve the desired set point for the three large-scale data centers. The maximum standard deviation of all the data centers under all the power set points is only 0.72kW.

We then examine the capability of our solution to differentiate PDUs based on the utilization data from the trace file. According to the controller design in Section 4.2.3, the data center-level controller tries to minimize the difference between the estimated maximum power consumption and the power budget for each PDU. Therefore, a PDU with a higher average CPU utilization should have a smaller difference because of its higher weight in the controller's cost function. Figure 4.21(a) shows the average CPU utilizations of the 6 PDUs in the experiment, while Figure 4.21(b) shows the difference (*i.e.*the estimated maximum power consumption minus the power budget) for each PDU. We can see that the difference order of the PDUs is consistent with the order of their average CPU utilizations. For example, PDU 2 has the highest average CPU utilization and thus the smallest difference. The results

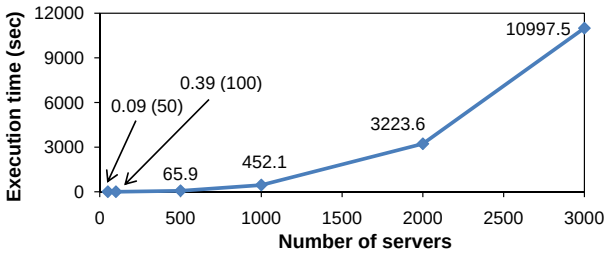


Figure 4.22: Average execution time of the MPC controller for different numbers of servers.

demonstrate that our solution can effectively achieve the desired control objectives in large-scale data centers.

The key advantage of the hierarchical control solution is that it decomposes the global control problem into a set of control subproblems at the three levels of the power distribution hierarchy in a data center. As a result, the overhead of each individual controller is bounded by the maximum number of units possibly controlled by a controller, which is 60 (racks in a PDU) in our simulations. Figure 4.22 shows that the average execution time of a centralized MPC controller (in 3 randomly generated data centers) increases dramatically when the number of directly controlled servers increases. Given that the control period of a data center-level power controller usually should be shorter than several minutes, a centralized MPC controller can only control 500 or fewer servers at the data center level. In addition, a centralized controller normally has undesired long communication delays in large-scale systems, resulting in degraded control performance. Therefore, centralized control solutions are not suitable for large-scale data centers.

4.3 Server-level Power Management

It is important do power control at the server level due to two reasons. First, power capping has been proposed at different levels in data centers that hold thousands of computing systems [72][22]. For example, a data center may allocate its power budget to different Power Distribution Units (PDU) and then, a group of racks. Further, the power budget of a single server may be allocated by the rack in which the server resides. Likewise, the server-level budget may be further divided among multiple components. Hence, it is important that

the power capping is enforced at all the levels. Second, when the power budget is tight, the power needs to be optimally allocated among different components at the server level so that maximized overall system performance can be achieved. For example, when CPU-intensive workload is running, the power should be shifted from the processor to the main memory by decreasing the processor frequency, to achieve improved system performance.

Power capping at the server level faces several major challenges. First, multiple components need to be manipulated simultaneously to control the power consumption of a server. The widespread adoption of multi-core processors and the rapid increase of applications' memory footprints have dramatically increased the demand on memory bandwidth and capacity. As a result, it is no longer valid to assume that the processor is the only major power consumer in a server. For example, a recent study [56] reported that the main memory systems in a multi-core server box, which has $32GB$ of Fully-Buffered DIMMs, may consume the same range of power as consumed by the processor. Hence, we need Multi-Input-Multi-Output (MIMO) strategies to coordinate both processor and memory for server power capping. Second, the components in a server are usually heterogeneous. Thus, we cannot simply allocate power proportional to the number of activities (*e.g.*, CPU utilization) since the power consumption contribution of a single activity varies for different components. For example, an instruction dispatched in a processor may contribute a different amount of power from a memory request. Therefore, we need to optimize the power allocation based on performance indicators. Third, workloads in different components in a server are usually synergetic. For example, processor frequency downscaling may decrease the number of memory requests so that the memory power consumption decreases accordingly. The coupling of the workload between components in a server is much stronger than that among servers in a cluster. Therefore, the synergy between components should be carefully addressed during power allocation at the server level. Fourth and most importantly, the workloads of different components are unpredictable at design time and may vary significantly at runtime. As a result, power management algorithms cannot rely on static power models or open-loop estimations. They must be self-adaptive to workload variations for optimal server performance.

In recent years, various power management solutions have been proposed for high-density servers. Most existing work focuses on minimizing the power consumption of a server within a specified performance degradation [53]. However, those solutions cannot provide any explicit guarantees for the server power to stay below a budget allowed by the server's power supply and cooling facilities. A recent study [51] relies only on the processor for server power capping, with the assumption that it is the only major power consumer in the server. As a result, their solution is limited to those small form-factor servers with small-size memory systems. Another recent paper [23] proposes to shift power between the processor and main memory proportionally to the number of activities. However, their solution does not explicitly measure power but relies on power estimation based on measured activities and off-line profiled power models, which may result in either power violations or performance degradation when the workload varies. In addition, their power actuation method does not take advantage of the availabilities of different power states in the processor and main memory to minimize idle power. Instead, they rely only on throttling the number of dispatched instructions and memory requests, which may lead to unnecessarily high idle power and a limited capacity of power adaptation.

In our work, we propose a novel server power capping solution that can precisely control the power consumption of a server to a desired budget. We periodically coordinate the processor and main memory to achieve improved performance, based on the memory queue level, by dynamically adjusting the voltage/frequency of the processor and placing memory ranks into different power states. Compared with the existing solutions, the newly proposed power management at the server level is expected to achieve better system performance, in terms of Instructions Per Cycle (IPC), while precisely capping the power consumption at the server level to the desired power budget.

4.3.1 System Design

In this section, we first introduce the power adaptation methods used in our work. We next provide a high-level description of the coordinated power control architecture that

Table 4.1: DRAM power states

Power state/Transition	Power/Delay
Active standby (ACT)	104.5 mW
Precharged (PRE)	76 mW
Active powerdown (APD)	19 mW
Precharged powerdown (PPD)	13.3 mW
Self refreshing (SR)	5 mW
PRE $\rightleftharpoons$ PPD	6 ns
APD $\rightleftharpoons$ ACT	6 ns
SR $\rightarrow$ PRE	$\geq$ 137.5 ns

periodically caps the server power and coordinates the processor and main memory for improved performance.

Power Adaptation for the Main Memory

To effectively enforce the power budget of a server, it is important to find efficient power actuators to dynamically change the power states of the processor and main memory. In this work, we conduct *Dynamic Voltage Frequency Scaling* (DVFS) in the processor. Our power adaptation method for the main memory leverages the fact that modern memory devices have multiple power states that retain stored data. Each power state consumes a different amount of power, whereas the transitions between states involve different performance overheads. In this work, we use the DDR memory architecture to describe our approach in the main memory, because DDR is the most common type of memory in today's server systems. As an example, Table 4.1 lists the power states, power consumption, and transition overheads of DDR2 SDRAM chips with 1Gb capacity [68]. A previous study [19] has shown that dynamically adjusting the power states of different memory devices can efficiently limit memory power consumption. By making a compromise between power saving and transition overhead based on Table 4.1, we utilize two power states: the precharged (PRE) state and precharged powerdown (PPD) state. The PRE state is an idle mode in which all banks are precharged and awaiting row activation commands to service a request, while the PPD state is the lowest power mode, other than the self-refreshing mode, in which all banks are precharged [44]. Hereinafter, we refer to the PRE state as the *active state* and the PPD

state as the *sleeping state*, for simplicity. We use the number of ranks in the active state as the actuator to dynamically change the memory power consumption. We place each memory rank either in the active state or in the sleeping state. The rank in the sleeping state needs to be activated first to service any arriving memory requests, which involves some overhead. We define the *active ratio* as the number of memory ranks in the active state normalized to the total number of ranks in the memory system. For example, if we keep 4 out of 16 ranks in the active state, the active ratio is 0.25.

The active ratio is kept constant within each control period while it is re-calculated at the end of each control period. The memory scheduler, shown in Figure 4.23, maintains a Most Recently Used (MRU) queue which records the most recently accessed rank at the head of the queue, and a status array which records the power state of each rank (*i.e.*, the active state or the sleep state). Its working process is based, generally, on an approach proposed by [19]. We briefly introduce it as follows.

- At the end of each control period, based on the difference between the newly calculated active ratio and the previous one, some sleeping ranks at the head of the MRU queue are transitioned to the active state (*i.e.*, the PRE state), or some active ranks from the tail of the MRU queue are transitioned to the sleeping state (*i.e.*, the PPD state).

- During a control period, when a memory request arrives, if the accessed rank is already in the active state, the request is sent to the transaction queue of the channel to which the accessed rank belongs. Otherwise, the accessed rank is transitioned to the active state and the memory access is held in the memory queue until the rank becomes active. At the same time, another rank in the active state needs to be switched to the sleeping state to keep the active ratio unchanged. To avoid frequent transitioning, a rank is chosen to be switched only if the time it has been in the active state is larger than the *break-even time* [19].

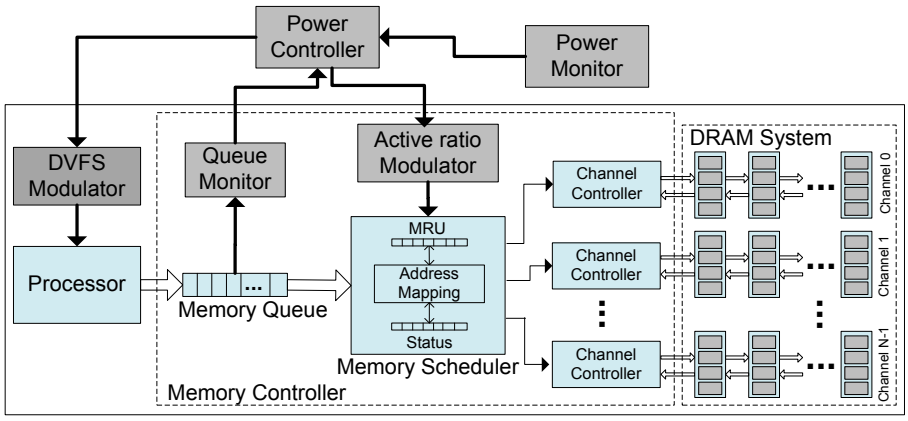


Figure 4.23: Coordinated Power Control Architecture

Coordinated Power Control Architecture

Figure 4.23 is a simplified illustration of a memory controller with multiple memory channels that can sustain multiple memory requests at each given time. The memory requests are generated by the processor and sent to a memory queue in the memory controller after a certain bus delay. The memory scheduler converts the physical address of the requests waiting in the memory queue into the memory address via an address mapping scheme. It then forwards them to the corresponding channels if the transaction queue in the channel has space. The channel scheduler reorders and schedules the requests in the transaction queue based on a scheduling algorithm (*e.g.*, Read or Instruction Fetch First). Finally, the scheduled requests are converted into a sequence of DRAM commands and serviced by the DRAM system.

As shown in Figure 4.23, the key components in the control loop include a power controller and a power monitor at the server level, a queue monitor and an active ratio modulator in the memory controller, and a DVFS modulator in the processor. The control loop is invoked periodically and its period is chosen based on a trade-off between actuation overhead and system settling time. The following feedback control loop is invoked at the end of each control period: 1) The power monitor (*e.g.*, a power meter) measures the average power consumption of the server in the last control period and sends the value to the power controller. The total power consumption is the *controlled variable* of the control loop. 2) The queue monitor samples the memory queue level (*i.e.*, the number of memory requests waiting in the queue)

in the memory controller multiple times within each control period. The average queue level is calculated at the end of each control period and sent to the power controller. The memory queue level is used as a power demand indicator for the power controller to coordinate the processor and main memory for improved performance. 3) The power controller collects the power value and queue level, calculates the new CPU frequency level for the processor and the new active ratio for the main memory, and then sends the values to the DVFS modulator and active ratio modulator, respectively. The CPU frequency level and active ratio are the *manipulated variables* of the control loop. 4) The CPU frequency modulator changes the CPU frequency level in the processor, and the active ratio modulator notifies the memory scheduler to power up/down memory ranks as introduced in Section 4.3.1, accordingly.

4.3.2 Coordinated Power Controller

In this section, we present the system modeling, design, and analysis of the coordinated power controller.

System Models

In order to have an effective controller design, it is important to model the dynamics of the controlled system, namely the relation between the controlled variable (*i.e.*, server power consumption) and the manipulated variables (*i.e.*, CPU frequency level and memory active ratio). Since the total power is the sum of the processor power and the main memory power [*], we have two power models: the processor power model and the memory power model. In this section, we discuss the two power models individually. We first introduce some notation. T_s is the control period of the power control loop. $f(k)$ is the average CPU frequency of the processor in the k^{th} control period. $r(k)$ is the active ratio of the main memory (*i.e.*, the number of memory ranks in the active state normalized to the total number of memory ranks). $\Delta f(k)$ is the difference between $f(k+1)$ and $f(k)$, *i.e.*, $\Delta f(k) = f(k+1) - f(k)$. $\Delta r(k)$ is the difference between $r(k+1)$ and $r(k)$, *i.e.*, $\Delta r(k) = r(k+1) - r(k)$. $F_{max} = 1$ and F_{min} are the maximum and minimum CPU frequency (normalized to the maximum)

[*]Since we do not change the power states of other components in a server, we assume that their power consumption can be approximated as a constant and is thus eliminated in the difference power model (4.17) introduced later.

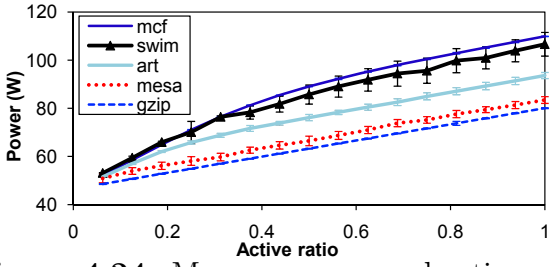


Figure 4.24: Memory power and active ratio

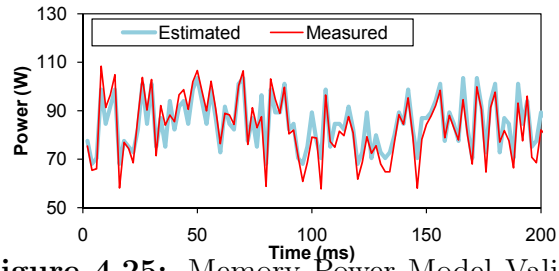


Figure 4.25: Memory Power Model Validation

while $R_{max} = 1$ and $R_{min} = 0$ are the maximum and minimum active ratio, respectively. $p_p(k)$, $p_m(k)$, and $p(k)$ are the power consumption of the processor, the main memory, and the whole server, respectively.

