11-10-2014

Delay-Sensitive Service Request Scheduling for Cloud Computing

Shuo Liu
mercedes1883@gmail.com

DOI: 10.25148/etd.FI14110731

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

DELAY-SENSITIVE SERVICE REQUEST SCHEDULING FOR CLOUD COMPUTING

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

by

Shuo Liu

2014

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Shuo Liu, and entitled Delay-Sensitive Service Request Scheduling for Cloud Computing, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Kang K. Yen

Jean H. Andrian

Nezih Pala

Ming Zhao

Gang Quan, Major Professor

Date of Defense: November 10, 2014

The dissertation of Shuo Liu is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2014

DEDICATION

To my family.

ACKNOWLEDGMENTS

First of all, I would like to express my deepest gratitude to my major advisor, Dr. Gang Quan, for his endless guidance and help during the last five years of my doctoral study. His dedication to science and research will inspire me for the rest of my life.

I would also like to express my deepest appreciation to my Ph.D. committee members, Dr. Ming Zhao, Dr. Kang K. Yen, Dr. Jean H. Andrian, and Dr. Nezih Pala, for their insightful suggestions and comments for improving the quality of this dissertation. It is extremely wonderful to have these fantastic and knowledgeable professors serving as my committee members.

Next, I want to thank my lab mates, Mr. Qiushi Han, Mr. Tianyi Wang, Mr. Shi Sha, Dr. Ming Fan, Dr. Vivek Chaturvedi, Dr. Huang Huang, and Dr. Guanglei Liu, for creating an amazing working environment and life-long friendships.

Additionally, I would like to thank the National Science Foundation (NSF). This dissertation is supported in part by NSF under projects CNS-0969013, CNS-0917021, CNS-1018108, CNS-1018731, and CNS-1423137.

Last but not least, thanks to my family for everything.

ABSTRACT OF THE DISSERTATION

DELAY-SENSITIVE SERVICE REQUEST SCHEDULING FOR CLOUD COMPUTING

by

Shuo Liu

Florida International University, 2014

Miami, Florida

Professor Gang Quan, Major Professor

Cloud computing realizes the long-held dream of converting computing capability into a type of utility. It has the potential to fundamentally change the landscape of the IT industry and our way of life. However, as cloud computing expanding substantially in both scale and scope, ensuring its sustainable growth is a critical problem. Service providers have long been suffering from high operational costs. Especially the costs associated with the skyrocketing power consumption of large data centers. In the meantime, while efficient power/energy utilization is indispensable for the sustainable growth of cloud computing, service providers must also satisfy a user's quality of service (QoS) requirements. This problem becomes even more challenging considering the increasingly stringent power/energy and QoS constraints, as well as other factors such as the highly dynamic, heterogeneous, and distributed nature of the computing infrastructures, etc.

In this dissertation, we study the problem of delay-sensitive cloud service scheduling for the sustainable development of cloud computing. We first focus our research on the development of scheduling methods for delay-sensitive cloud services on a single server with the goal of maximizing a service provider's profit. We then extend our study to scheduling cloud services in distributed environments. In particular, we develop a queue-based model and derive efficient request dispatching and processing

decisions in a multi-electricity-market environment to improve the profits for service providers. We next study a problem of multi-tier service scheduling. By carefully assigning sub deadlines to the service tiers, our approach can significantly improve resource usage efficiencies with statistically guaranteed QoS. Finally, we study the power conscious resource provision problem for service requests with different QoS requirements. By properly sharing computing resources among different requests, our method statistically guarantees all QoS requirements with a minimized number of powered-on servers and thus the power consumptions. The significance of our research is that it is one part of the integrated effort from both industry and academia to ensure the sustainable growth of cloud computing as it continues to evolve and change our society profoundly.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Cloud computing realizes a long-held dream of converting computing capability into a web-based utility. Cloud service providers have been striving to enrich their product lines since the outset to facilitate people's lives. People now are able to share pictures or videos (e.g. DropBox [32], GoogleDocs [44]) and deploy heavy computation tasks (e.g. Amazon EC2 [5]) easily on the Internet without worrying about the limit of storage space, computing capacity, and the locations where storage and computations take place. Meanwhile, IT companies greatly benefit from cloud computing as well. The elasticity and high accessibility of cloud computing allows them to deploy their online services (e.g. Google AppEngine [43]) or even their IT infrastructures (e.g. Rackspace [97]) in an easier, cheaper, and faster way [12].

Cloud computing is drastically changing not only people's living style, but also the IT industry landscape. While people are enjoying the advances brought by cloud computing, cloud service providers are facing challenging resource management problems because of growing service demands. Simply increasing servers' computing capacities is not sufficient for today's cloud ecosystem. More sophisticated resource management strategies are urgently needed to achieve multiple objectives simultaneously.

In this chapter, we first introduce cloud computing and its prosperity, and then present the opportunities and challenges in the design of effective and efficient resource management approaches for delay-sensitive services in a cloud environment. Whereafter, we define our research problem and summarize our contributions. The structure of this dissertation is illustrated at the end of this chapter.

1.1 Cloud Computing

Cloud computing enables web-based on-demand deliveries of elastic computing resources and applications with "pay-as-you-go" pricing [18][125]. It helps organizations or IT companies avoid the high capital investments on computing facilities and the consequent expenses on facility management. At the same time, cloud computing also provides them with reliable accessibility, scalability, disaster tolerance, etc.

Cloud computing has been widely employed by various participants. The past decade witnessed an expeditious growth of cloud computing. Based on the statistics gathered by NSK Inc. [109], banking contributes the most activity in the cloud (around 64%), especially after the introduction of mobile banking. Web social media takes second place with around 58% cloud activities, followed by online games, photos, and file sharing. Even the governmental IT deployment turns towards the cloud [109]. Because of the scalable and close control over the private cloud environment, over half of the United States government is using cloud computing services and around $2 billion is spent on those services annually. Among the surveyed companies, 82% of them saved money by moving to the cloud, and 14% downsized their IT infrastructures after cloud adoptions.

Another set of statistics showing the popularity of cloud computing comes from Amazon. Figure 1.1 shows the growth of EC2's (Amazon's infrastructure service) over each year from 2007 to 2010. In 2007, only $2,643$ EC2 launches were recorded. Since then, this number has been increasing drastically. In particular, from 2008 to 2009, a 375% boost was recorded. Figure 1.2 shows an exponential increase of the objects stored in Amazon S3 (Amazon's storage service) over each year from 2006 to 2013. There were around 2 trillion objects stored in Amazon S3 in the second

quarter of 2013, which is a tremendous achievement compared to the number of stored objects in the fourth quarter of 2006 (less than 0.1 trillion).

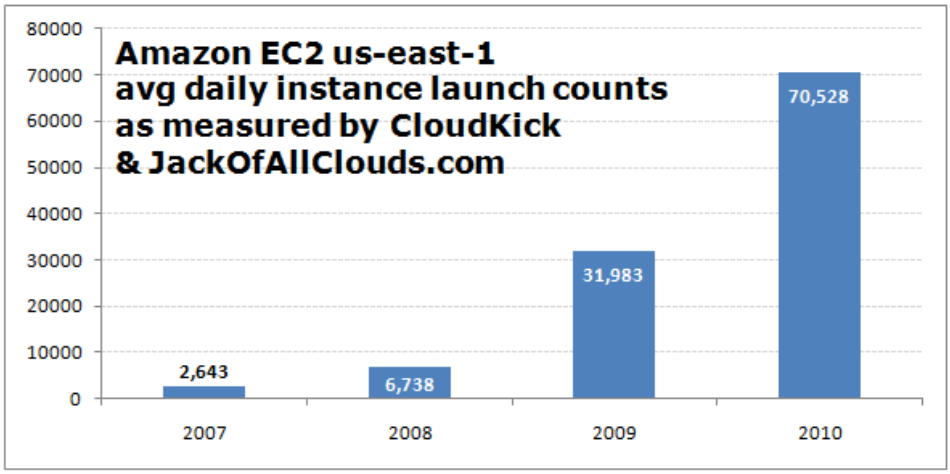


Figure 1.1: Amazon EC2's growth over the years [33].

1.2 Data Centers for Cloud Computing

Data centers, as the backbone of service provision, have played a critical role in the proliferation of cloud computing's scale and scope. The number of data centers has grown rapidly for the last decade. Now a service provider may have one or several geographically distributed data centers (leased from other data center services or built by the service provider themselves) for the sake of high availability, disaster tolerance, uniform response time, etc. Take Google as an example, it has data centers in Portland, Oregon; Houston, Texas; Atlanta, Georgia, etc.

Besides the growing number of data centers, the sizes of newly built data centers constantly refresh the record. Take Microsoft as an example; its first data center built in 1989 just occupied $89,000$ sq ft. In 2006, the size of its new data center increased to $500,000$ sq ft [34]. Around 2013, Google built a data center of approximately $1,000,000$ sq ft in Mayes county, Oklahoma, which was the largest known

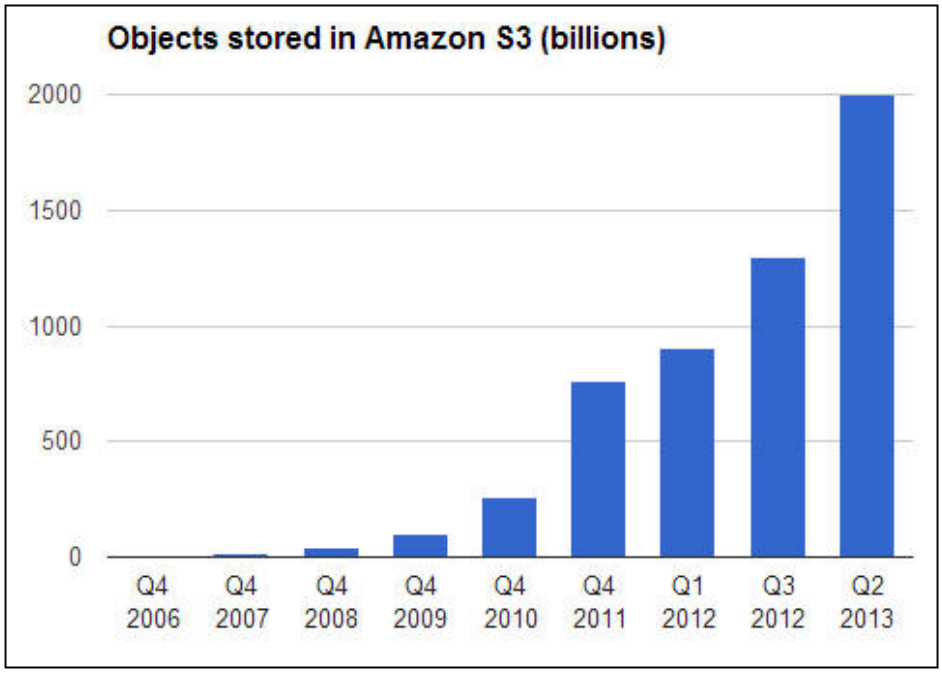


Figure 1.2: The growth of the objects stored in Amazon S3 over the years [103].

data center at that time. However, recently SuperNap built a $2,200,000$ sq ft data center in Las Vegas, which is more than twice the size of Google's largest data center [38].

Data centers provide robust support for the services hosted inside. They help service providers generate revenue by providing quality guaranteed services. Meanwhile, a significant amount of expenses are incurred to operate these data centers.

1.3 Challenges for Cloud Computing Resource Management

Behind the glory of cloud computing's prosperity, service providers have to face several grave challenges which make cloud computing resource management critical for a service provider's sustainable growth.

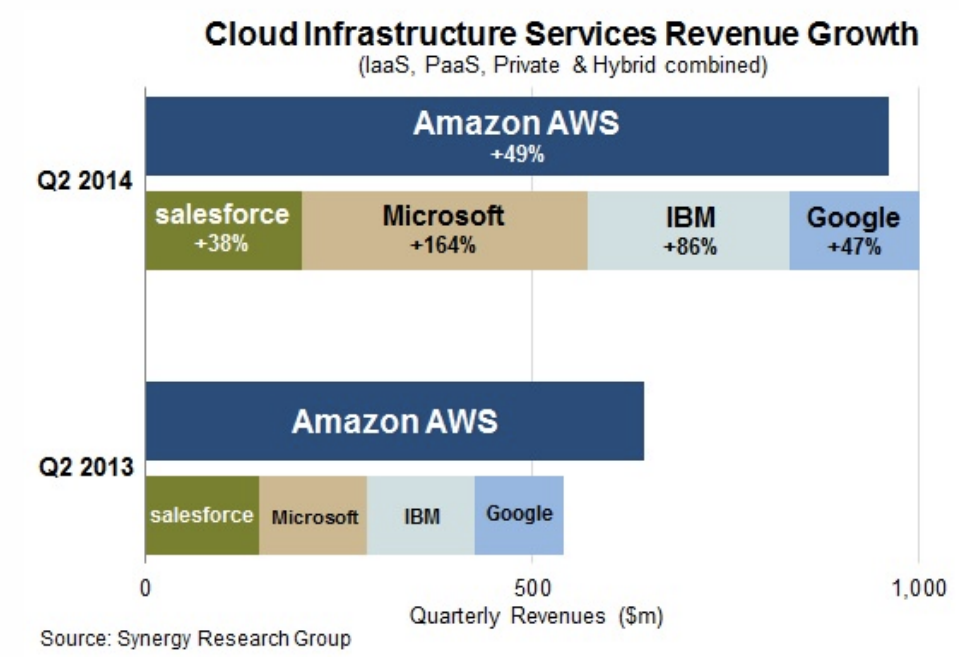


Figure 1.3: Cloud infrastructure revenue growth of several major cloud service providers [46][111].

1.3.1 Revenue and Operational Cost

Service providers are experiencing a fast growth in both their business scales and their revenues. As reflected by Figure 1.3, in the second quarter of 2014, all the web giants have an increase on their revenues for cloud infrastructure services. Microsoft's revenue shot up with a 164% increase, and took the first place among the different service providers, followed by IBM, Google, Amazon, and Salesforce. Even though the annual growth of Amazon dropped to 49% because of the strong rivals, it could not conceal the vitality of the web service market [46][111].