Memory Power Modeling. Based on the power adaptation method of the main memory introduced in Section 4.3.1, we need to model the dynamics of the relationship between the memory power consumption (*i.e.*, $p_m(k)$) and active ratio (*i.e.*, $r(k)$). However, a well-established physical equation is unavailable between $p_m(k)$ and $r(k)$. Therefore, we use a standard approach to this problem called *system identification*[26]. Instead of trying to build an analytical equation between $p_m(k)$ and $r(k)$, we infer their relationship by collecting data in the simulation environment introduced in Section 4.3.4 and establish a statistical model based on the measured data.

First, we observe the relationship between $p_m(k)$ and $r(k)$ from experiments with randomly selected workloads from SPEC CPU2000. Figure 4.24 plots the memory power consumption with the standard deviation under different active ratios for 5 workloads. Each data point in the curve is the average of 10 values which are sampled every 64 million CPU cycles (*i.e.*, the control period introduced in Section 4.3.4). The data points in the same curve are produced by forwarding the same number of instructions of the workload. As can be seen in Figure 4.24, the memory power consumption decreases when the active ratio decreases. The reduction of the memory power can be categorized into two parts: the reduction of idle power due to the decreased number of ranks in the active state and the reduction of activity power due to the reduced available memory bandwidth. As shown in the figure, the curves of CPU-intensive workloads (*e.g.*, *gzip* and *mesa*) are closer to a linear model than those of memory-intensive workloads (*e.g.*, *mcf*, *swim*, and *art*). This is because the decrease of the

active ratio for CPU-intensive workloads has a smaller impact on the bandwidth, compared with the memory-intensive workloads. Based on these experiments, we make three important observations: 1) There exhibits an approximately linear relationship between the memory power consumption and active ratio for each workload, except some offset. 2) The slope of all curves varies within a limited range, which is $[34.42, 53.68]$, based on our analysis. 3) The memory power consumption is stable with a fixed active ratio within a certain phase. The maximum standard deviation of all the data points in Figure 4.24 is only 6.63 W.

Using the system identification approach for each randomly selected workload, we find that a linear model fits very well for all of them (smallest $R^2 > 94\%$). Therefore, it is valid to assume that there exists a model between the memory power consumption and active ratio as follows:

$$p_m(k) = k_r r(k) + C_i, \tag{4.13}$$

where k_r is a generalized parameter that may vary for different workloads and C_i is a constant representing the offset. The dynamic model as a difference equation is

$$p_m(k+1) = p_m(k) + k_r \Delta r(k). \tag{4.14}$$

To validate our system model, we first stimulate the main memory system with pseudo-random white-noise input to change the active ratio in a random manner. We then compare the actual power consumption with the value predicted by our model. Figure 4.25 plots the predicted power and measured power by running *swim* which has the largest standard deviation. We can see that the predicted power is adequately close to the actual power of the memory system, even for the workload with a significant power variation.

Processor Power Modeling. Previous studies [72][51] have shown that the processor power is approximately linear to the DVFS level. In this work, we model the processor power in the same way as the memory power by using system identification. The power consumption of a processor is modeled as:

$$p_p(k) = p_p(k-1) + k_f \Delta f(k), \tag{4.15}$$

where k_f is a generalized parameter that may vary for different workloads running in the processor. The detailed analysis of the processor power model can be found in [51].

Server Power Modeling. The server power consumption is the sum of power of all components in the server:

$$p(k) = p_p(k) + p_m(k) + p_o, \tag{4.16}$$

where p_o is the power consumption of other components, which can be eliminated in a difference power model. Based on (4.14) and (4.15), we can get the difference power model as follows:

$$p(k+1) = p(k) + \mathbf{K}\mathbf{\Delta v}(k), \tag{4.17}$$

where $\mathbf{K} = \begin{bmatrix} k_f & k_r \end{bmatrix}$ and $\mathbf{\Delta v}(k) = \begin{bmatrix} \Delta f(k) & \Delta r(k) \end{bmatrix}^T$. The actual values of k_f and k_r in a real system may change for different workloads at different time and are unknown at the design time. However, in Section 4.3.3, we prove that a system, controlled by the controller designed with the estimated parameters, can remain stable as long as the variations of k_f and k_r are within allowed ranges. Note that since we do not manage the power of other server components, we assume that their power consumption can be approximated as a constant, which is eliminated in the difference model (4.17).

4.3.3 Controller Design

We apply *Model Predictive Control* (MPC) theory [64] to design the controller based on the system model (4.17). MPC is an advanced control technique that can deal with coupled MIMO control problems with constraints on the plant and the actuators. MPC enables us to combine power prediction, optimization, constraint satisfaction, and feedback control into a single algorithm. This property makes MPC well suited for coordinating the processor and main memory for server power capping.

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, which is referred to as the *prediction horizon*, based on the feedback $p(k)$ from the power monitor. The control objective is to select an input trajectory that minimizes the cost function while satisfying the constraints. An input trajectory includes the control inputs in the M control periods, which is referred to as *control horizon*. Once the input trajectory is computed, only the first element $\mathbf{\Delta v(k)}$ is applied as the control input to the system. The

input trajectory is continuously corrected based on feedback information in every control period.

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 $\mathbf{\Delta v(k)}$ that minimizes the following cost function under constraints.

$$V(k) = \sum_{i=1}^{P} \|p(k+i|k) - ref(k+i|k)\|^2_{Q(i)} + \sum_{i=0}^{M-1} \|\mathbf{\Delta v(k+i|k)} + \mathbf{v(k+i|k)} - \mathbf{V_{max}}\|^2_{\mathbf{R(i)}} \quad (4.18)$$

where $\mathbf{V_{max}} = \begin{bmatrix} F_{max} & R_{max} \end{bmatrix}^T$, $Q(i)$ is the *tracking error weight*, and $\mathbf{R(i)}$ is the *control penalty weight vector*. 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. The first term in the cost function represents the *tracking error*, *i.e.*, the difference between the total power $p(k+i|k)$ and a reference trajectory $ref(k+i|k)$. The reference trajectory defines an ideal trajectory along which the total power $p(k+i|k)$ should change from the current value $p(k)$ to the set point P_s (*i.e.*, power budget of the server)[64]. The second term in the cost function (4.18) represents the *control penalty*. The control penalty term causes the controller to optimize system performance by minimizing the difference between the highest power states of the processor and the main memory, $\mathbf{V_{max}}$, and the new power states, $\mathbf{v(k+i+1|k)} = \mathbf{\Delta v(k+i|k)} + \mathbf{v(k+i|k)}$, along the control horizon. The control weight vector, $\mathbf{R(i)} = \begin{bmatrix} R_c & R_m \end{bmatrix}_{1\times M}$, is tuned, based on the memory queue level, to shift power to either the processor (*i.e.*, R_c) or the main memory (*i.e.*, R_m), which will be discussed in Section 4.3.3.

This control problem is subject to two sets of constraints. First, both the DVFS level of the processor and the active ratio of the main memory should be within allowed physical ranges. Second, the total power consumption should not be higher than the desired power constraint. Therefore, the constraints are modeled as:

$$F_{min} \le f(k+1) \le F_{max}, R_{min} \le r(k+1) \le R_{max} \qquad (1 \le j \le N)$$

$$p(k) \le P_s$$

Based on the above analysis, the problem of server power capping has been modeled as a constrained MIMO optimal control problem. This constrained optimization problem can be easily transformed to a standard constrained least-squares problem [64]. Please note that

the computational complexity of an MPC controller can be significantly reduced by using a multi-parametric approach proposed in a recent study [77]. This approach can divide the problem into an offline part and an online part. At runtime, the controller only needs to solve the online part incrementally, which is a piecewise linear function. Therefore, the complexity MPC is small enough to be used for server-level power control in practice, especially when we have only one controlled variable and two manipulated variables..

Weight Allocation in Coordinated Controller

The power controller discussed above can precisely control the total power consumption of a server to the desired budget by solving an optimization problem. However, it cannot guarantee that the power states of the processor and main memory (*i.e.*, the CPU frequency and active ratio, respectively) are coordinated to achieve further improved performance. The experiments in Section 4.3.5 demonstrate that the system performance, in terms of Instructions Per Cycle (IPC), differs significantly by giving different preferences to the processor and main memory. In this subsection, we introduce a simple, but efficient, coordination scheme that allocates the control penalty weights for the processor and main memory based on the memory queue level, so that improved performance can be achieved. The reason why we use the memory queue level as the indicator is as follows: (1) the memory queue physically exists in many memory controllers [42], so it does not incur any other implementation overhead. (2) More importantly, when compared with other related metrics such as CPU stall time or L2 cache miss ratio, the queue level is a fair indicator for all workloads and can quantitatively reflect the power demand of the memory system. For example, if the queue level increases, it means that the memory requires more power to increase its capability so that the waiting time of the memory requests in the queue can be decreased.

We propose an algorithm called *Moving Average* (MA) to assign weights in the coordinated controller as follows. We keep the weight of the processor (*i.e.*, R_c) constant at 1 and adjust the weight of the memory (*i.e.*, R_m) at runtime. At the end of each control period, the controller calculates the moving average of the queue level $q_a(k)$ in a window with a size of L, after receiving the queue level $q(k)$ from the queue level monitor. Specifically,

$q_a(k) = \sum_{i=0}^{L-1} q(k-i)$. Clearly, the selection of the window size is a trade-off between a smooth weight allocation for the main memory and the response speed to memory workload variations. Based on our experiments, we found that the window size of 4 works well for most workloads. If the moving average of queue level $q_a(k)$ is higher than the reference level Q_{ref}, we set the weight of the memory as $(q_a(k) - Q_{ref})\alpha + 1$ to indicate that the memory needs more power. If $q_a(k)$ is smaller than Q_{ref}, the weight of the memory is set as $(q_a(k)/Q_{ref})^{\beta}$, which is lower than 1, to indicate that the processor needs more power. Both α and β are experiential parameters that map the queue level to the weight. Based on our profiling experiments with SPEC CPU 2000, we find that $\alpha = 10$ and $\beta = 2$ have approximately the best results.

The selection of the reference level Q_{ref} also plays an important role in the allocation scheme. Due to the dependencies among instructions, the length of the memory queue usually stops increasing when the processor stalls, instead of increasing infinitely. We define the *saturation level* as the maximum queue level that can be accumulated before the processor stalls. Clearly, the saturation level relies highly on the degree of dependency among the instructions. The stronger the dependency, the smaller the saturation level. If Q_{ref} is too high (*e.g.*, higher than the saturation level), the processor may stall before the queue level can be accumulated to the reference level. Likewise, if Q_{ref} is too low, the preference is more likely to be always given to the processor. Both of these may lead to degraded performance. Based on our profiling experiments, most workloads have a saturation level from 6 to 50. Without loss of generality, we set the reference level Q_{ref} as 2. Note that no reference level can be guaranteed to be always optimal in terms of performance for all workloads, since the interaction between the processor and memory is distributed into each instruction at runtime.

Control Analysis for Model Variations

A fundamental benefit of the control-theoretic approach is that it gives us confidence for system stability, even when the system power model (4.17) may change at runtime due to workload variations. We say that a system is *stable* if the total power $p(k)$ converges to the desired set point P_s, that is, $\lim_{k\to\infty} p(k) = P_s$.

Table 4.2: Simulator parameters

Parameters	Values
Processor	4 cores, 8 issues per core
Frequency scaling	3.2GHz at 1.3V, 2.8GHz at 1.15V, 1.6GHz at 0.95V, 0.8GHz at 0.8V
Functional unit	4 IntALU, 2 IntMult, 2 FPALU, 1 FPMult
L1 caches (per core)	64KB Inst/64KB Data, 2-way, 64B line size, 3-cycle hit latency
L2 cache (shared)	8MB, 8-way, 64B line size, 12-cycle hit latency
Memory	4 channels, 4 DIMMs/channel, 8 banks/DIMM, 1 rank/DIMM
Channel bandwidth	667MT/s, FB-DIMM DDR2

We now outline the general steps to analyze the stability of the system controlled by the coordinated power controller, when the actual system model is different from the *estimated* model used to design the coordinated controller. First, given a specific system, we derive the control inputs $\mathbf{\Delta v(k)}$ that minimize the cost function based on the estimated system model (4.17) with estimated parameters $\mathbf{K}$. Second, we construct the *actual* system model by assuming the actual parameter $k'_f = g_f k_f$ and $k'_r = g_r k_r$ for the processor and memory, respectively, where g_f and g_r represent the unknown system gain. Third, we derive the closed-loop system model by substituting the control inputs derived in the first step into the actual system model. Finally, we analyze the stability of the closed-loop system by computing the poles of the closed-loop system. According to control theory, if all the poles locate inside the unit circle in the complex space, the controlled system is stable.

Following the steps above, we have proven that the closed-loop system is guaranteed to be stable when $0 < g_f, g_r < 5.3$. This means that a system, controlled by the coordinated controller designed based on the estimated model (4.17), can remain stable as long as the real system parameters k'_f and k'_r are smaller than 5.3 times of the values used to design the controller. This stability analysis gives us confidence in the performance of our controller since we use the average k_f and k_r of all workloads from SPEC CPU2000 when we determine the nominal values of k_f and k_r at the control design time. It is reasonable to consider that our closed-loop system is stable for other workloads.

4.3.4 System Implementation

We integrate two cycle-accurate simulators: SimpleScalar [12] and DRAMsim [82], to simulate both the processor and main memory. SimpleScalar is heavily modified to simulate a quad-core CMP with one thread per core. To accurately simulate memory dependency, we modify the static main memory latency in SimpleScalar and hold all instructions which require main memory accesses from dispatching until the main memory accesses they depend on are returned by DRAMsim. DRAMsim is configured to simulate a FB-DIMM (Fully-Buffered DIMM) DDR2 SDRAM system with four channels and 32 GB capacity. The major parameters of the processor and main memory are shown in Table 4.2. Specifications of the main memory are based on the Micron data sheet [66]. We integrate Wattch [11] with SimpleScalar to estimate the power consumption of the processor. The power calculation in DRAMsim is based on the power model in [67]. We also assume that the simulated quad-core CMP has an idle power of 50 W when running at the lowest frequency, based on a recent quad-core Xeon processor from Intel. Hence, the peak power of the simulated system can be as high as 270 W. We assume that the processor and memory contribute the majority of the total power consumption in a server. However, our framework described above can be extended to include other components, such as network or disks.