Behind the business and revenue growth, high operational cost becomes a hard nut to crack. The increasing rate of computing capacity (Moore's Law) is greater than the increasing rate of power efficiency [22]. As shown in Figure 1.4, with a persistent exponential rise of the number of installed servers in data centers, the spending on power and cooling, and management and administration in data centers

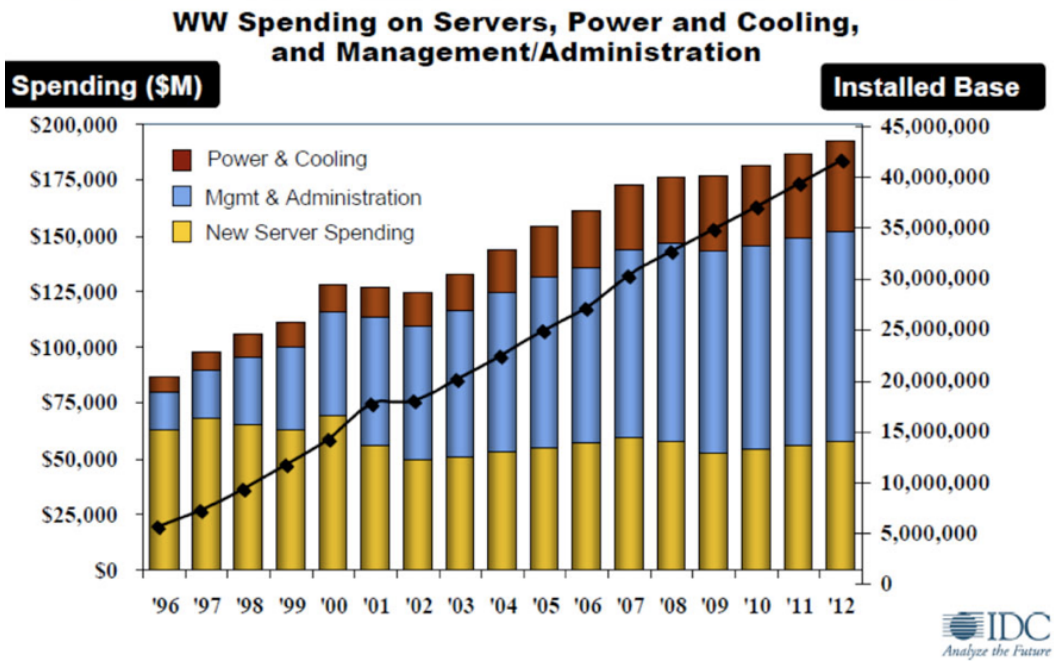


Figure 1.4: Worldwide spending on servers, power and cooling, and management and administration [31].

has been continuously increasing since 1996. There are a total of $900,000$ servers running in Google's data centers and cost approximately 220 megawatts of power. This number accounts for 0.01% of the world wide electricity use, and is enough to power $200,000$ homes [45][34]. Google has to pay around $36M$ annually for their electricity costs [96].

As a result, the effective resource management for data centers is vital not only for the survival and continue growth for service providers in the highly competitive business environment, but also to promote green and environmentally responsible computing and minimize the adverse impacts on our global environment.

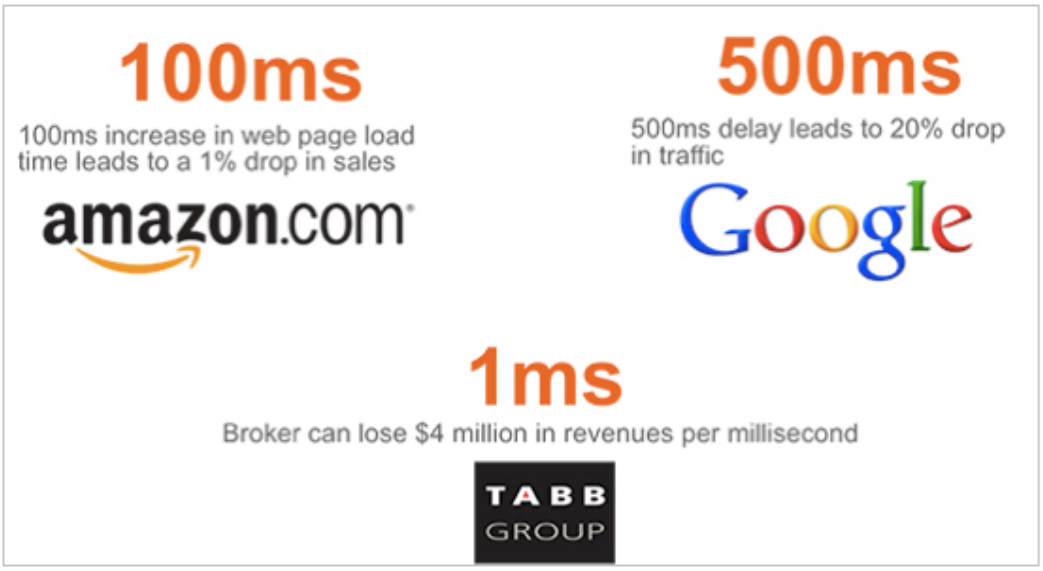


Figure 1.5: Response time effects on service providers[50]

1.3.2 Delay-Sensitive Service Requests

While power/energy efficiency is critical for cloud computing, service providers must provide services that can satisfy the QoS requirements for a user's service requests. The service latency, for example, is usually closely related to the QoS.

From the revenue perspective, many cloud services usually consist of interactive applications, which are commonly soft real-time in nature (e.g. online gaming and stream media). A late response will not cause any catastrophic consequences. However, it degrades service quality and may lead to low profits and loss of future customers [93][68]. As shown in Figure 1.5, for Amazon, a $100ms$ increase in the web page loading time leads to a 1% drop in sales. In Google, a $500ms$ delay results in a 20% drop in traffic. The statistics gathered by TABB GROUP indicate that $1ms$ may cause a broker to lose $4 million in revenues. Therefore, successfully guaranteeing service request response time is critical for a service provider's reputation and, ultimately, its revenue.

1.3.3 Heterogeneities of Physical Servers, Data Centers, and Services

The challenges for resource management also come from the large heterogeneities in the cloud. From the hardware infrastructure perspective, a service provider may deploy a large variety of computation components [67]. Previously, Xeon almost monopolized the data center CPU market. Many energy-efficient processors, such as Atom [52] and ARM [11], are exploring their own places in today's data centers. Moreover, newly developed subsystems (e.g. GPU accelerators and solid state drives) increase the heterogeneities of computing infrastructures.

From the data center perspective, data centers of the same service provider are usually geographically distributed, and work in different environments with different electricity prices, computing capacities, and communication costs.

With so many heterogeneities being considered, it is not easy to achieve the balance of the trade-off between service quality and processing efficiency in terms of low dollar cost or low power/energy consumption.

Additionally, cloud services impose more heterogeneities from the perspective of various service level agreements (SLAs). As the contracts decided between cloud service providers and cloud customers, SLAs define the specific QoS requirements. In order to make revenue, a service provider has to honor the QoS requirement. However, how to guarantee the different QoS requirements simultaneously is not a trivial work, especially in resource stringent situations.

The heterogeneities in cloud computing raise many resource management challenges for service providers. Meanwhile, the heterogeneities provide service providers with opportunities to improve their efficiencies by judiciously managing their computing and energy resources.

1.3.4 Dynamics in Cloud Computing

Last but not the least, the large run-time dynamics in cloud computing make the resource management even more challenging for service providers. The dynamics come from both customers and computing facilities.

Since the service demands change abruptly during a day, traditional long-term capacity plans with more than enough resources provisioned lead to a considerable resource waste. In order to cope with fluctuations of service demands, service providers need to plan the capacity provisions with more fine-grained time scales (e.g. hourly based capacity plans). In addition, computing facility dynamics affect resource management decisions as well. For example, the electricity price at a data center location may vary notably throughout a day because of the electricity generating technology and electricity consumption statistics at the location [120]. A service provider has to judiciously dispatch the workload among its data centers locations according to the corresponding electricity prices in order to lower its expense on power consumptions.

As evidenced in the previous work [120][99], to deal with run-time dynamics effectively is not only necessary but also can be extremely lucrative for service providers.

1.4 Research Problem

In this dissertation, we are interested in the research problem of developing advanced, effective resource management methods and techniques for service providers to balance the trade-offs among various factors. Specifically, our research problem can be formulated as the following:

Problem. *Given:*

- *the performance characteristics of a cloud platform,*

  — e.g. number of potentially geographically distributed data centers, servers in a data centers, network performance, power consumption, etc.

- *the specifications of a set of service requests,*

  — e.g. timing specifications, workload demands, QoS requirements, profit/penalty features, etc.

- *the design objectives and goals,*

  — e.g. profit maximization, electricity cost minimization, power consumption minimization, etc.

determine where and how to serve the requests to optimize the design objectives.

1.5 Our Contributions

Towards the research problem, we make the following contributions:

1. First, we studied the scheduling problem for single-tier delay-sensitive services on a single server. Specifically, we studied how to employ the profit and penalty-aware scheduling approaches to maximize a service provider's net profit. Each service request is associated with a preferred finishing time. The earlier a request can be finished, the higher the profit is. Contrarily, the more delay exceeding the preferred finishing time, the higher the penalty is. Two time utility functions (TUFs), one for profit and the other for penalty, are assigned to each request for indicating the priority it has. Based on the priorities, the requests in a server are scheduled in a non-preemptive manner. Compared to the reference model based on which our methods are built, our

methods have a significant improvement on a service provider's net profit. We also extended the scheduling approaches to a preemptive one with carefully designed preemption constraints.

2. Then, we studied the problem of single-tier delay-sensitive service scheduling in a distributed environment. Specifically, we studied the profit and penalty-aware request scheduling in a multi-electricity-market environment, in which the electricity prices at different locations vary significantly. Even for the same location, its electricity price varies differently throughout a day. By taking advantage of different processing costs and processing capacities at different data center locations, we devised a request dispatching and scheduling method, which significantly outperforms the static load dispatching strategy consistently. We further extended this method by adopting the queuing model to model the service requests and their processing, which are highly dynamic. From our simulation study, we found that our method achieved 130% maximum higher net profit compared to a static workload dispatching method.

3. Next, we studied the scheduling problem for multi-tier services. Different from single-tier services, multi-tier services have intertwined dependencies among different tiers. A delay at early stages may result in an unpredictable request failure at the end. In addition, many services provided in the cloud environment have finer defined SLAs (e.g. the response time of the service requests have to be guaranteed statistically instead of just guaranteeing the average cases). To this end, we designed an algorithm to statistically assign sub deadlines to the service tiers. A request is removed from the request queue if it misses its local deadline. The method is able to discover potential failure requests and remove them at early stages. The precious computing resources can be saved for other requests that are more likely to be successfully ful-

filled. Compared to other traditional methods (e.g. fist come first serve), our method is able to achieve an 80% shorter average response time with statistically guaranteed QoS in resource stringent situations. We further considered the situation in which different multi-tier services share the same processing units. This problem is more complicated because the sub-deadline decisions are made interdependently among the different services. Our designed sub-deadline assignment approach achieves a 50% shorter average response time compared to traditional methods with statistically guaranteed QoS requirements.

4. Finally, we studied the problem of power minimization with statistically guaranteed QoS. The study was conducted in a shared environment, in which the same service requests but with different QoS requirements share the same computing unit. Since a server's static power takes a large portion (around 70%) of its total power consumption, we converted the request allocation problem to a bin packing related problem and devised a method that is able to minimize the number of powered-on servers by judiciously packing various types of requests into the same server. Our solution achieves a 55% maximum in power saving compared to traditional methods.

1.6 Structure of the Dissertation

The rest of the dissertation is organized as follows. We discuss the related work in Chapter 2. In Chapter 3, we present our research for the scheduling problem for single-tier services on a single server in order to maximize a service provider's net profit. In Chapter 4, we extend the profit and penalty-aware strategy into a multi-electricity-market environment. We discuss our study for the multi-tier service

scheduling problem in Chapter 5 and devise a sub deadline assignment method to statistically guarantee each request's QoS. Chapter 6 presents a power minimization approach which is able to minimize a service provider's power consumption with statistically guaranteed QoS requirements. Finally, we conclude this dissertation and discuss possible future work in Chapter 7.

CHAPTER 2

RELATED WORK

In this chapter, we present the related work. We first discuss traditional real-time scheduling for embedded systems. Next, we discuss the differences between the scheduling for embedded systems and the scheduling for cloud systems, followed by the discussion of extensive scheduling studies that focus on cloud computing. Then, we make a summary at the end of this chapter.

2.1 Real-Time Scheduling

In real-time systems, the success of an execution depends not only on the correctness of the logical computational results, but also on the time at which the execution finishes. An execution in a real-time system is usually associated with a deadline. The violation of a deadline (i.e. the timing constraint) degrades the quality of service, and may even result in unexpected catastrophes (e.g. air traffic control) [106][75]. Therefore, scheduling becomes an effective way to avoid the timing constraint violations.

In general, real-time scheduling studies the problem of judiciously deciding when and where to start the execution of a task in order to successfully guarantee a system's timing constraints. Meanwhile, some other design optimizations can be achieved (e.g. power/energy minimization).

Real-time scheduling can be categorized into several sub-divisions. For example, according to the timing constraint property, real-time scheduling can be divided into hard [74][87][113] and soft [133][100][59][60] real-time scheduling. In hard real-time scheduling, a request has to meet its timing constraint (i.e. deadline) strictly. No timing constraint violation is allowed since an violation in hard real-time systems will cause severe consequences. On the contrary, soft real-time does not have a

strict timing constraint. It is allowed to have some deadline violations. However, the violations will degrade the service quality that the requests receive.

According to when scheduling decisions are made, real-time scheduling can be divided into static [91][21] and dynamic [105][7] real-time scheduling. Static scheduling methods make the scheduling decisions based on the prior knowledge of a request's parameters (e.g. each request's arrival time, deadline, execution time, etc). Once such a decision is made, it will not be changed during its execution. An example of static real-time scheduling is the rate monotonic scheduling (RMS) [69]. On the other hand, dynamic scheduling uses run-time information (e.g. the earliest deadline, the highest profit, the most recent arrivals, etc). The scheduling decision changes during its execution. The earliest deadline first (EDF) is an instance of dynamic real-time scheduling.

According to the scheduling mechanism, real-time scheduling can be divided into preemptive [135][42][123][88] and non-preemptive [17][72][56] real-time scheduling. If the highest priority request is able to stop the execution of a low priority request and start its own execution immediately, then the scheduling is called a preemptive scheduling. Otherwise, it is a non-preemptive scheduling, in which a high priority request has to wait until its preceding low priority request finishes its execution.

According to the platform, real-time scheduling can be categorized into single-core [74] and multi-core [106] scheduling. Different from single-core scheduling, multi-core scheduling not only has to decide task processing sequence, but also needs to decide where a task should be processed. Therefore, multi-core scheduling is more complicated than the scheduling for single cores.

2.2 Scheduling for Delay-Sensitive Cloud Services

There is no distinct boundary between the scheduling for traditional real-time systems and the scheduling for cloud computing. However, they have different focuses in the following perspectives:

- *Task Model:*

  The task model for traditional embedded system can be significantly different from that in cloud computing. The tasks in embedded systems usually have well-defined parameters (e.g. arrival times and execution times). To this end, the scheduling for traditional embedded systems is usually deterministic. On the contrary, in cloud computing, most parameters of a service request are statistic ones, and correspondingly, statistic scheduling is needed for cloud computing in most cases.

- *Platform:*

  The scheduling for traditional embedded systems is commonly conducted on single-core or multi-core platforms. All the computing components are closely coupled. Differently, cloud computing takes place in distributed environments (e.g. in a server farm or geographically distributed data centers), where computing units are loosely coupled. Additionally, more factors have to be considered (e.g. electricity prices, large heterogeneities of computing units, service demand dynamics, etc).

- *Design Objective:*

  The scheduling for traditional embedded systems usually works in time critical situations and is devised according to worst case analyses. Different from the traditional real-time scheduling, the scheduling for cloud computing is more

focused on statistical timing constraint satisfactions (e.g. 90% of the requests have to be finished before the deadline).

As our research work is focused on cloud computing platform, in what follows, we survey the scheduling research for cloud platforms. Specifically, we category the existing work into three groups: (i) scheduling in a single server; (ii) scheduling in a single data center; and (iii) scheduling among multiple data centers.