The processor has 4 DVFS levels in our simulations, as shown in Table 4.2. Since the new frequency level periodically received from the coordinated controller could be any value that is not exactly one of the four supported DVFS levels, we implement the first-order delta-sigma modulator proposed in [51] to approximate the desired value via a series of supported DVFS levels. For example, to approximate $3GHz$ during a control period, the modulator would output the sequence, 2.8, 3.2, 2.8, and 3.2, on a smaller timescale. Apparently, the more subintervals the modulator is invoked within one control period, the better the approximation is, but with a higher overhead. In this work, we choose to use 20 subintervals to approximate the desired DVFS level. Based on a recent study [75], the DVFS overhead is approximately $10\mu s$. As a result, we choose a subinterval of $1ms$ so that the overhead is up to 1% in the worst case when the DVFS level is changed every subinterval. Therefore, the coordinated control period is $20ms$, which is 64 million CPU cycles in our simulation

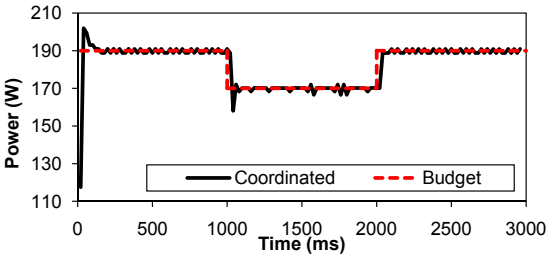


Figure 4.26: A typical run of the coordinated

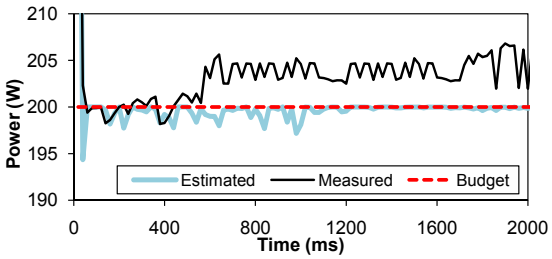


Figure 4.27: Estimation error of PLI

environment. To simulate the overhead of DVFS, we assume there is no instruction executed during transitions [43].

In our simulation environment, we implement the coordinated controller as a process separately from the processor-memory simulator. They communicate with each other via a pipe. In real systems, the controller can be implemented in the service processor firmware. The queue level is sampled every $1,000$ CPU cycles. The controller executes the control algorithm presented in Section 4.3.2 by calling a Matlab library which implements a standard constrained least-square solver. The parameters of the coordinated controller, used in all experiments, include the prediction horizon as 8 and the control horizon as 2.

Our experiments are driven by pre-compiled alpha binaries of SPEC CPU2000, among which we randomly pick 6 CPU-intensive ones and 7 memory-intensive ones. Since our solution can shift power between the processor and memory based on workload power demands, similar performance improvement can be achieved for other benchmarks. All the cores in the processor are configured to run a copy of the same benchmark since our focus is on the benefits of shifting power between the processor and the main memory, instead of workload scheduling on different cores. The performance is accumulated across all the cores in term of IPC.

4.3.5 Evaluation

In this section, we first introduce two state-of-the-art baselines. We then compare the coordinated solution with the two baselines, in terms of power capping performance and application performance. Finally, we investigate the impact of weight allocation on server performance.

Baselines

Our first baseline, referred to as *ProcOnly*, is a server power capping solution based on feedback control theory, proposed in a recent publication [51]. ProcOnly represents a typical server power capping solution that assumes the processor is the only major power consumer in a server. ProcOnly leverages frequency scaling in the processor to control the power consumption of the whole server to be within a certain power budget. We compare the proposed coordinated solution against ProcOnly to highlight that coordinating the processor and main memory, when power budget is limited, is important to achieve better performance than only considering the processor. The control scheme of ProcOnly is briefly introduced as follows. 1) A power monitor periodically measures the power consumption of a server and sends the value to the controller at the end of each period. 2) The controller calculates the desired CPU frequency level based on the measured power consumption and pre-defined power budget. 3) The processor scales the CPU frequency based on the calculated level accordingly. A fundamental difference between the coordinated solution and ProcOnly is that ProcOnly only manipulates the CPU frequency while disregarding the synergy between the processor and main memory. In contrast, the coordinated solution manipulates the power states of both the processor and memory, and adaptively adjusts the power states of the two components in a coordinated way.

The second baseline, referred to as *Proportional-by-Last-Interval* (PLI), is a recently proposed server power capping scheme [23] that shifts power between the processor and main memory based on the number of activities. PLI profiles the processor power model as a function of the number of dispatched instruction per cycle (DPC), while the memory power is modeled as a function of memory bandwidth. It periodically estimates the power consumption of the processor and main memory based on the two off-line power models. Given a power budget, in every period, PLI calculates the maximum number of activities in the processor (*i.e.*, dispatched instructions) and memory (*i.e.*, memory requests) that can occur in the next period without violating the power budget, as the thresholds, proportionally to the measured number of activities in the last period. The power budget is enforced by only running the calculated numbers of activities in the processor and main memory. There are

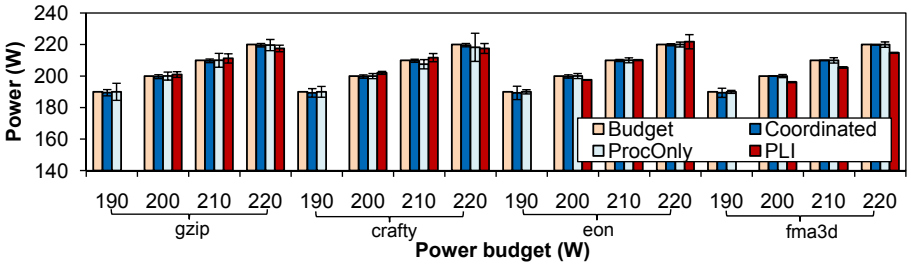
three fundamental differences between PLI and the coordinated solution. *First of all*, PLI is based on an estimation strategy that does not explicitly measure the power but relies on estimation, and thus cannot guarantee budget enforcement when the workload's power model becomes different from the profiled power model. Although the coordinated solution also predicts the power consumption based on profiled power models, the fundamental advantage of the coordination solution is that the prediction is continuously corrected based on feedback information, as discussed in Section 4.3.1. *Second*, the power allocation of PLI is proportional to the number of activities. In contrast, in the coordinated solution, the power allocation is driven by a performance indicator, the memory queue level. *Finally*, PLI enforces the power budget by directly throttling the number of activities (*i.e.*, DPC and bandwidth). In contrast, the coordinated solution exploits different power states to effectively reduce idle power for both the processor and memory. When the budget is tight, our solution can allow more system activities by utilizing the reduced idle power. As a result, our solution has a higher capacity of power adaptation and can achieve better application performance.

Power Capping Performance

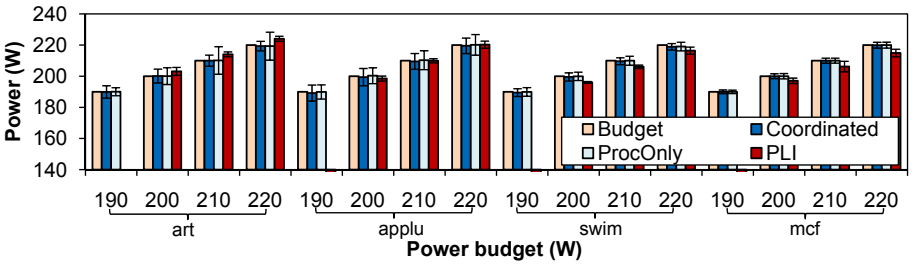
In this experiment, we compare the power capping performance of the coordinated solution with the two baselines, in terms of power control accuracy and power adaptation capacity.

Power Control Accuracy. To test the power control accuracy of the coordinated solution in a scenario where the power budget of the system needs to be changed at runtime due to various reasons (*e.g.*, thermal emergencies), we select a workload from SPEC benchmarks that is not used in the profiling of the memory power model in Section 4.3.2, *twolf*. As shown in Figure 4.26, the power budget is reduced from 190 W to 170 W at time 1,000ms, and then restored to 190 W at time 2,000ms. We can see that the coordinated solution quickly responds to the power budget reduction and precisely control the total power consumption of the server to the budget by adjusting the CPU frequency of the processor and the active ratio of the memory.

Figures 4.28(a) and (b) show the average power consumption under the proposed coordinated solution, ProcOnly, and PLI with standard deviations for CPU-intensive and memory-intensive workloads, respectively. To reduce the time of each experiment in the



(a) CPU-intensive workloads



(b) Memory-intensive workloads

Figure 4.28: Comparison of power capping among the coordinated solution, ProcOnly, and PLI

bar figures and thoroughly investigate the performance of our algorithm by running more experiments, we run all the experiments in the bar figures in a control period of 6.4M CPU cycles, which is 10 times shorter than the designed 64M control period. To have the same actuation overhead, we also scale down the overhead of DVFS from $10\mu s$ to $1\mu s$, accordingly. This is a reasonable approximation since the DVFS overhead can be reduced to nanoseconds in the near future, based on a recent study [47]. We also have verified that the variation of our memory power model is very small when we reduce the control period. Each bar in the figures is the average of 110 data points after the controllers enter the steady state. The reason why there are no results for PLI at the budget of $190W$ is because PLI is incapable of lowering the power below its idle power (*i.e.*, $191W$). We can see that the average power consumption under both the coordinated solution and ProcOnly precisely converges to the budgets. However, the standard deviation of ProcOnly tends to be greater than the coordinated solution. For example, the maximum standard deviation of ProcOnly in all runs is 8.94W compared to 4.3W for the coordinated solution. This is because the coordinated solution has two manipulated variables (*i.e.*, CPU frequency and active ratio) which generally can handle larger workload variations than ProcOnly, which

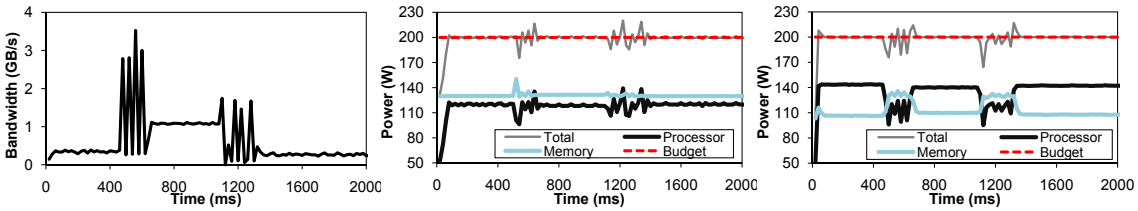
Table 4.3: Power adaptation capacity of the proposed coordinated solution and two baselines.

	Coordinated	ProcOnly	PLI
Lowest budget (W)	105	132	191

has only one manipulated variable (*i.e.*, CPU frequency). The result is consistent with the stability analysis of the coordinated solution, presented in Section 4.3.3, and the analysis of ProcOnly, presented in [51]. The analyses show that the stability range of the coordinated solution (*i.e.*, $(0, 5.3]$) is larger than that of ProcOnly (*i.e.*, $(0, 2)$). Therefore, the coordinated solution is less vulnerable to workload variations and more robust.

As for PLI, the average power consumption fails to converge to the power budget due to estimation errors in most runs. For example, the average power consumption under PLI is either smaller (*e.g.*, *fma3a* and *mcf*) or larger (*e.g.*, *art*) than the power budget. As a result, PLI either violates the power budget or leads to degraded performance. This is because PLI features a strategy that does not explicitly measure power but estimates power based on measured activities. As a result, it relies heavily on the accuracy of the off-line profiling of power models so that it has the risk of violating the power budget. To further verify our analysis of estimation inaccuracy about PLI, Figure 4.27 plots both the measured and estimated power consumption in a typical run of PLI by using *lucas* ($0.8B$ instructions are forwarded) under a power budget of 200 W. We can see that the actual power consumption (*i.e.*, measured) is successfully enforced to the power budget at the beginning because the power characteristic of activities in the processor and main memory are accurately captured by the power models used by PLI for estimation. After the time around 580ms, the power characteristic varies. However, PLI neglects the variation due to the lack of feedback information and still allocates power between the processor and memory based on the measured number of activities. This results in a constant power violation of approximately $4.5W$, which is highly undesirable and may cause system failures in real systems.

Power Adaptation Capacity. Power adaptation capacity is defined as the power budget range that a control solution can achieve. A higher power adaptation capacity



(a) Memory workload (b) ProcOnly (c) Coordinated solution

Figure 4.29: Comparison with ProcOnly during a typical run of gzip

means that a control solution can still manage to conduct power capping even when the power budget becomes very tight (*e.g.*, due to thermal emergency). PLI caps power by throttling the number of dispatched instructions in the processor and the number of memory requests in the main memory, instead of exploiting the low-power states. As a result, it has the smallest power adaptation capacity among the three solutions. As introduced in Section 4.3.5, a fundamental difference between the coordinated solution and ProcOnly is that the coordinated solution utilizes the low-power states of both the processor and main memory while ProcOnly only manipulates the DVFS level of the processor. Therefore, the coordinated solution has the largest power adaptation capacity. Table 4.3 shows the average minimum power budget that can be achieved by the three power capping algorithms using 8 workloads randomly selected from SPEC CPU2000. If the maximum power budget can be achieved is 270 W, we can see that the coordinated solution improves the power adaptation capacity up to 20% and 109%, compared with ProcOnly and PLI, respectively.

Application Performance

Now we compare the coordinated solution with the two baselines in terms of application performance.

ProcOnly. In this experiment, we run ProcOnly and the coordinated solution by using *gzip* under a power budget of 200 W. Figure 4.29(a) shows the memory bandwidth of running *gzip* in our simulation environment by forwarding $0.4B$ instructions. Figures 4.29(b) and (c) show the power consumption of the whole server, the processor, and the main memory in a typical run of ProcOnly and the coordinated solution, respectively. We can see tha *gzip* experiences time-varying memory workload from a bandwidth of $0.3GB/s$ to more than $3GB/s$ at runtime. However, ProcOnly only adjusts the power states of the processor,

disregarding the variations of memory workload. Consequently, the power consumption of the memory system is unnecessarily high even at the time when the memory system experiences a low workload. In contrast, the proposed coordinated solution adapts to the low memory workload by putting the main memory in low-power states and putting the processor in relatively high-power states. When the memory workload increases significantly, the coordinated solution shifts power from the processor to the main memory. As a result, performance increases by approximately 11% when compared with ProcOnly, due to the coordination between the processor and main memory. Please note that, though the power temporarily violates the budget due to severe workload variations in Figures 4.29(b) and (c), these overshoots are instantaneous and the system is safe as long as the server power can be controlled back to the desired budget within the designed time interval that the power supply can sustain a power overload.

To more thoroughly investigate the performance comparison between the coordinated solution and ProcOnly, we run experiments by using a variety of workloads under different power budgets. Figures 4.30(a) and (b) show the performance comparison for CPU-intensive and memory-intensive workloads, respectively. Performance is measured as the average IPC of 120 data points from the beginning of each run. As we can see, the coordinated solution has better performance for all CPU-intensive workloads (*i.e.*, 7.3% on average across all budgets). This is because the coordinated solution can coordinate the power states of the processor and main memory by dynamically shifting power between them. As for the memory-intensive workloads shown in Figure 4.30(b), the coordinated solution has similar performance to ProcOnly except for *applu*. This is because those memory-intensive workloads have high memory traffic at the majority of time. As a result, the coordinated solution places the memory in high-power states almost all the time which is similar to what ProcOnly does. The reason why the coordinated solution has a very slightly lower performance (around 1%) than ProcOnly for *applu* is because *applu*'s memory workload oscillates more than other memory-intensive workloads so that the queue level varies significantly at runtime. As a result, the coordinated solution has a higher overhead by frequently adjusting the power states of the processor and main memory. This set of experiments demonstrates that the

coordinated power control solution achieves considerably better application performance than ProcOnly for CPU-intensive workloads and similar performance for memory-intensive workloads.