2.2.1 Scheduling in a Single Server

We first discuss the related work for single servers. Many of the studies in this category employ the traditional utility accrual (UA) approach [30][79].

Jensen et al. [57] first introduced the concept of TUFs, which are usually employed to indicate the fact that the different completion times of a request will contribute differently to the system. Based on the TUF model, several UA based [30, 79] scheduling approaches have been developed (e.g. [70, 71, 128, 126, 127]).

For example, Li et al. in [70] proposed an algorithm called "Generic Benefit Scheduling" (GBS) based on TUF to schedule activities that are subject to various timing and mutually exclusive resource constraints. Utility density is implemented as the activity's priority metric. In their work, only the profit that a request contributes to the system when successfully processed is considered.

Later, a number of methods [16][29][94][53] were proposed to account for the penalty when a request is discarded or misses its deadline. For example, Bartal [16] et al. studied the on-line scheduling problem when penalties have to be paid for rejected real-time tasks. Chun et al. [29] and Irwin et al. [53] adopted an extended time utility function. A decay rate is associated with each real-time task, reflecting

the increasing risk of completing the task late. Therefore, when a real-time task is completed late, it earns a negative utility, indicating a penalty rather than a profit.

In [132], Yu et al developed a profit and penalty-aware scheduling method. Specifically, a task is associated with two different TUFs: profit and penalty TUFs. A profit is assigned to the system if a request completes by its deadline. The system may suffer from a penalty because of a deadline missing or an early discard. By considering the potential profit and penalty that the requests may achieve, the method judiciously schedules the requests to maximize a service provider's net profit.

2.2.2 Scheduling in a Single Data Center

We then discuss the literature related to the scheduling for a single data center, which is more complicated than the scheduling for a single server. Several design parameters have to be decided simultaneously (e.g. workload dispatching, resource allocation, server working frequencies, etc.). The studies in this category can be divided into two sub-divisions according to the platforms based on which their methods are designed (e.g. configurable infrastructures and fixed infrastructures).

Configurable Infrastructure:

The studies in this sub-division employ configurable infrastructures. In order to achieve the desired design criteria, the studies dynamically decide the sizes of virtual machines, conduct server consolidations, or adjust the number of powered-on servers and their working frequencies.

- ***Dynamically Adjust Virtual Machine Sizes:***

  Chen et al. [26] proposed a new virtual machine sizing approach called "effective sizing". They considered various factors that will impact the aggregated

resource demand of a physical server, in which the virtual machine will be deployed. Based on the effective sizing approach, they further designed a set of polynomial time virtual machine allocation algorithms to minimize the data center operational overhead (e.g. virtual machine migration).

Liu et al. in [78][9] studied a method for multi-tier architecture that decides the workload distribution and computing capacity allocation to optimize the SLA-based profit a data center may achieve. Later, in [134], energy consumption was considered and an energy consumption control method was proposed to satisfy certain SLAs and energy constraints.

Similarly, Urgaonkar et al. [115] proposed a method that combines admission control and routing together with resource allocation to balance the trade-off between the throughputs of applications and the energy costs of the data centers.

Different from traditional virtual machine resource provisioning approaches, in which the size of a virtual machine is usually decided individually, Meng et al. [86] decided the sizes of multiple virtual machines together based on their aggregate capacity needs. By exploring the workload patterns in those virtual machines, their method employs statistical multiplexing to allocate required computing resources to the virtual machines as a whole.

- ***Server Consolidation:***

  Server consolidation has been a common approach to achieve high power/energy efficiency for data centers. The virtualization technology has enabled multiple virtual machines to be executed on the same physical server. Since a server's power consumption is not exactly proportional to its utilization, a server may consume a significant amount of power even when it is not fully

utilized. Therefore, consolidation helps increase the utilization of a physical server and thus minimize the number of activated physical servers.

For example, in [116], Verma et al. presented consolidation decisions that are made based on the correlations among different workloads. In [85], Mastroianni et al. devised two probability functions to model the effects of the virtual machine allocation and migration. By statistically analyzing the need of a virtual machine allocation or migration, they minimized the number of powered-on servers, and reduced the power consumptions in data centers.

Marty et al. [83] proposed a server consolidation method based on the utilization of memory, where most sharing occurs among virtual machines. Their method maximizes memory accesses for each virtual machine. Meanwhile, it minimizes interferences among different virtual machines by dynamically reassigning virtual machines and supporting content-based page sharing among the virtual machines.

Besides the above mentioned studies, other work also employs server consolidation to realize its objective [108][84][107][110][23].

- ***Dynamic Power Management and Dynamic Voltage and Frequency Scaling:***

  Extensive research has been conducted for dynamic resource provision by employing the dynamic power management (DPM) technology (e.g. [121][64][73][77]). These approaches dynamically modulated the availability of the underlying computing facilities (e.g. increase/decrease the number of powered-on physical servers) according to the workload predictions or in response to the workload changes.

Lin et al. [73] analytically formulated their optimal offline solution and developed the corresponding online algorithm to bound the number of powered-on servers with respect to certain delay constraints.

In [25], Chase et al. presented an architecture for resource management in a hosting center operating system. They adaptively provisioned server resources according to the offered workload. The efficiency of server clusters was improved by dynamically resizing the active server set in accordance with SLAs to respond to power supply disruptions or thermal events.

Another commonly used approach is to employ the dynamic voltage and frequency scaling (DVFS) in order to dynamically adjust the performance and the power dissipation of a server. For example, in [19], Beloglazov et al. proposed a two-layer optimization (e.g. global and local layers) to minimize the power consumption in data centers. The local layer monitors each virtual machine's utilization, and dynamically adjusts a physical server's working frequency through DVFS [119]. The global layer uses bin-packing methods to find a new destination for the virtual machine that has to be migrated.

Wang et al. [121] proposed a power minimization approach based on the traditional queuing model. Their method determines the appropriate number of powered-on servers and their running modes (i.e. execution speeds) to minimize power consumptions while statistically guaranteeing the QoS requirements.

Sarood et al. [104] proposed a model that takes cooling energy consumptions into consideration. By reasonably distributing workloads and employing DVFS, they successfully lowered the overall energy consumed by their cooling system while satisfying temperature constraints.

Chen et al. [27] developed a metric called "frequency gradient" to study the impact of changes in processor frequency on the end-to-end response times of multi-tier applications. Their work makes the end-to-end performance-aware DVFS strategy capable for multi-tier applications.

Wang et al. in [120] solved a problem of managing power consumption in multi-tier web clusters equipped with heterogeneous servers. Their method employs dynamic voltage scaling (DVS). By adjusting the number of powered-on servers and their working frequencies, they effectively reduced the energy consumption in their web clusters.

Fixed Infrastructure

The studies in this sub-devision are developed on fixed computing infrastructures. According to the scheduling technologies applied, they can be divided into three groups in general.

- ***Acceptance and Queue Length Control:***

  To control the requests execution subject to their end-to-end deadlines or other timeliness requirements, acceptance control and queue length control via random removal are commonly used [62][122][76][89][36], especially when a system is under overloading situations.

  Wang et al. [122] employed acceptance control to optimize a service provider's revenue with the least operational costs for multi-tier services in a virtualized environment. Liu et al. [76] designed an adaptive acceptance control method to optimize the performance for web applications by adjusting the queue lengths. Acceptance control and queue length control help service providers alleviate the system workload and thus guarantee timeliness requirements.

- ***Sub Deadline Assignment:***

  Sub deadline assignment is another popular approach for guaranteeing end-to-end deadlines. For example, Hong et al. [49] introduced a technique to assign a sub deadline for each service tier. The end-to-end deadline can be guaranteed if all sub services can meet their sub deadlines.