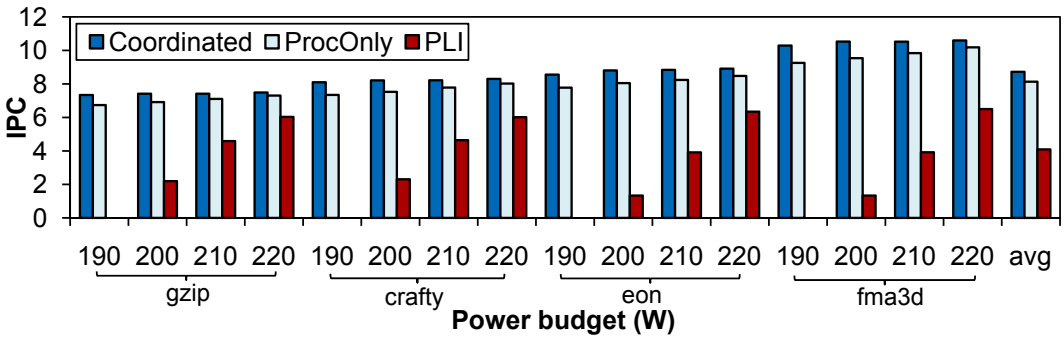
It is important to note that the *runtime* complexity of the proposed coordinated solution is comparable to that of ProcOnly, as discussed in Section 4.3.2. Please also note that the performance improvement is highly dependent on the percentage of memory power in the total power consumption of the system. The higher the percentage, the higher the improvement will be. As presented in [49], memory power may have a much higher percentage in many high-end servers than our configuration (*i.e.*, around 50%). Therefore, the performance improvement of the coordinated solution can be more significant.

PLI. The high idle power places PLI in a disadvantageous situation when the power budget is tight (*e.g.*, lower than the average power consumption). As shown in Figures 4.30(a) and (b), PLI has much lower performance than both the coordinated solution and ProcOnly. On average across all the budgets, the coordinated solution improves PLI by 110% and 120% for CPU-intensive and memory-intensive workloads, respectively. However, as the increases of power budgets, differences between PLI and the other two solutions becomes smaller.

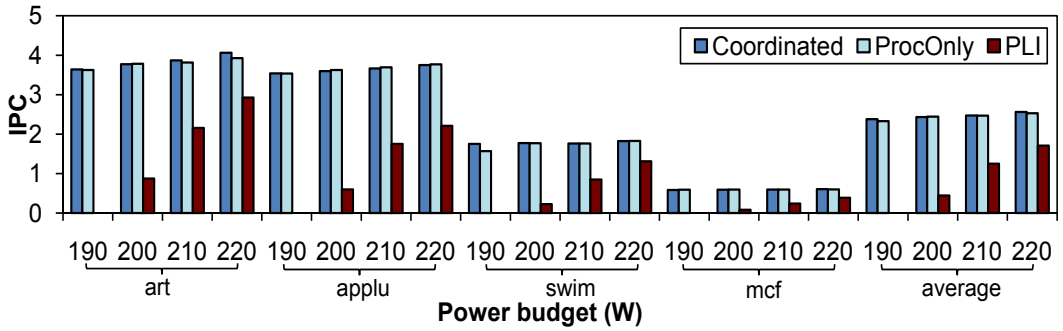
Weight Allocation Schemes

Now we investigate the control penalty weight $\mathbf{R}(i)$ in the cost function (4.18) and the impacts of different allocation schemes on the coordination between the processor and main memory. To highlight the importance of the dynamic weight allocation based on the memory queue level, we compare the proposed moving average scheme (MA) with three static allocation schemes. The first one, referred to as *Equal*, gives the same preference to the processor and main memory by assigning an equal weight to them. The other two, referred to as *Proc-preferred* and *Mem-preferred*, always give preference to the processor and main memory, respectively.

To stress-test the allocation schemes, we select 8 workloads and run all of them under a tight budget, 150 W, which is only approximately 56% of the system's peak power. We plot the average IPC of 200 control periods from the beginning in Figure 4.31. We can



(a) CPU-intensive workloads



(b) Memory-intensive workloads

Figure 4.30: Comparison of performance among the coordinated solution, ProcOnly, and PLI

observe that: 1) For CPU-intensive workloads, MA has better performance than both Equal and Mem-preferred. The reason is that using lower processor power states unnecessarily for CPU-intensive workloads may hurt performance significantly. At the same time, MA has similar performance to Proc-preferred. Note that Proc-preferred has slightly better performance than MA (around 2%) for *gcc*. This is because MA has some overhead to find the best allocation weight based on the memory queue level, which results in slightly worse performance. 2) For memory-intensive workloads, MA has significantly better performance than Proc-preferred. In addition, MA has slightly better performance than both Equal and Mem-preferred (with the average 2% and 6%, respectively). This is because shifting power to the main memory for memory-intensive workloads has less impact on performance than shifting power to the processor for CPU-intensive workloads, since the memory latency dominates the performance bottleneck.

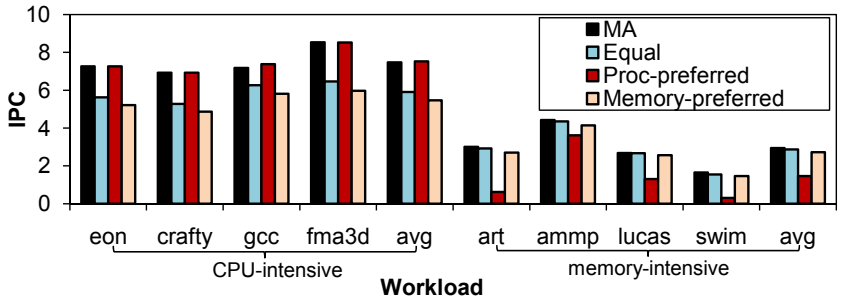


Figure 4.31: Comparison of weight allocation schemes

In general, MA, which dynamically allocates control penalty weights to the processor and main memory based on the memory queue level, is superior to other static allocation schemes.

4.3.6 Conclusions

Existing power capping solutions either rely solely on processor frequency scaling or shift power simply based on estimated system activities. Our solution shifts power between processor and main memory in a coordinated manner by dynamically adjusting the voltage/frequency of the processor and placing memory ranks into different power states, based on their power demands indicated by the memory queue level, to achieve improved server performance. Our coordinated control solution is systematically designed based on Model Predicative Control (MPC) theory for guaranteed control accuracy and system stability. We compare with two state-of-the-art server power capping solutions. One baseline relies only on processor frequency scaling while the other one uses estimated system activities for power shifting. Our experimental results demonstrate that our solution, on average, achieves 7.3% better performance than the first baseline for CPU-intensive benchmarks and doubles the performance of the second baseline when the power budget is tight. In addition, our solution significantly improves the power adaptation range and has better power control accuracy.

Chapter 5

Power-efficient Performance Management

Today's data centers face two critical challenges. First, various customers need to be assured of meeting their required service-level agreements such as response time and the throughput. Second, server power consumption must be controlled in order to avoid failures caused by power capacity overload or system overheating due to increasing high server density. However, the work we have discussed in the previous chapters address the two problems separately and cannot simultaneously provide explicit coordination between them. In this chapter, we propose power-efficient performance management for virtualized data centers.

5.1 Introduction

High power consumption in modern enterprise computing systems has led to high energy costs, expensive cooling hardware, floor space, and adverse impact on the environment. While power consumption must be minimized, another important goal of computing systems is to meet the service-level agreements (SLAs) required by customers, such as response time and throughput. SLAs are important to operators of computing systems because they are the key performance metrics for customer service and are part of customer commitments.

Therefore, it is important to guarantee the SLAs of the applications while minimizing the power consumption of the computing systems.

Recently, many computing systems have begun adopting server virtualization strategies for resource sharing. Virtualization technologies such as VMware and Xen can provide the required isolation layer to consolidate applications previously running on multiple physical servers onto a smaller number of physical servers, which is referred to as server consolidation. Server consolidation is based on the observation that many enterprise computing systems do not maximally utilize the available server resources all of the time. More importantly, live migration allows the movement of a virtual machine (VM) from one physical host to another with a negligible downtime [1]. This function makes it possible to use server consolidation as an on-line management approach, *i.e.*, having run-time estimation of resource requirements of every VM and dynamically re-map 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. Therefore, the coordinated performance and power management can be achieved by using virtualization to resize VM containers of applications and migrate VMs at runtime to consolidate the workload on an optimal set pf physical servers.

Coordinating performance and power management for distributed computing systems introduces several major challenges. First, applications in distributed computing systems often face significant, unpredictable workload variations. To guarantee SLAs, the coordinated management scheme must respond to a workload variation quickly by adjusting system resource allocation. Second, complex system software, such as web applications, sometimes has strongly correlated workload, which is due to either multi-tier installations of applications (*i.e.*, user interfaces, functional logic, and data accesses), or simultaneous peak hours of applications. To save maximized power, the VM placement controller should be correlation-aware so that positively correlated VMs should not be placed in the same physical server while negatively correlated VMs should be placed in the same server as much as possible. Third, the size of VM (*i.e.*, the resource demand) is stochastic. A good VM sizing scheme (*i.e.*, to estimate the resource demand of VMs) plays an important role in the

management solution. For example, the size of a VM is highly dependent on the workload of the server in which it is placed. A VM should have a larger size in a physical server whose workload is positively correlated with that of the VM than in a physical serve whose workload is negatively correlated with that of the VM. Therefore, instead of using a fixed VM size (*i.e.*, 95% percentile), a dynamic sizing scheme should be used to ahcieve more power saving with the same guarantee of performance. Finally, servers in a cluster may be manufactured by different hardware vendors and have different power efficiencies, *i.e.*, some servers are more power-efficient than others. A power management solution must be able to utilize this kind of heterogeneity to further reduce total power consumption.

In recent years, various VM placement schemes have been proposed to minimize the power consumption of computing systems while guaranteeing the performance requirements. However, most of them either use fixed sizing of each VM without any correlation consideration, or only considers positive correlations. As a result, more power saving with the same performance guarantee could be achieved. Our work features two novel properties. Firstly, the coordinated performance and power management is to propose a novel VM placement schemes that resizes each VM (*i.e.*, the resource demand of each VM dynamically based on the workload of the physical server in which it is placed, by exploiting both the positive and negative correlation between VMs. Secondly, our management features a two-level architecture. In the cluster level, we apply our correlation-aware placement algorithm with dynamic VM sizing. In the server level, we apply a utilization controller that controls the CPU utilization of each server to the desired CPU capacity, *i.e.*, 80% of the total computing capacity, by doing DVFS, so that power consumption can be further saved at the server level. Our newly proposed solution is expected to achieve more power saving while has the same performance guarantees, compared with the existing solutions.

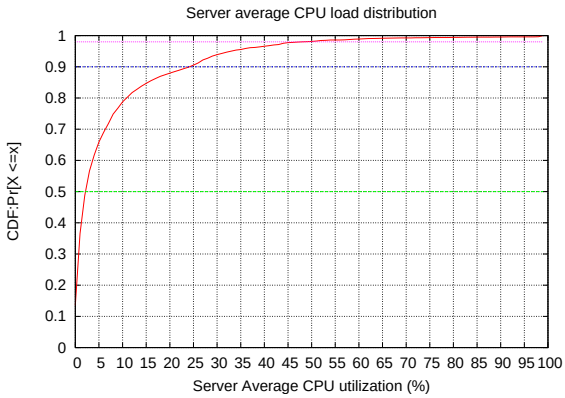


Figure 5.1: Cumulative Distribution Function of Server Average Load in one week

5.2 Data Center Workload Characterization

In this section, we present the study on a large data center workload trace. We call the objects in the trace *servers* before consolidation and *VMs* after consolidation throughout the chapter.

The trace includes the resource utilization data of $5,416$ servers from the IT systems of ten large companies covering manufacturing, telecommunications, financial, and retail sectors. The trace records the average CPU utilization of each server in every 15 minutes from $00:00$ on July 14th (Monday) to $23:45$ on July 20th (Sunday) in 2008. Among them, $2,525$ of the servers have the hardware spec information including the processor speed (in MHZ) and processor/core number. There is other server information including the OS, installed memory, number of disks, etc.

5.2.1 Individual Servers

Static Properties. We first look at the server average load distribution in the 1-week period. Figure 5.1 shows the Cumulative Distribution Function (CDF) of the server load averaged on the one week. Most of the servers have very low utilization most of the time. Among them, 50% of the servers have the avergae CPU load $\leq 3.12\%$; 90% of the servers have the average CPU load $\leq 25.2\%$; only 2% of the servers have load $\geq 50\%$.

Observation 1. *There is a potential of huge energy saving from consolidating the servers in the enterprise IT systems into a virtualized data center.*

Next, we examine the distributions of maximal (the 100-percentile), the 99-percentile, and the 95-percentile load of the servers in the week. Maximal load is used for conservative VM sizing (*i.e.*, over-provisioning). Recent work [80] pointed out that VM sizing over the x-percentile (e.g., $x = 99$ or 95) load could lead to substantially low resource provisioning cost with tolerable performance degradation. To show the relative demand increase atop the average load base, we normalize each server's maximal load with its average load during the week (*i.e.*, dividing the maximal load by the average load). The same normalization process is applied on the 99-percentile load and the 95-percentile load. Figure 5.2 shows the three CDF distributions. The maximal load is much larger than the average load; 90% of the servers have the maximal load at least 2.2 times larger than their average load; 50% of the servers have the maximal load at least 7.2 times larger than their average load. On the contrary, the 99-percentile load and the 95-percentile load are much closer to the average load. For example, 90% of the servers have the 95-percentile load at most 4.0 times larger than their average load; 50% of the servers have the 95-percentile load at most 1.6 times larger than their average load.