  Yu et al. [130][131][129] proposed a sub deadline assignment method by proportionally dividing the overall end-to-end deadline according to the best or worst execution times in the service tiers. A similar idea was employed by Mao et al. in [82].

- ***Request Mapping and Scheduling:***

  As applications usually consist of a series of intricate inter-dependent services, such applications are usually modeled as direct acyclic graphs (DAG). Request mapping and scheduling sometimes are employed together to optimize a system's performance in terms of short makespans or low processing costs.

  Kamthe et al. [58], proposed a scheduling approach that accurately estimated the earliest start time of each sub service in a DAG. Then, the request execution sequence and thus the makespan were optimized based on those earliest start times. Similarly, Tang et al. [112], developed a stochastic heterogeneous earliest finish time (SHEFT) scheduling approach to reduce the makespan of a DAG. Arabnejad et al. [8] proposed a novel list-based scheduling algorithm for heterogeneous computing systems to minimize a task's makespan.

2.2.3 Scheduling in Multiple Data Centers

Different from the scheduling for a single data center, in order to achieve the desired design criteria (e.g. low operational cost, power/energy consumption), the schedul-

ing methods for multiple data centers have to take the geographical information into consideration (e.g. different electricity prices or energy efficiencies of the different data centers). Most of the studies in this category focus on designing energy-efficient resource management approaches on configurable infrastructures. Specifically, they judiciously dispatch workload among different data center locations, and cautiously decide the number of powered-on servers and their working frequencies in the corresponding data center.

The research in [98][99] extended the work in [120] to a distributed data center architecture in a multi-electricity-market environment. Rao et al. modeled their problem as a constrained mixed-integer linear programming formula, and proposed an efficient approach to approximate the problem using a linear programming formulation.

Le et al. studied the advantages of using green energy (e.g. energy generated by wind or solar energy). These studies help replace the usage of "brown" energy (produced via carbon-intensive means) with "green energy" during a data center's operation in order to cut down the cost spent on energy consumptions. Frameworks for multi data center service was introduced in [65][66].

Garg et al. [37] proposed near-optimal scheduling policies that exploit heterogeneity across multiple data centers in order to improve energy utilization efficiencies for benefiting both service providers and our society. Several energy efficiency factors (e.g. energy cost, carbon emission rate, workload, and CPU power efficiency) that vary across the multiple data center locations were considered in their work.

Similarly, Chen et al. [28] developed a set of on-line request dispatching and resource allocation solutions for distributed hosting centers in order to minimize the energy consumption of the servers and thus the operational costs.

2.3 Summary

In this chapter, we first introduced traditional real-time scheduling. Then, we discussed the differences between the scheduling for embedded systems and the scheduling for cloud computing, followed by the discussion of related work.

In the following chapter, we present our contributions for single-tier delay-sensitive service scheduling design on a single server.

CHAPTER 3

SINGLE-TIER SERVICE SCHEDULING ON SINGLE SERVERS

For real-time services, timeliness is a major criterion of judging real-time service quality levels. Due to the high variability of the Internet, cloud applications are more of soft real-time in nature. Guaranteeing hard deadlines for real-time services would be neither practical nor necessary in most scenarios. In this regard, besides pre-assigned deadlines, some other timing information that is closely related to QoS become important metrics when processing delay-sensitive cloud requests, e.g. profit and cost. In this chapter, we present our research for single-tier service scheduling on a single server. A set of utility accrual based scheduling approaches for maximizing a service provider's profit are discussed.

3.1 Research Problem Introduction

To improve the real-time service performance, one approach is to employ the traditional UA approach [30, 79]. In [57], Jensen et al. first proposed to associate each task with a TUF, which indicates that the completion of a task will assign the system a certain value of utility, and the utility value varies with the time when the task is finished. Specifically, a TUF as shown in Figure 3.1(a) describes the value or utility accrued by a system at the time when a task is completed. Based on this model, there were extensive research results published on the topic of UA scheduling ([70, 71, 128, 126, 127]). While Jensen's definition of TUF allows the semantics of soft timing constraints to be more precisely specified, all these variations of UA-aware scheduling algorithms imply that the aborted tasks neither increase nor decrease the accrued value or utility of the system.

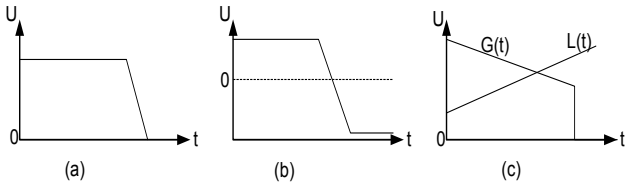


Figure 3.1: Time utility functions.

We believe that, to further improve the performance of real-time services over the Internet, it is important to not only measure the profit when completing a task in time, but also account for the penalty when a task is aborted or discarded. In addition, the time at which a real-time service is aborted is also important. First, the more service requests are discarded and the longer a client waits fruitlessly, the lower the quality of service client receives. As a result, service providers have to pay higher cost, either in the form of monetary compensation or losing future service requests from unsatisfied clients. Second, before a task is aborted or discarded, it needs to consume system resources, including network bandwidth, storage space, and processing power, and thus can directly or indirectly affect the system performance. This is especially true if we assume real-time applications may be dissected and migrated across an entire cloud infrastructure [24, 63]. Therefore, if a real-time task is deemed to miss its deadline with no positive semantic profit, a better choice should be one that can detect it and discard it as soon as possible.

A number of models [16, 29, 94, 53] were proposed to account for the penalty when a real-time service request is discarded or misses its deadline. For example, Bartal [16] et al. studied the on-line scheduling problem when penalties have to be paid for rejected real-time tasks. Chun et al. [29] and Irwin et al. [53] adopted an extended time utility function as shown in Figure 3.1(b). According to this model, a decay rate is associate with each real-time task, reflecting the increasing risk of completing the task late in the future. Therefore, when a real-time task is completed

late, it earns a negative utility, indicating a penalty rather than the profit. These models, however, do not account for different penalties when aborting a real-time task at different times.

In this research, we study the real-time service scheduling problem based on a task model similar to the one proposed by Yu et al. [132]. Specifically, a task is associated with two different TUFs, as showed in Figure 3.1(c), a profit TUF ($G(t)$) and a penalty TUF ($L(t)$). The system takes a profit (determined by its profit TUF) if the task completes by its deadline, and suffers a penalty (determined by its penalty TUF) if the task misses its deadline or is dropped before its completion. The penalty to abort a pending real-time service request can be the same or different from that of missing the deadline, which depends on the characteristics of the penalty TUF. Different from Yu's model, we use a novel method to calculate task's utility and use utility density to describe a task's priority. The "critical time" for each task is more strict, and we add an admission step when there is a new task arrives since congested ready queue will decrease the system's performance. It is a waste of system resources if tasks wait fruitlessly.

We conduct analysis on how to optimize the accrual utility when scheduling a set of aperiodic real-time service requests. We first assume that the service requests are scheduled in a non-preemptive manner. Two scheduling methods are presented. The first scheduling method is developed based on the concept of "opportunity cost" [20] from economics that can help evaluate the fulfillment of a real-time service request. The second method employs a more sophisticated but robust method to formulate the potential system profit by developing a speculated execution order for the ready tasks. We then extend our scheduling methods to deal with real-time services that may preempt each other. In addition to carefully choosing the ready task to run, our scheduling methods judiciously discard pending requests,

abort task executions, cautiously preempt current running tasks, and therefore can achieve better performance. Our experimental results also show that the proposed algorithms can significantly outperform the traditional scheduling approaches such as the Earliest Deadline First (EDF), the traditional UA scheduling algorithm i.e. the Generic Utility Scheduling(GUS) [70], the Risk/Reward algorithm [53], and a previous scheduling approach based on a similar model, i.e. the Profit Penalty aware scheduling (PP-aware scheduling) [132].

3.2 Preliminary

In this section, we first introduce the task and architecture models considered in this paper. We then use an example to motivate our research.

3.2.1 Task Model and System Architecture

In this paper, we consider a single sequence of randomly arrived real-time tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, with τ_i defined using the following parameters:

- $[B_i, W_i]$: The best case execution time B_i and the worst case execution time W_i of τ_i;
- D_i: The relative deadline of τ_i;
- $f_i(T)$: The probability density function for the execution time of τ_i;
- $G_i(t)$: The profit TUF, which represents the profit accrued when a task is completed at time t. We assume $G_i(t)$ is a non-increasing unimodal function before its deadline, i.e. $G_i(t_p) \geq G_i(t_q)$ if $t_p \leq t_q$, and $G_i(D_i) = 0$.
- $L_i(t)$: The penalty TUF, which represents the penalty suffered when a task is discarded or aborted at time t. We assume $L_i(t)$ is a non-decreasing unimodal

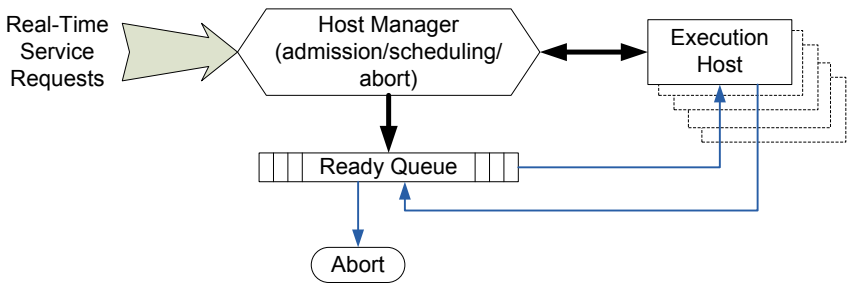


Figure 3.2: The architecture for the service provider.

function before its deadline, i.e. $L_i(t_p) \leq L_i(t_q)$ if $t_p \leq t_q$, and a task is immediately discarded once it missed its deadline.

Note that, even though the deadline of a task can be implicitly defined using appropriate profit and penalty TUFs, we opt to list the deadline explicitly as a parameter for ease of presentation. As shown above, a task is associated with both a profit function and a penalty function with function values varying with time. Therefore, while executing a task the system has a potential to gain profit, it also has a potential to encounter a penalty at a later time. The system performance is therefore evaluated by its total utility after penalty is deducted from profit.

We assume an architecture for the service provider depicted in Figure 3.2. Specifically, the service provider contains two computing components, i.e., the manager host and the execution host, that can work concurrently. The manager host is in charge of accepting, scheduling and aborting real-time service requests, and the execution host fulfill the selected service requests from the manager host. There may be one or more execution hosts for each service provider. We limit our research to one single execution host in this paper.

With the task and architecture model introduced as above, our problem can be formally formulated as follows.

Problem 3.2.1. *Given a task set* $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ *as described above, develop on-line scheduling methods such that the total accrued utility is maximized.*

3.2.2 A Motivation Example

The problem defined in Problem 6.2.1 is NP-hard since a simpler version of this problem, i.e. the total weighted completion time scheduling problem [13], is shown to be NP-hard. To show that the commonly used scheduling policy such as the EDF or the traditional utility accrual approach such as the GUS [70] become ineffective to address this problem, consider the example shown in Figure 3.3.

Assume that two real-time service requests arrive at the same time ($t = 0$) with their characteristics shown in Figure 3.3. We assume that the actual processor time of each request is evenly distributed between the interval of its best case and worst case execution time. To make the example more concrete, we assume that the actual processing times for these two requests are 50 and 60, respectively.

When EDF is applied, τ_1 has a higher priority than τ_2 and is executed first. It completes at $t = 50$ with profit of $G_1(50) = 180 - 2 \times 50 = 80$. Then τ_2 starts its execution. At $t = 100$, it misses its deadline and will incur more penalty if its execution continues. Therefore, the execution of τ_2 is discarded at $t = 100$ with penalty of $L_2(100) = 2 \times 100 = 200$. The total utility to process these two requests is therefore $80 - 200 = -120$.

The GUS algorithm chooses the task with the largest expected profit density to execute first. Under our task model, the expected profit of τ_1 and τ_2, i.e. $\overline{G}(\tau_1)$ and

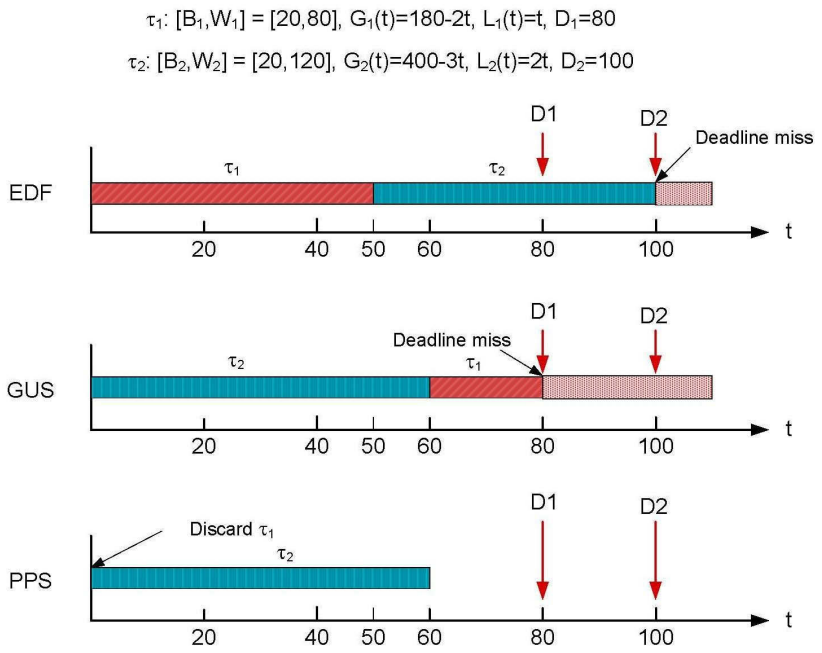


Figure 3.3: Three different schedules for two real-time tasks τ_1 and τ_2 arriving at the same time $t = 0$.

$\overline{G}(\tau_2)$, can be calculated as:

$$\begin{aligned} \overline{G}(\tau_1) &= \int_{20}^{80} (180 - 2t) \times \frac{1}{80 - 20} dt = 80 \\ \overline{G}(\tau_2) &= \int_{20}^{120} (400 - 3t) \times \frac{1}{120 - 20} dt = 190 \end{aligned}$$

At $t = 0$, we have no knowledge of the actual execution time of τ_1 and τ_2, a reasonable estimate would be the one using their expected values, i.e. 50 and 70, respectively. As a result, τ_2 is chosen to execute first since its expected profit density (expected profit divided by expected execution time) 190/70 is higher than that of τ_1, i.e. 80/50. It completes at $t = 60$ with profit of $G_2(60) = 400 - 3 \times 60 = 220$. Then τ_1 starts its execution. At $t = 80$, it misses its deadline and is aborted to prevent even higher loss. The total utility to process these two requests is therefore $220 - 80 = 140$.

An astute reader may immediately point out that, after τ_2 completes at $t = 60$, it is less likely that τ_1 can complete by its deadline, given that its best case execution time is 20. Therefore, τ_1 should be immediately aborted at $t = 60$ with a total utility profit of $220 - 60 = 160$. Note that, after τ_2 is selected to execute first, its expected execution time would be 70. Given the expected execution time of τ_1 being 50, it is more likely that τ_1 will miss its deadline. Therefore, a better scheduling decision would discard it at $t = 0$ with total profit of 220 in this case, as the third schedule shown in Figure 3.3.