Observation 2. *Adopting a VM sizing approach like the X-percentile based could lead to another potential of large energy saving with the trade-off of minor performance degradation (e.g., 1% of performance degration probabiity when provisioning over the 99-percentile load).*

We examine the server load dynamics through the metric Coefficient of Variation (CV) $CV = \frac{\sigma}{\mu}$, where σ is the standard deviation, and μ is the mean of the server load distribution

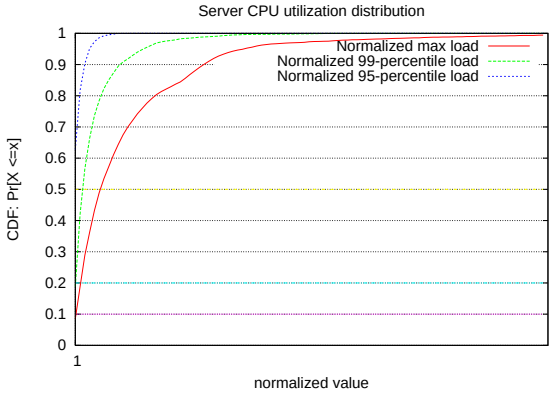


Figure 5.2: Cumulative Distribution Function of Server Normalized -percentile Loads

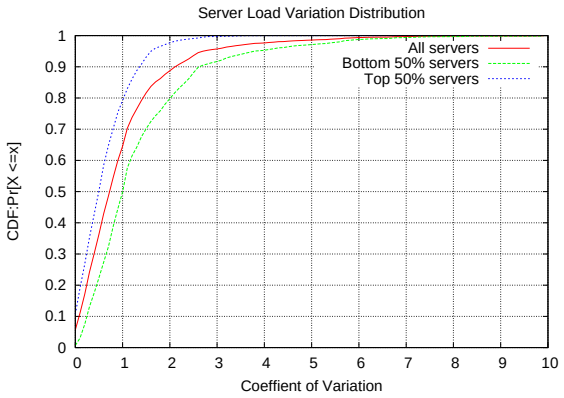


Figure 5.3: Cumulative Distribution Function of Server Load Dynamics

Server set	r_{xy}
All	-0.27
avg. load $\geq 5\%$	-0.23
avg. load $\geq 10\%$	-0.25

Table 5.1: Correlation between average load and load variation

in the week. In statistics, distributions with $CV > 1$ are considered having high-variance, distributions with $CV < 1$ are low-variance, and $CV = 1$ corresponds to an exponential distribution. Figure 5.3 shows the CDF distributions of the server's CV value, and contains three curves: one for all servers, another for the top 50% servers ranked by the average load, and the third for the bottom 50% servers ranked by the average load. Among all the servers, 64% have low-variance and 36% have high-variance. However, only 21% have high-variance in the top 50% servers (with large load), while more than 50% have high-variance in the bottom 50% servers (with small load).

We also examine the dependence between the average load and the load variation CV for the servers. The dependence metric, Pearson correlation coefficient r_{xy}, is defined on two attributes X, Y with the measurement values $\{x_1, x_2, \ldots, x_n\}$ and $\{y_1, y_2, \ldots, y_n\}$:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{(n-1)s_x s_y}$$

where $\bar{x}$ and $\bar{y}$ are the means, and s_x and s_y are the standard deviations of X and Y. In our case, X is the server average load, Y is the server load variation CV, and Table 5.1 show the calculated correlation between them for all the servers, servers with average load $\geq 5\%$, and

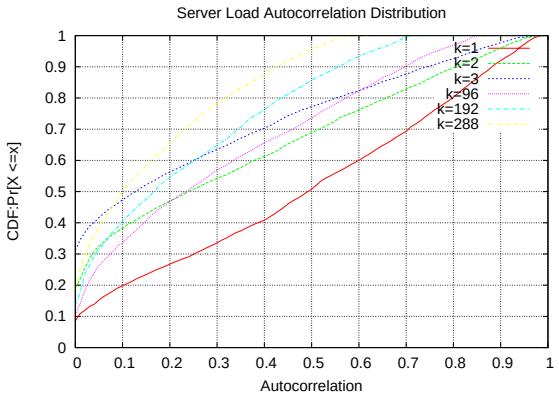


Figure 5.4: Cumulative Distribution Function of Server Load Autocorrelation

servers with average load $\geq 10\%$. Clearly, there is a consistent negative correlation between the average load and the load variation; it implies that the load variation is decreasing for the servers with larger average load.

Observation 3. *Consolidating small-load servers and large-load servers on separate physical machines is a simple but effective idea in our case: the VMs co-hosted will likely have similar load variation, which results in good statistical multiplexing effect.*

Temporal Properties To understand the load stationarity along the time, we examine the autocorrelation of the server load data. Autocorrelation is a measure of the similarity between observations as a function of the time separation between them. In our case, for a server X's CPU load data of length n $\{X_1, X_2, \ldots, X_n\}$, an estimate of the autocorrelation may be obtained as

$$\hat{R}(k) = \frac{1}{(n-k)\sigma^2} \sum_{t=1}^{n-k} [X_t - \mu][X_{t+k} - \mu]$$

for any positive integer $k < n$, where μ is the mean load and σ^2 is the load variance. The higher $\hat{R}(k)$ is, the more similar the loads at time t and $t-k$ are to each other, which means more accurate prediction on the load at time t can be made based on the observed load at the past time $t-k$.

Figure 5.4 shows the CDF of server load autocorrelation during the week. We examine the short-term load similarity with $k = (1, 2, 3)$ which represents the time separation of $(15, 30, 45)$ minutes, and the long-term load similarity with $k = (96, 192, 288)$ which

represents the time separation of $(1, 2, 3)$ days. Clearly, the servers have their load continuity in a very short time interval ($k = 1$), but the load similarity on a time horizon beyond that fades away quickly. $k = 1$ is still enough for local energy management tasks such as DVFS control; for data center server consolidation running at a minimal time interval of several hours or even days, a VM sizing approach has to take the load predictability problem into consideration.

Observation 4. *Energy management at small time scale may take reliable forecast with recent workload; energy management at large time scale has to take an adaptive approach for resource planning.*

5.2.2 Data Center-wide Workload

Physical resource demand low bound Instead of searching for the optimum bin packing solution which is computationally intractable with thousands of servers, we calculate *a low bound* that will be no more than the optimum solution. The idea is simple: we first calculate the sum of the CPU demand from all servers at each time point (like consolidating them into a single warehouse-sized machine); for a server consolidation scenario with a period of X hours, we divide the time into epochs with a length of X hours, and use the maximal aggregate CPU demand among all the time points in an epoch as the low bound for this epoch. We calculate the absolute CPU demand of those servers at a time point through multiplying the CPU utilization by the processor speed and the number of cores. To make the output more intuitive, we then normalize the absolute CPU demand to the minimum number of physical machines needed to meet it in the consolidation, where we assume a homogeneous virtualization platform with one 3GHz-Quadcore CPU in each machine. In practice, server consolidation has many other factors to consider such as memory and other resource constraints, VM affinity policies, and more; we focus on the CPU resource demand here due to the lack of other information in the trace, and note that the low bounds will still hold even with the additional consolidation constraints.

Figure 5.5 shows the derived low bounds for server consolidation on the $2,525$ servers. The *run-time low bound* curve shows the server demand low bound calculated at each time point

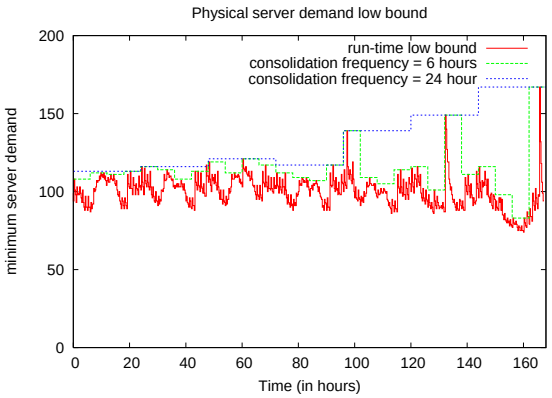


Figure 5.5: Physical Server demand low bounds at different consolidation frequency

(*i.e.*, every 15 minutes). There are clear periodic patterns with wide range which indicate the energy saving opportunity if server consolidation can be done in fast frequency (called dynamic consolidation in [80]). For example, the curve "*consolidation frequency = 6 hours*" shows the closeness of the consolidation low bound to the run-time resource demand. The latent problem with dynamic consolidation is the operational cost of VM re-location (e.g., VM migrations); the demand spikes at hour 99, 134, and 166 suggest the significant changes of both active machine set and VM-machine hosting relationship if the consolidation is too frequent. A semi-static consolidation like the curve "*consolidation frequency = 24 hours*" shows graceful changes of the consolidation low bound, even though the gap between it and the run-time resource demand is wide. While the above indicates the trade-off between energy efficiency and operational cost in server consolidation, it also leads to the following observation:

Observation 5. *A combination of server consolidation with a moderate frequency (e.g.,* 12 *hours) and local server energy management (*e.g.*, utilization control by DVFS) could drive the data center close to the energy-proportional computing model with acceptable operational cost.*

Server-pair Workload Correlation When consolidating the servers, it is important to consider the load correlation relationship for the VMs to be co-hosted. Intuitively, it is not desirable to consolidate two servers with strongly positive-correlated workload in the same

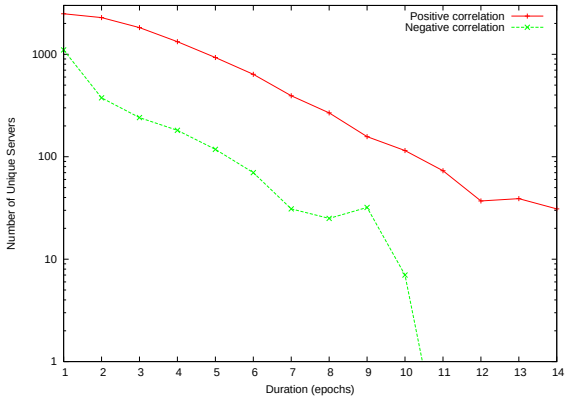


Figure 5.6: Number of unique servers with high correlation: consolidation frequency - 12 hours

physical server to avoid concurrent peak loads, while high statistical-multiplexing gain can be made to co-host two servers with strongly negative-correlated workload.

We examine the number of unique servers that belong to any strongly correlated server pair in the one-week trace file. We pick a correlation cutoff value 0.8 recommended in [80] for best-effort applications, and say both two servers are with *high correlation* if their load correlation coefficient has an absolute value no less than 0.8. Figures 5.6 shows the number of unique servers with high correlation during a number of epochs consistently; for example, there are 115 unique servers which have strongly positive correlation with other servers for up to 10 epochs (120 hours). Overall, most of the servers have strong correlation with another server for at least one consolidation epoch, while there are still a significant number of servers which have strong correlation with another server for a longer time duration (e.g., $>$ 72 hours). We also check the results under different consolidation frequencies like 24 hours and 48 hours, and have similar results; in addition, we observe that when the consolidation becomes more frequent, the servers with high correlation are more pervasive.

Observation 6. *Server consolidation has to take load correlation into consideration, especially in dynamic consolidations.*

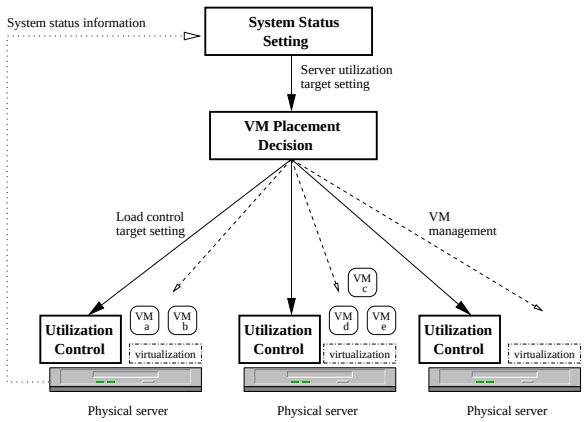


Figure 5.7: Coordinated Energy Management logic view

5.3 Coordinated Energy Management

Architecture

Figure 5.7 shows the architecture of the proposed COordinated Energy Management (COEM) solution. COEM has three components:

- *System Status Setting.* This component analyzes the run-time system information (such as server workload, application performance), and decides server utilization target settings desired in the system management process.

- *VM Placement Decision.*This component handles the server consolidation process. It takes VM workload, server inventory, and existing VM hosting information, and makes new VM placement decision with the consideration of VM hosting history information and server utilization targets. It then notifies available VM management tools for the VM-related execution (such as VM migration or rebooting), and local utilization control components at individual physical servers for energy-related management tasks (such as server on-off control or DVFS control target setting).

- *Power-aware utilization Control.* This component takes orders on local power management and executes them to meet requirements.

COEM manages the system in a closed loop. The actions from the VM placement decision and local utilization control components impact the system operational status, which serves

as feedback to the system setting component that adopts the right server utilization settings accordingly. In practice, the system setting functionality is done by the operators who set server utilization targets as management rules (e.g., the utilization range setting in VMware Dynamic Power Management tool [81]). Designing an automatic/semi-auto solution for this component is a real need and an open problem in the industry.

We focus on the designs of VM placement decision and local utilization control on DVFS. The two components are coordinated in COEM through two ways: at run-time, they are coordinated through the server utilization target setting; in performance optimization, they both benefit from load variance reduction through the introduction of *effective sizing*, which is described as the following.

5.3.1 Effective Sizing

Problem Formulation. We introduce *effective sizing* in the context of the stochastic bin packing problem. The original problem is defined as the following [29]: given a set of items, whose size is described by independent random variables $S = \{X_1, X_2, \ldots, X_n\}$, and an overflow probability p, partition the set S into the smallest number of set (bins) $S_1, \ldots, S_k$ such that

$$Pr[\sum_{i:X_i \in S_j} X_i > 1] \leq p \tag{5.1}$$

for all $1 \leq j \leq k$.

This is a natural analog of the deterministic bin packing problem, and is NP-complete. Actually, for some random variables (such as *Bernoulli trials*) even computing

$$Pr[\sum_{i:X_i \in S_j} X_i > 1]$$

is NP-complete [48]. Mapped back to the server consolidation application, a random variable represents the workload distribution of a VM, and each bin S_i represents a physical server. The constraint $Pr[\sum_{i:X_i \in S_j} X_i > 1] \leq p$ can be translated into a probabilistic Server Level Agreement (SLA): the probability that the aggregate resource demand exceeds the server

utilization target is at most p. The bin size 1 represents the server utilization consolidation target (100% by default), and can be other value (e.g., 90% or 200%). In the original stochastic bin packing problem, workload correlation is not considered.

Solution. *Effective sizing* follows the basic idea of *effective bandwidth* [39]: it associates each random variable with a fixed value, and then simplifies the stochastic problem into a deterministic version, which has many approximation algorithms with good performance. However, previous work focuses on random variables with specific distributions (e.g., weighted Bernoulli trials), and does not take load correlation into consideration. We introduce *effective sizing* to address the above issues with the following definition: let a random variable X_i represent a VM i's resource demand, and another random variable X_j represent a server j's existing aggregate resource demand from the VMs already allocated to it; The Effective Size ES_{ij} of i if hosted on server j consists of two parts:

- *Intrinsic demand*

$$ES_{ij}^{I} = \frac{C_j}{N_{ij}} \tag{5.2}$$

and N_{ij} is the maximal value of N satisfying the following constraint

$$Pr[\sum_{k=0}^{N-1} U_k > C_j] \leq p \tag{5.3}$$

where U_k are independent and identically distributed (i.i.d.) random variables with the same distribution as X_i, and C_j is the server utilization target of server j. Intuitively, N_{ij} is the maximal number of VMs that can be packed into server j without breaking the probabilistic SLA when all the VMs have the same workload pattern as VM i.

- *Correlation-aware demand*

$$ES_{ij}^{CA} = Z_\alpha(\sqrt{\sigma_i^2 + \sigma_j^2 + 2\rho_{xy}\sigma_i\sigma_j} - \sqrt{\sigma_i^2 + \sigma_j^2}) \tag{5.4}$$

where σ_i^2 and σ_j^2 are the variances of the random variables X_i and X_j; ρ_{xy} is the correlation between X_i and X_j; Z_α denotes the α-percentile for the unit normal

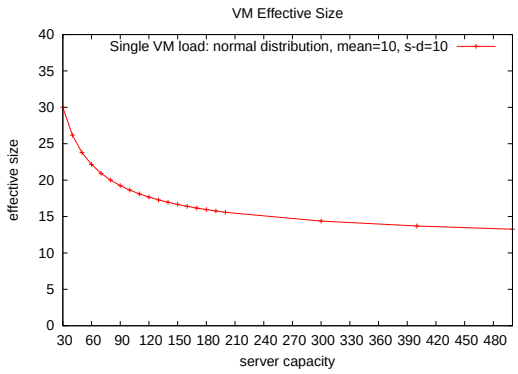


Figure 5.8: Effective sizing example: i.i.d random variables with normal distribution

distribution ($\alpha = 1 - p$). For example, if we want the overflow probability $p = 0.25\%$, then $\alpha = 99.75\%$, and $Z_\alpha = 3$.

- Finally, $ES_{ij} = ES^I_{ij} + ES^{CA}_{ij}$.

Clearly, the effective size of a VM is dependent not only on its own demand, but also on the demand of the co-hosted VMs along the time; this is a natural reflection of the statistical multiplexing effect in virtualzied data centers when multiple VMs are packed into one server. Figure 5.8 shows a VM's effective size as a function of the server capacity, for an example load model: normal distributions (a good approximation for the server load in practice due to the central limit theorem [20]). Assume the VMs to be packed have i.i.d. load with normal distribution ($\mu = 10, \sigma = 10$), and the overflow probability is 5% (the corresponding $Z_\alpha = 2$). When the server capacity is 30, the effective size of a VM is 30, which is actually the 95-percentile load of this VM. When the server capacity increases, the effective size of the VM decreases: the effective size is 20 (the 70-percentile load of this VM) when the server capacity is 80; the effective size is less than 14.4 after the server capacity is large than 300. Intuitively, effective sizing allocates less resource buffer for smaller VMs (relative to the hosting server's capacity) since the statistical-multiplexing gain increase with the number of random variables in a bin.

On intrinsic demand ES^I_{ij}, we note that the computation of Equation (5.3) is much simpler than that for the constraint (5.1) in the original problem. It is easy to show that Equation (5.3) can be solved in polynomial time even for Bernoulli trials. Lastly, we would

Data Structure:

- VM list with the load distributions.
- physical server list including the server utilization target and the overflow probability.

Algorithm

assume a cluster with homogeneous servers.

1. Calculate the intrinsic load ES_i^I of the VMs.
2. Sort the VMs in decreasing order by ES_i^I.
3. Place each VM i in order to the best non-empty server in the list which has sufficient remaining capacity and yields the minimal correlation-aware demand ES_{ij}^{CA} for i. If no such server is available, pick the next empty server in the server list and repeat.

Figure 5.9: COEM Correlation Aware (CA) VM Placement Algorithm

like to point out that in a homogeneous server cluster with the same utilization target setting, the intrinsic demand ES_{ij}^I can have the subscript j dropped and be computed only once for the whole cluster.

On correlation-aware demand ES_{ij}^{CA}, we note that it is a heuristic to approximately estimate the demand variation affected by load correlation. Equation (5.4) essentially approximates X_i and X_j with two correlated random variables with normal distributions, and compensates the error in ES_{ij}^I due to the ignorance of load correlation. Lastly, we would like to point out that the computation complexity for ES_{ij}^{CA} in a server consolidation process is $O(NMd)$ instead of $O(N^2d)$ in other correlation-aware placement schemes [80] (N: number of VMs; M: number of servers; d: data points of workload time series); in a virtualized data center, M is typically an order less than N.

5.3.2 VM Placement Decision

In this subsection, we present three VM placement algorithms based on effective sizing which have their pros and cons in terms of algorithm simplicity, consolidation efficiency, and migration cost. The solution choice in a consolidation scenario depends on the trade-off made among the three factors.

Data Structure:

- VM list with the load distributions.
- physical server list including server load information, the server utilization target and the overflow probability.
- existing VM hosting information $(VM_i, Server_j)$.

Algorithm

1. Sort physical servers in decreasing order by the server load.
2. For each overloaded server j with violated SLAs (overflow probability $> p$)
   (a) Sort the VMs in the server in decreasing order by the correlation-aware demand ES_{ij}^{CA}.
   (b) Place each VM i in order to the best non-empty and non-overloaded server k in the list which has sufficient remaining capacity and yields the minimal correlation-aware demand ES_{ik}^{CA}. If no such server is available, pick the next empty server in the server list and repeat.
   (c) If j's load meets the overflow constraint after moving out VM i, terminates the searching process for this server and go to next overloaded server; otherwise, continue the searching for the remaining VMs.
3. For the remaining non-overloaded server, sort them in increasing order by the server load.
4. For each non-overloaded server in the order
   (a) Place each VM i in the server to the best non-empty and non-overloaded server k in the list which has sufficient remaining capacity and yields the minimal correlation-aware demand ES_{ik}^{CA}. If no such server is available, terminates the searching process for this server and go to next non-overloaded server.
   (b) When all the VMs in this server can find a target non-overloaded server to move out, label this server as an empty server and migrate out all the VMs.

Figure 5.10: COEM History and Correlation Aware (HCA) VM Placement Algorithm

VM Placement - Correlation Aware (COEM-CA) We present the second VM placement algorithm which drops the load independence assumption.

As shown in Figure 5.9, COEM-CA VM Placement Algorithm is a combination of effective sizing and a variant of the Best Fit Decreasing (BFD) strategy, which has been shown to have the same approximation performance as FFD for the deterministic bin packing problem.

COEM-CA VM placement algorithm is correlation-aware, but may cause high migration cost like the IW algorithm. It fits for semi-static server consolidation which takes a moderate frequency and also wants to response to workload variation in some degree.

In practice, it might be a problem to solve Equation (5.3) without a simple load model. One alternative is using normal distribution model for approximation and calculating (μ_i, σ_i) for each VM i based on its load. The closed-form solution for Equation (5.3) under normal distribution is:

$$N_{ij} = \frac{2Z_\alpha^2\sigma_i^2 + 4\mu_i C_j - 2Z_\alpha\sigma_i\sqrt{Z_\alpha^2\sigma_i^2 + 4\mu_i}}{4\mu_i^2} \tag{5.5}$$

For a heterogeneous cluster where a VM's effective size can not be pre-computed before step 3, we can modify Step 1 and 2 so that the placement order is based on the value $(\mu_i + Z_\alpha\sigma_i)$, and use the actual effective size in the target server for step 3. To incorporate the server power efficiency factor into consideration, we can sort the servers in decreasing order by the power efficiency metrics in Step 3, like that in [79]. To consider other resource constraints such as memory, we can add them in Step 3 when judging a server's availability. The above discussions apply to the next two COEM VM placement algorithms as well, and will not be repeated for brevity.

VM Placement - History and Correlation Aware (COEM-HCA) We present the third VM placement algorithm which drops the load independence assumption and considers the VM hosting history information.

As shown in Figure 5.10, COEM-HCA VM Placement Algorithm is a combination of effective sizing and a heuristic for history-aware bin packing. An extension to be migration-cost aware like the algorithm in [79] is straightforward. It is higher in computation

complexity and makes best effort for achieving trade-off between consolidation efficiency and migration cost. It fits for dynamic consolidation which takes a fast frequency and utilizes workload variation along the time for maximal energy efficiency.

5.3.3 Power-aware Utilization Control

Power-aware utilization control at the server level may cover many components including CPU, memory, disks, and network cards. We focus on CPU power control as it is still the major energy sinker in common data center applications.

Problem Formulation We first introduce some notations. T is the control period of the server-level utilization controller. R_{ref} is the set point for the desired utilization of each server. $u(k)$ is the measured CPU utilization in the k^{th} control period. (kT sec after the system starts). $e(k)$ is the control error, specifically, $e(k) = R_{ref} - u(k)$. $f(k)$ is the CPU frequency of the server normalized to the maximum frequency, which has the maximum value equal to 1.

During the VM placement, each server has a capacity threshold so that the server can be guaranteed to meet with the CPU resource demand of all the VMs it holds. After the cluster-level VM placement, the CPU utilization on each server may still have some space from the capacity threshold. It gives us opportunities to further save power by down-scaling the CPU frequency. The goal of the server-level utilization controller is to control the CPU utilization of each server to the desired set point (i.e., R_{ref}) by dynamically adjusting the CPU frequency (*i.e.*, DVFS). The set point, R_{ref}, is decided by the placement decision component which has the expected workload information, and is the same as the server utilization target during the consolidation. The utilization controller will affect the monitored CPU utilization of the server. Therefore, the monitored utilization is calibrated to the actual resource demand based on the current frequency level.

In the sever-level utilization control loop, the CPU utilization of the server is the *controlled variable* while the CPU frequency is the *manipulated variable*. This utilization controller may be applied to control the utilization of each core by per-core DVFS or the aggregate utilization by chip-wide DVFS, in multi-core CPU architecture.

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., $u(k)$) and the manipulated variables (i.e., $f(k)$). If the CPU utilization of the server at the $(k-1)^{th}$ control period is $u_t(k-1)$ when running at the highest CPU frequency and we down-scale the CPU frequency to $f(k-1)$, the CPU utilization at the k^{th} control period becomes

$$u(k) = \frac{u_t(k-1)}{f(k-1)}. \tag{5.6}$$

Since the utilization controller works in a control period (*i.e.*, tens of milliseconds) which is much shorter than the period in which VMs report their CPU resource demand (*i.e.*, tens of seconds), $u_t(k-1)$ is slow time-varying compared with the control period of utilization control loop. We can make an assumption that $u_t(k)$ can be taken as a constant u_t within a certain time of period. If we define $d(k) = \frac{1}{f(k)}$, we get the following system model:

$$u(k) = u_t d(k-1). \tag{5.7}$$

Controller Design Proportional-Integral (PI) control can provide robust control performance despite considerable modeling errors. Based on the system model (5.7), we design a PI controller as follows:

$$d(k) = d(k-1) + K_1 e(k) - K_1 K_2 e(k-1), \tag{5.8}$$

Following the standard Root-Locus method, we can choose our control parameters as $K_1 = 1/u_t$ and $K_2 = 0$ such that our closed-loop transfer function is:

$$G(z) = z^{-1}. \tag{5.9}$$

It is easy to prove that the controlled system is stable and has zero steady state errors when the system model (5.7) is accurate. The detailed proofs can be found in a standard control textbook and are skipped due to space limit. The desired CPU frequency in the k^{th} control period is:

$$f(k) = \frac{f(k-1)u_t}{u_t + f(k-1)(u(k) - (R_{ref} - u(k))}. \tag{5.10}$$

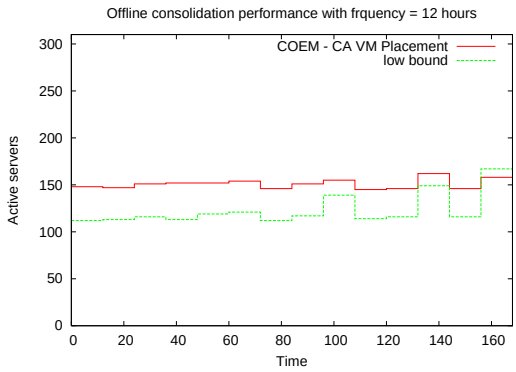


Figure 5.11: Off-line Consolidation of COEM VM Placement: no memory constraint

5.4 Evaluation

In this subsection, we present the COEM evaluation through both testbed and simulation results.

5.4.1 Simulation

Methodology. We implemented a stand-alone COEM simulator in Python. It runs on the workload trace described in Section 5.2.2 to evaluate the coordinated energy management performance when consolidating the servers into a virtualized data center. In the simulations, we use the $2,525$ servers with known CPU spec information, and calculate the absolute CPU demand of those servers at a time point through multiplying the CPU utilization by the processor speed and the number of cores. We assume that the physical machines in the virtualized data center are homogeneous with the following hardware specs:

- CPU: 3GHZ Quadra-core (the most common CPU model in the trace).
- Memory: we use the metric *memory constraint* which specifies the maximal number of VMs allowable on a physical server. The simulations evaluate the values $(8, 16, 32, \infty)$, where ∞ means no memory constraint.

For each run of the simulations, we define a consolidation frequency X (in hours), and divide the data trace along the time into epochs with a length of X hours. At each data sample point (every 15 minutes in the trace), we measure the consolidation performance with three metrics:

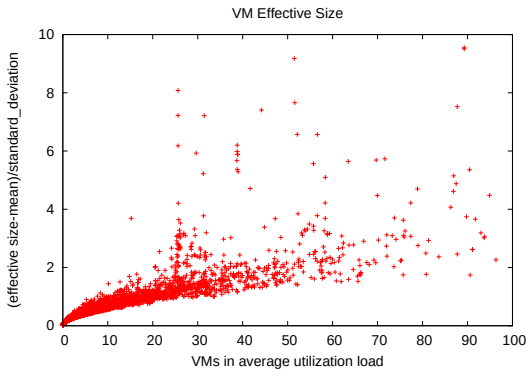


Figure 5.12: Effective sizing example: on real data center workload

- The number of active servers. An active server hosts at least one guest VM.
- The number of performance violations. One violation is defined as a server overloading event on a physical server where the sum of CPU demand from the hosted VMs is larger than 100% at a time point.
- Overflow probability. The overflow probability is the SLA metric, which equals the number of performance violations divided by the product of the number of active servers and the number of time points in each epoch.
- Migration cost. The migration cost is equal to the number of VM migrations in each epoch.

We compare our solutions (*i.e.*, COEM-CA and COEM-HCA) with four baselines (*i.e.*, B1-B4):

- Baseline scheme 1 (B1): B1 is a combination of VM sizing on average load and the First Fit Decreasing (FFD) placement scheme. This is an energy-biased scheme.
- Baseline scheme 2 (B2): B2 is a combination of VM sizing on maximal load and the FFD placement scheme. This is an over-provisioning scheme.
- Baseline scheme 3 (B3): B3 is a combination of VMware DPM's VM sizing scheme [81] on $(\mu + 2\sigma)$, where μ is the mean and σ is the standard deviation of a VM's load, and the FFD placement scheme. This is a representative scheme from state-of-art industry products.

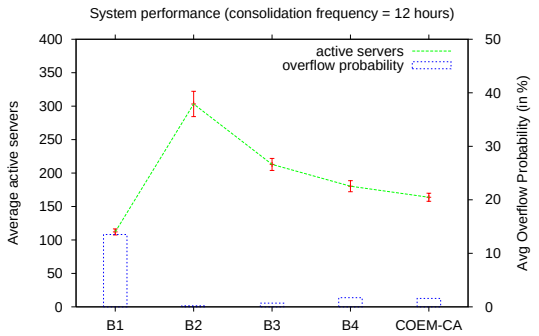


Figure 5.13: Online Consolidation of COEM VM Placement: no memory constraint

- Baseline scheme 4 (B4): B4 is a consolidation scheme in [80], which is a combination of VM sizing on the 95-percentile load, the FFD placement scheme, and the correlation-aware affinity rules (*i.e.*, two VMs will not be hosted in the same physical when their load correlation coefficient is higher than a threshold t; we use $t = 0.8$, recommended in [80] for best-effort applications). This is a representative scheme from the state-of-art research work.

- COEM-CA: In COEM CA placement scheme, we use $p = 0.05$ as the target overflow probability, the same as that used in B3 and B4. We will compare the above 4 baseline schemes with COEM-CA as all those schemes do not consider migration cost which leads to the fair comparison on VM sizing technologies.

- COEM-HCA: In COEM-HCA placement scheme, we also use the target overflow probability $p = 0.05$. We will compare COEM-CA and COEM-HCA with the focus on the migration cost efficiency.

Figure 5.11 shows the offline consolidation performance of COEM-CA VM placement compared to the resource demand low bound defined in Section 5.3.2. The consolidation frequency is 12 hours. COEM server consolidation performs close to the low bound; it uses 24% (with the standard deviation 11%) more physical servers on average than the low bound. Note that COEM-CA uses even less resource than the low bound in the last 12 hours; in this case COEM-CA incurs performance violations, while the low bound has the condition of no physical server overloading. Overall, COEM-CA server consolidation leads to 2% overflow (server overload) probability in average, with the standard deviation of 3%.

To take a close look at the impact of the effective sizing on server consolidation, Figure 5.12 shows a snapshot of the effective sizing results in one epoch during the above consolidation. The x axis is the VMs ranked by the average load; the y axis is the value of $\frac{\text{effective size}-\mu}{\sigma}$, which is the normalized over-provisioned resource on a VM atop its base average load. We see a clear trend of less over-provisioned resource for smaller VMs (in terms of average load), and more over-provisioned resource for larger VMs; the reason is explained in Section 5.3.1.

Online Consolidation. In online consolidation scenarios, a scheme is provoked at the end of each epoch and use the trace data in that epoch to make the consolidation decision for the next epoch. We assume that the workload in the next epoch will be the same as that in the previous epoch. The error of this simple workload prediction is compensated by lowering the server utilization target with a buffer factor b (e.g., $b = 80\%$ leads to lowering server utilization target to $C_i * 80\%$).

Figure 5.13 shows the online consolidation performance of the five schemes with a consolidation frequency of 12 hours, no memory constraint, and buffer factor $b = 80\%$. The performance includes both the number of active servers per epoch on average along with the standard deviation, and the performance violations per epoch on average. B1, which provisions the VMs on their average load, has the lowest active servers but very high performance violations. COEM-CA uses 46% fewer active servers than B2 with max-load based VM sizing, 23% fewer than B3 with VMware DPM's sizing approach, and 10% fewer than B4 with the 95-percentile sizing. COEM-CA also achieves its performance goal with below 5% overflow probability; it even has fewer performance violations than B4.

We run another simulation with a consolidation frequency of 12 hours, memory constraint of maximally 16 VMs per server, and buffer factor $b = 100\%$, which is shown in Figure 5.14. COEM-CA uses 34% fewer active servers than B2, 16% fewer than B3, and 11% fewer than B4. COEM still achieves its performance goal with below 5% overflow probability and has fewer performance violations than B4. Simulations with other parameter combinations were run, including consolidation frequency (4, 6, 12, 24 hours), memory constraint $(8, 16, 32, \infty)$, buffer factor (80%, 90%, 100%). The relative performance of the five schemes in those

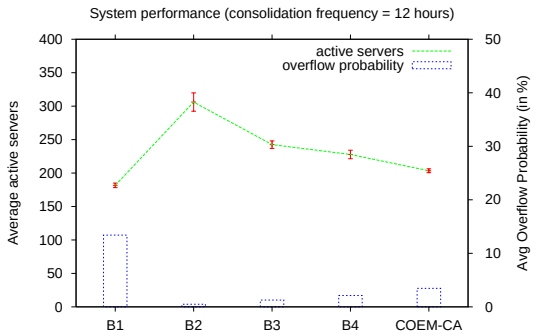


Figure 5.14: Online Consolidation of COEM-CA VM Placement: memory constraint = 16 VMs/server

	the number of active servers	overflow probability	migration cost
COEM-CA	163.4	1.41%	2271
COEM-HCA	141.6	3.10%	408.5

Table 5.2: Comparison between CA and HCA

simulations are consistent with the above results, and the report is skipped due to space limit.

Migration Cost Consideration. COEM-CA and COEM-HCA have two fundamental differences: 1) COEM-HCA considers the VM hosting history information while COEM-CA does not; 2) COEM-HCA takes exhaustive searching of the servers with all the VMs when having a VM to place, while COEM-CA takes the BFD scheme and a VM to be placed can only search the servers with VMs already placed so far. As a result, COEM-HCA has much higher computation overhead than COEM-CA, but expects to have more efficient placement results. To evaluate the performance of COEM-HCA, we run on-line consolidation of both COEM-CA and COEM-HCA under the same configuration as in Figure 5.13. Table 5.2 shows the comparison results in terms of the average number of active servers per epoch, the average overflow probability per epoch, and the average migration cost per epoch. We can see that COEM-HCA can significantly decrease the migration cost, while achieve more energy saving and similar SLA performance. We have run simulations with other parameter combinations and the relative performance comparison is similar.

Power-aware Utilization Control. The goal of the utilization controller at the server-level is to guarantee each active server after consolidation to be efficiently utilized by adjusting the CPU frequency so that more power conservation could be achieved. To

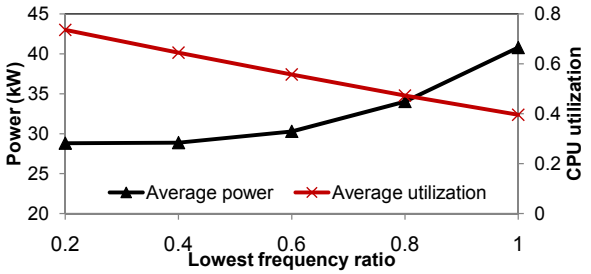


Figure 5.15: Power consumption and average utilization of the cluster after applying the utilization controller

understand the potential energy saving from the local power-aware utilization controller, we derive a server power model under CPU-intensive applications by using the methodology proposed in [84], and integrate it into the simulator. The power model describes the relationship between the server power consumption and the CPU frequency. It is learned on a physical server equipped with a dual-core AMD Opteron Processor 2222 SE, which supports up to 8 frequencies ranging from $1GHz$ to $3GHz$, by running both a CPU-intensive benchmark (*libquantum*) and a memory-intensive benchmark (*mcf*)from CPU SPEC 2006.

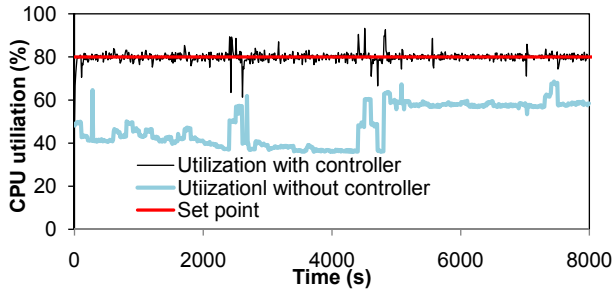
Apparently, the amount of power that could be conserved by the utilization controller depends on hardware capability, aka, CPU voltage/frequency scaling range for DVFS. To understand the impact of hardware platform on the DVFS controller performance, we implemented in the simulator a homogeneous cluster with physical servers having a normalized frequency scaling range $[f_{low}, 1]$, where 1 corresponds to the full CPU frequency, and f_{low}, a percentage of the full CPU frequency, is a variable in the simulation. We calculate the power consumption of active servers during the COEM consolidation shown in Figure 5.13 by using the derived power model and different f_{low} value. Figure 5.15 shows the power consumption and average CPU utilization of the whole cluster after applying the DVFS control. We can see that most of the power saving due to DVFS control can be achieved even with a limited CPU frequency scaling capability (e.g., f_{low} in $[0.6, 1]$). Actually, the further power saving becomes very limited when lowering f_{low} to be < 0.6. This is partially due to high server utilization with COEM server consolidation.

5.4.2 Physical Experiments

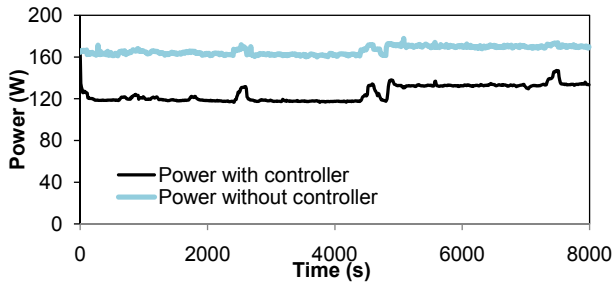
COEM Implementation. Our testbed is based on Usher [65], a virtual machine management framework originally developed at UCSD. We extended Usher to add the three components described in Section 5.3. We implemented system status analysis and VM placement decision as a plugin on the Usher controller, which is invoked periodically with a fixed interval. We extended each Usher local node manager to provide the power-aware utilization control described in Figure 3.20. The utilization control component steers the CPU utilization on each physical host to a desired utilization, which is set by the Usher controller. It formulates a control loop including three parts: a utilization monitor, a utilization controller, and a CPU frequency modulator.

Testbed Experiments

Due to the limitation of our testbed resource, we can not repeat all workload from the trace in the experiment. Instead, we randomly select ten active servers from the simulation for the COEM consolidation shown in Figure 5.13, and reproduce the load demand of each server on the testbed physical servers where the power model in Section 4.3.4 is derived. For each active server, we use the recorded aggregate CPU resource demand to drive the physical server so that we can emulate the load variation of each active server from the simulation on the physical server. We then evaluate the utilization controller in terms of average CPU utilization and power saving. Our experiment shows that the average CPU utilization of the active servers is increased from 49.1% to 79.9%, which is very close to the desired set point of 80% by scaling the CPU frequency. As a result, the average power consumption of the ten servers is reduced from 166.2$watts$ to 135.4$watts$, which ends up with around 18.5% power savings. Figure 5.16 shows the CPU utilization of one active server with and without the utilization controller. We can see that the CPU utilization with the utilization controller can be controlled around the desired set point (*i.e.*, 80%) by scaling down the CPU frequency, in spite of the significant variation of server workload. As a result, the power has a reduction of around 25% in the power measurement, as shown in Figure 5.16(b).



(a) CPU utilization



(b) Power consumption

Figure 5.16: Effects of utilization controller on the utilization and power consumption

This experiment demonstrates that applying the utilization controller on each active server after the server consolidation can efficiently further improve the CPU utilization so that more power is conserved.

Load Variation Testing. As we mentioned in Section 5.3, effective sizing benefits both server consolidation and local utilization control through load variance reduction. In this subsection, we first compare COEM with the baseline of B4 in terms of load variation, measured in Coefficient of Variation (CV), in the simulations. We then investigate the effect of the variation on the performance of the utilization controller in terms of control accuracy and overshooting. Other baseline schemes have similar load variations like B4 and the details are skipped.

Figure 5.17 shows the CV CDF of all active servers under the consolidation schemes of COEM-CA and B4 for the scenarios shown in Figure 5.13. The load variation of each server is calculated based on the workload through the one-week period. Based on the statistics in Figure 5.17, we may conclude that COEM-CA has smoother workload variation on each server than B4. For example, there are 84.6% of the servers which have a CV smaller than 0.5 under COEM-CA, compared with 62.4% of the servers under B4.

	Load 1 (COEM)	Load 2 (B4)
Overshooting time	13.5%	15.2%

Table 5.3: Effects of load variation on controller performance

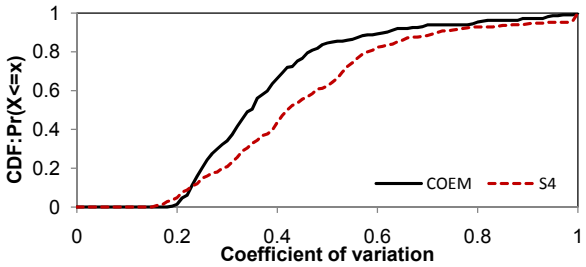


Figure 5.17: Cumulative Distribution Function of the variation of all the servers

Intuitively, smaller load variation may lead to smaller control overhead for local utilization controllers. To investigate the effect of load variation on the performance of the utilization controller, we generate two workloads with different load variations and use them separately to drive a physical server in the testbed. The mean and CVs of the two generated loads are the same as the expected mean and CVs of all the servers in COEM-CA and B4 consolidations shown in Figure 5.13, respectively. We compare the DVFS controller's behaviors under the two workloads in terms of overshooting time. The overshooting time is defined as the time duration in which the CPU utilization is larger than the steady state of the controller (the steady state is defined as ±5% of the set point.). The longer the overshooting, the higher the probability that the performance requirement is violated. Table 5.3 shows the comparison results. We can see that the COEM representative load results in a shorter (11% reduction) overshooting time (in percentage of the total time) of the utilization controller than the B4 representative load.

This load variation testing shows that local utilization controllers can have smaller risk of SLA violation by applying effective sizing.

5.5 Conclusions

We present a case study of virtualized data center energy management based on a large workload trace, which serves a quantitative proof on the potential of significant infrastructure

and energy cost reduction through server consolidation. We also offer a coordinated energy management solution with the goal of rendering the potential into reality.

Server consolidation is a multi-dimensional bin packing problem; we only discuss VM sizing on CPU demand and take memory as a constraint. Other resources like disk IO, network bandwidth can be considered in the future work. Note that effective sizing as a general technology can be applied to those statistically multiplexed resources as well in the provisioning process.

Chapter 6

Conclusions and Future Work

The complexity of large-scale systems has raised unprecedented challenges for system management, which has significantly increased operational costs in recent years and affected system reliability and availability. Basically, today's computing systems face two critical challenges. First, various customers need to be assured by meeting their required service-level agreements such as response time and throughput. Second, server power consumption must be controlled in order to avoid system failures caused by power capacity overload or system overheating due to increasing high server density. Existing work, unfortunately, either relies on open-loop estimations based on off-line profiled system models, or evolves in a more ad hoc fashion, which requires exhaustive iterations of tuning and testing, or oversimplifies the problem by ignoring the coupling between different system characteristics (*i.e.*, response time and throughput, power consumption of different servers). As a result, all of them lack rigorous guarantees on performance and power consumption for computing systems, and may result in degraded overall system performance.

In this thesis, we investigate the application of control-based methodology in computing systems, for adaptive performance management, power management, and power-efficient performance management respectively, by utilizing both a MIMO optimal control design and a hierarchical control design.

- For adaptive performance management, we designed an integrated solution that controls both the average response time and CPU utilization in information dissemination

systems to achieve bounded response time for high-priority information and maximized system throughput in an example information dissemination system, and a hierarchical control solution to guarantee the deadlines of real-time tasks in power grid computing by grouping them based on their characteristics, respectively.

- For adaptive power management, we proposed power control solutions at different levels: cluster level, data center level, and server level. The MIMO control solution for the cluster-level power control can adaptively shift power among servers in a cluster; the MIMO control solution for the server-level power control can optimally allocate power between processor and main memory based on the memory queue level so that improved performance can be achieved; the hierarchical design for power management in data centers is consistent with the power distribution hierarchy of computing systems in large-scale data centers.

- For power-efficient performance management, we discussed a two-layer coordinated management solution for virtualized data centers. At the cluster level, we apply a novel VM sizing technique: effective sizing, on server consolidation which improves server utilization by 11%; at the server level, we apply a power-aware utilization controller to further improve server utilization which give another 18% power saving.

For all of them, experimental results in either physical testbeds or simulations demonstrate that our proposed designs can capture the coupling among different system characteristics and coordinate control signals at different control levels so that they outperform state-of-the-art solutions in performance and power management by significantly improving overall system performance.

I plan to continue my system-oriented research focusing on performance/power challenges in computing systems. My future work will mainly focus on the following two parts: 1) Virtualization, as an efficient technology to consolidate physical resources, simplify deployment and administration, and reduce power and cooling requirements, has been widely adopted in cloud computing. There are a lot of research issues on the challenges of performance and power management in virtualized data centers. For example, most power and performance

management strategies only consider CPU resources during server consolidation, without considering constraints on other resources, such as memory requirement, network bandwidth, etc. As a result, a more systematic strategy that captures the coupling of various resources to optimize server consolidation is very necessary for power and performance management in virtualized data centers. 2) As the ever-increasing complexity of cloud services, many mission critical systems are deployed by integrating thousands of heterogeneous components. The numerous software components running in virtualized computing systems often obscure the dependencies and interactions among system components. Therefore, the interesting questions I want to explore are as follows: 1) how can we locate the performance bottleneck automatically and efficiently? 2) How can we mine bugs out of huge amount of logging information from applications deployed in virtual machines? 3) How can we achieve reliability by providing just-in-time recovery mechanisms? My research work in this direction will adopt new potential methodologies, such as machine learning and data mining, in cloud computing.

Bibliography

Bibliography

[1] Reliability standards for the bulk electric systems of north america. http://www.nerc.com/files/Reliability_Standards_Complete_Set_1Dec08.pdf, 2008. 41, 42, 54

[2] T. Abdelzaher, V. Sharma, and C. Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Trans. on Computers*, 53(3):334–350, 2004. 7

[3] T. Abdelzaher, J. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in sofware services. *IEEE Control Systems*, 23(3), June 2003. 10

[4] AMD. *White Paper Publication 26094: BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors, Revision 3.30*. Advanced Micro Devices, Inc., Feb. 2006. 70

[5] M. Amirijoo, J.Hansson, and S.Son. Specification and management of qos in real-time databases supporting imprecise computations. *IEEE Transactions on Computers*, 55(3):304–319, Mar. 2006. 7

[6] U. N. Bhat. *An Introduction to Queueing Theory: Modeling and Analysis in Applications*. Birkhäuser, 2008. 36

[7] A. Bhattacharjee and M. Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *ISCA*, 2009. 8

[8] R. Bianchini and R. Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11):68–74, 2004. 60

[9] P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. The case for power management in web servers. *Power Aware Computing*, 2002. 62

[10] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *HPCA*, 2001. 9

[11] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a Framework for Architectural-level Power Analysis and Optimizations. In *ISCA*, 2000. 107

[12] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. In *Technical Report CS-TR-1997-1342, University of Wisconsin, Madison*, 1997. 107

[13] F. Chen, X. Han, Z. Pan, and L. Han. State estimation model and algorithm including pmu. *DRPT*, 2008. 51

[14] M. Chen, C. Nolan, X. Wang, S. Adhikari, F. Li, and H. Qi. Hierarchical utilization control for real-time and resilient power grid. In *Euromicro Conference on Real-Time Systems (ECRTS)*, Jul 2009. 5

[15] M. Chen, X. Wang, R. Gunasekaran, H. Qi, and M. Shankar. Control-based real-time metadata matching for information dissemination. In *RTCSA*, 2008. 4

[16] M. Chen, X. Wang, and B. Taylor. Integrated control of matching delay and cpu utilization in information dissemination systems. In *International Workshop on Quality-of-Service (IWQoS)*, Jul 2009. 4

[17] V. Delaluz, M. T. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In *HPCA*, 2001. 60

[18] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server. In *Network Operations and Management Symposium*, 2002. 6, 10

[19] B. Diniz, D. Guedes, W. M. Jr., and R. Bianchini. Limiting the Power Consumption of Main Memory. In *ISCA*, 2007. 9, 96, 97

[20] R. Durrett. *Probability: theory and examples.* Duxbury Press, Belmont, CA, second edition, 1996. 131

[21] Electronic Educational Devices Inc. Watts Up Pro Power Meter. http://www.wattsupmeters.com. 69

[22] X. Fan, W.-D. Weber, and L. A. Barroso. Power Provisioning for a Warehouse-sized Computer. In *ISCA*, 2007. 8, 77, 79, 93

[23] W. Felter, K. Rajamani, and T. Keller. A Performance-Conserving Approach for Reducing Peak Power Consumption in Server Systems. In *ICS*, 2005. 9, 95, 109

[24] W. Felter, K. Rajamani, T. Keller, and C. Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *ICS*, 2005. 60

[25] M. E. Femal and V. W. Freeh. Boosting data center performance through non-uniform power allocation. In *ICAC*, 2005. 9

[26] G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems, 3rd edition.* Addition-Wesley, 1997. 15, 16, 17, 18, 19, 24, 50, 64, 84, 100

[27] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, London, UK, 1981. 86

[28] G. Glanzmann and G. Andersson. Incorporation of n-1 security into optimal power flow for facts control. In *PSCE*, 2006. 41

[29] A. Goel and P. Indyk. Stochastic load balancing and related problems. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 579–586, 1999. 129

[30] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: dynamic speed control for power management in server class disks. In *ISCA*, 2003. 60

[31] T. Gustafsson and J. Hansson. Dynamic on-demand updating of data in real-time database systems. In *ACM Special Interest Group on Applied Computing (SIGAPP)*, Mar. 2004. 7

[32] T. Gustafsson and J.Hansson. Data management in realtime systems: a case of on-demand updates in vehicle control systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2004. 7

[33] J. R. Haritsa, M. J. Carey, and M. Livny. Value-based scheduling in real-time database systems. *VLDB Journal: Very Large Data Bases*, 2(2):117–152, 1993. 7

[34] F. Hayes-Roth. Model-Based Communication Networks and VIRT: Orders of Magnitude Better for Information Superiority. *Military Communications Conference, 2006. MILCOM 2006*, pages 1–7, Oct. 2006. 12

[35] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. 9, 24

[36] C. Hollov, V. Misra, D. Towsley, and W.-B. Gong. A control theoretic analysis of red. In *INFOCOM*, 2001. 10

[37] 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. 6, 8, 10

[38] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. Improving Energy Efficiency by Making DRAM Less Randomly Accessed. In *ISLPED*, 2005. 8

[39] J. Y. Hui. Resource allocation for broadband networks. pages 358–368, 1991. 130

[40] INFOD-WG. *Information Dissemination in the Grid Environment - Base Specifications*. Open Grid Forum (2004-2007), May 2007. 12

[41] Innovative Computing Laboratory, University of Tennessee. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/. 72

[42] Intel. Intel 845 Chipset: 82845 Memory Controller Hub (MCH) Datasheet. http://developer.intel.com/Assets/PDF/datasheet/290725.pdf, 2002. 104

[43] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *MICRO*, 2006. 9, 108

[44] B. Jacob, Spencer, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007. 96

[45] K. Kang, S. Son, and J. Stankovic. Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(10), Oct. 2004. 7

[46] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *ISCA*, July 2001. 8

[47] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators. In *HPCA*, 2008. 111

[48] J. Kleinberg, Y. Rabani, and E. Tardos. Allocating bandwidth for bursty connections. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, New York, NY, USA, 1997. ACM. 129

[49] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, 2003. 115

[50] C. Lefurgy, X. Wang, and M. Ware. Server-level power control. In *Proceedings of the 4th IEEE International Conference on Autonomic Computing (ICAC)*, 2007. 60, 61, 62, 63, 70, 72, 73, 75, 87

[51] C. Lefurgy, X. Wang, and M. Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11, 2008. 9, 10, 95, 101, 107, 109, 112

[52] F. L. Lewis and V. L. Syrmos. *Optimal Control, Second Edition*. John Wiley & Sons, Inc., 1995. 68

[53] X. Li, R. Gupta, S. V. Adve, and Y. Zhou. Cross-Component Energy Management: Joint Adaptation of Processor and Memory. *ACM Transactions on Architecture and Code Optimization*, 4, 2007. 8, 95

[54] X. Li, Z. Li, F. David, P. Zhou, Y. Zhou, S. Adve, and S. Kumar. Performance Directed Energy Management for Main Memory and Disks. In *ASPLOS*, 2004. 8

[55] J. Lin, H. Zheng, Z. Zhu, H. David, and Z. Zhang. Thermal Modeling and Management of DRAM Memory Systems. In *ISCA*, July 2007. 9

[56] J. Lin, H. Zheng, Z. Zhu, E. Gorbatov, H. David, and Z. Zhang. Software Thermal Management of DRAM Memory for Multicore Systems. In *SIGMETRICS*, 2008. 94

[57] B. Liscouski and W. Elliot. Final report on the august 14, 2003 blackout in the united states and canada: Causes and recommendations. In *A report to U.S. Department of Energy*, 2004. 40

[58] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000. 7, 43, 53, 54

[59] X. Liu and T. Abdelzaher. On Non-Utilization Bounds for Arbitrary Fixed Priority Policies. In *RTAS*, Apr. 2006. 7

[60] C. Lu, J. A. Stankovic, and S. H. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 23:85–126, 2002. 7, 10

[61] C. Lu, X. Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):550–561, June 2005. 44, 47, 49, 53, 86

[62] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for qos guarantees and its application to differentiated caching services. In *IWQoS*, 2002. 10

[63] Y. Lu, A. Sexana, and T. Abdelzahe. Differentiated caching services; a control-theoretical approach. In *ICDCS*, 2001. 10

[64] J. M. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, 2002. 65, 67, 84, 102, 103

[65] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In *Proceedings of the 21st Large Installation System Administration Conference (LISA)*, November 2007. 143

[66] Micron. Data Sheet. http://download.micron.com/pdf/datasheets/modules/ddr2 /HTF9C64_128x72F.pdf, 2007. 107

[67] Micron. Micron System Power Calculator. http://download.micron.com/downloads /misc/ddr2_power_calc_web.xls, 2007. 107

[68] Micron. Data Sheet. http://download.micron.com/pdf/datasheets/dram/ddr2/1GbDDR2.pdf, 2009. 96

[69] 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. 60, 61

[70] Q. N.Ahmed and S. V.Vrbsky. Triggered Updates for Temporal Consistency in Real-TimeDatabases. *Real-Time Systems*, 19(3):209 – 243, Nov. 2000. 7

[71] C. Patel, C. Bash, R. Sharma, M. Beitelmal, and R. Friedrich. Smart cooling of data centers. In *Proceedings of the ASME Interpack*, Maui, Hawaii, July 2003. 61

[72] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No Ṕower Struggle!́ Coordinated Multi-level Power Management for the Data Center. In *ASPLOS*, 2008. 9, 78, 93, 101

[73] P. Ranganathan, P. Leech, D. Irwin, and J. S. Chase. Ensemble-level power management for dense blade servers. In *ISCA*, 2006. 9, 10, 71, 78

[74] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *HPCA*, Washington, DC, USA, 2002. 61

[75] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware Microarchitecture: Modeling and Implementation. *ACM Transactions on Architecture and Code Optimization*, 1, Mar. 2004. 107

[76] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodicresponsiveness in hard real-time environments. *IEEE Trans. on Computers*, 44(1), Jan. 1995. 7

[77] P. Tondel, T. Johansen, and A. Bemporad. An algorithm for multi-parametric quadratic programming and explicit MPC solutions. In *CDC*, 2001. 104

[78] M. Verhaegen and V. Verdult. *Filtering and System Identification, A Least Square Approach*. Cambridge University Press, 2007. 15, 24

[79] A. Verma, P. Ahuja, and A. Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, New York, NY, USA, 2008. 134

[80] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. Server Workload Analysis for Power Minimization using Consolidation. In *USENIX*, 2009. 8, 122, 126, 127, 132, 139

[81] VMware. *VMware Distributed Power Management Concepts and Use*. http://www.vmware.com/files/pdf/DPM.pdf. 129, 138

[82] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, , and B. Jacob. DRAMsim: A Memory System Simulator. *SIGARCH Computer Architecture News*, 33, Nov. 2005. 107

[83] X. Wang and M. Chen. Feedback Multi-Server Power Control in a Computing Cluster, Tech Report, Electrical Engineering and Computer Science, University of Tennessee. http://www.ece.utk.edu/∼xwang/papers/power-tr.pdf, 2007. 88

[84] X. Wang and M. Chen. Cluster-level feedback power control for performance optimization. In *HPCA*, 2008. 5, 142

[85] X. Wang, M. Chen, C. Lefurgy, and T. W. Keller. SHIP: Scalable Hierarchical Power Control for Large-Scale Data Centers. In *PACT*, 2009. 5

[86] X. Wang, D. Jia, C. Lu, and X. Koutsoukos. DEUCON: Decentralized end-to-end utilization control for distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(7), 2007. 78, 84

[87] Y. Wang, X. Wang, M. Chen, and X. Zhu. Power-efficient response time guarantees for virtualized enterprise servers. In *RTSS*, 2008. 6

[88] X. Wei. On estimation of autoregressive signals in the presence of noise. *IEEE Trans. on Circuits and Systems*, 53(12), Dec. 2006. 53

[89] A. J. Wood and B. F. Wollenberg. *Power Generation, Operation, and Control, the second edition*. Wiley-Interscience, 1996. 41, 42

[90] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors. In *ASPLOS*, 2004. 8

[91] Q. Wu, P. Juang, M. Martonosi, L.-S. Peh, and D. W. Clark. Formal control techniques for power-performance management. *IEEE Micro*, 25(5), 2005. 61

[92] H. Xue, Q. quan Jia, N. Wang, Z. qian Bo, H. tang Wang, and H. xia Ma. A dynamic state estimation method with pmu and scada measurement for power systems. *IPEC*, 2007. 51

[93] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency. In *Micro*, 2008. 8

[94] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping Disk Arrays Sleep through the Winter. In *SOSP*, 2005. 8

Vita

Ming Chen was born in Hengyang, Hunan Province, China. He received B.S. and M.S. both in Electronics and Information Engineering from Northwestern Polythechnical University in Xi'an, China, in 2002 and 2005, respectively. Before he began his Ph.D study at the University of Tennessee, Knoxville, in August 2006, he worked as an embedded software engineer at Embedded Multimedia Software Division of ZTE Corp. in Shenzhen, China.