In our example, we can see that the EDF has the worst performance since it makes scheduling decisions solely based on tasks' deadlines. The traditional utility accrual scheduling method takes the individual value function into consideration and therefore can achieve better performance. The problem, however, is that the traditional utility accrual scheduling approaches (such as GUS) fail to take the abortion or discard penalty and the timing for the abortion or discard penalty into consideration. Clearly, how to select the appropriate task to run so as to maximize the profit and how to discard real-time tasks as soon as possible in overloaded situations in order to control the penalty are vital for our research problem.

3.3 Non-Preemptive Approach

In this section, we present our on-line non-preemptive scheduling solutions to address the problem defined in the previous section. Since the execution of a task may gain positive profit or suffer penalty and thus degrade the overall computing performance, judicious decisions must be made with regard to executing a task, discarding or aborting a task, and when to discard or abort a task. In what follows, we present

two metrics to measure the expected utility when executing a real-time task, and based on which, we develop two scheduling algorithms.

3.3.1 The Opportunity Cost Based Utility Metric

Our first utility metric is built upon the concept of *opportunity cost* [20] in economics. In economics, the *opportunity cost* refers to the value associated with the next best available choice that one has to give up after making a choice. When scheduling a set of real-time tasks at $t = T$, let expected utility of running τ_i alone be $\overline{U_i}(T)$ and its opportunity cost be $\overline{OC_i}(T)$. Then we can conveniently formulate the expected utility $\widetilde{U}(\tau_i, T)$ to run τ_i at $t = T$ as

$$\widetilde{U}(\tau_i, T) = \overline{U_i}(T) - \overline{OC_i}(T). \tag{3.1}$$

The problem becomes how to calculate $\overline{U_i}(T)$ and $\overline{OC_i}(T)$.

Since the task execution time is not known a prior, we do not know if executing the task will lead to positive profit or loss. Given its probabilistic distribution, we can determine the expected profit and loss statistically. Given a task τ_i with arrival time of r_i, let its predicted starting time be T. Then the expected profit ($\overline{G_i}(T)$) to execute τ_i can be represented as

$$\begin{aligned} \overline{G_i}(T) &= \int_0^{\infty} G_i(t + (T - r_i)) f_i(t | t + T < D) dt \\ &= \int_{B_i}^{D_i} G_i(t + (T - r_i)) f_i(t) dt \end{aligned} \tag{3.2}$$

Similarly, the expected loss ($\overline{L_i}(T)$) to execute τ_i can be represented as

$$\begin{aligned} \overline{L_i}(T) &= L_i(D) P(t + T > D) \\ &= L_i(D) \int_{D_i - (T - r_i)}^{W_i} f_i(t) dt. \end{aligned} \tag{3.3}$$

Therefore, the expected utility $\overline{U_i}(T)$ can be represented as

$$\overline{U_i}(T) = \overline{G_i}(T) - \overline{L_i}(T). \tag{3.4}$$

When $\overline{U_i}(T) > 0$, it means that the probability to obtain positive profit is no smaller than that to incur a loss if we choose to execute τ_i at $t = T$. Since $\overline{G_i}(T)$ is a monotonic decreasing function of T, and $\overline{L_i}(T)$ is a monotonic increasing function of T, $\overline{U_i}(T)$ must be a monotonic decreasing function of T.

Note that, even though two tasks may have the same expected utility, they may have different expected execution times. We define a parameter ρ_i to capture the *expected utility density* for task τ_i as follows:

$$\overline{\rho}_i(T) = \overline{U_i}(T)/\overline{C_i}. \tag{3.5}$$

where $\overline{C_i}$ is the expected execution time of task τ_i. There exists a t_0 such that

$$\overline{\rho}_i(t_0) = 0. \tag{3.6}$$

The time $t = t_0$ is called the *critical point*. Apparently, when $t > t_0$, it is more likely that it will incur a loss rather than a profit if we choose to execute τ_i. We can further relax Equation (3.6) by imposing a threshold (δ), i.e.

$$\overline{\rho}_i(t_0) \geq \delta. \tag{3.7}$$

We call δ as the *utility density threshold*.

We next introduce how to formulate the opportunity cost when choosing to run task τ_i at $t = T$. The original concept of "opportunity cost" is the value for the next best available choice. It is hard to identify the "next best choice" since the exact reason we need the opportunity cost is to set up the preference order when choosing tasks to run. In our metric, the opportunity cost is calculated as the decay of expected utilities by other tasks. Specifically, let the expected utility of τ_j at

$t = T$ be $\overline{U_j}$. Then if we choose τ_i to execute at $t = T$ and after its completion, the expected utility of τ_j is reduced to $\overline{U_j}(T + \overline{C_i})$, where $\overline{C_i}$ is the expected execution time of τ_i. Provided we can remove the task timely when its expected utility is less than zero, we thus define the opportunity cost to run τ_i at $t = T$, i.e. $\overline{OC_i}(T)$, as

$$\overline{OC_i}(T) = \frac{1}{n-1} \sum_{j=1, j \neq i}^{n} max((\overline{U_j} - \overline{U_j}(T + \overline{C_i})), 0). \tag{3.8}$$

With both $\overline{U_i}(T)$ and $\overline{OC_i}(T)$ formulated, we are now ready to introduce our scheduling algorithm. Our non-preemptive scheduling algorithm works at scheduling points that include: the arrival of a new task, the completion of the current task, and the critical point of the current task. The detailed algorithm is described in Algorithm 1.

Algorithm 1: THE SCHEDULING ALGORITHM BASED ON OPPORTUNITY COST

Input : Let $\{\tau_1, \tau_2, ..., \tau_k\}$ be the accepted tasks in the ready queue, and let $\overline{C_i}$ be the expected execution time of τ_i. Let current time be t and let τ_0 be the task currently being executed, expected execution time of τ_0 is $\overline{C_0}$. Let the expected utility density threshold be δ.

1 **if** *A new task, i.e.* τ_p *arrives* **then**
2 Accept τ_p if $\overline{\rho}_p(\overline{C}_0) > \delta$;
3 Reject τ_p if $\overline{\rho}_p(\overline{C}_0) \leq \delta$;
4 Remove τ_j in the ready queue if $\overline{\rho}_j(\overline{C}_0) \leq \delta$;
5 **if** τ_0 *is completed* **then**
6 Choose τ_i with the largest system utility density, i.e. $\widetilde{\rho}_i(t) = max_k \widetilde{\rho}_k(t)$;
7 Remove τ_j in the ready queue if $\overline{\rho}_j(\overline{C}_i) < \delta$;
8 **if** t = *the critical time of* τ_0 **then**
9 Abort τ_0 immediately;
10 Choose τ_i with the largest system utility density, i.e. $\widetilde{\rho}_i(t) = max_k \widetilde{\rho}_k(t)$;
11 Remove τ_j in the ready queue if $\overline{\rho}_j(\overline{C}_i) < \delta$;

When a new job arrives, its expected utility density is calculated based on Equations (3.1), (4.5), and (3.8). If its expected utility density is larger than the pre-set

threshold, it is accepted and is rejected otherwise. When the current running task completes, the task in the ready queue with the highest expected system utility density is chosen to be executed. When the time reaches the critical point of the current running task, it implies that it will mostly likely incur utility density less than the threshold and is thus worthless of continue execution. In that case, the task is immediately discarded, and a new task will be chosen to execute. At every scheduling point, the expected utility density of the tasks in the ready queue are checked. Since the expected utility density decreases monotonically with time, the task with expected utility density less than the threshold is aborted. The complexity of Algorithm 1 comes from the calculation of the expected system utility values for the task set, with the complexity of $O(n^2)$ where n is the number of tasks in the ready queue.

3.3.2 The Speculation Based Utility Metric

From Equation (4.3), (4.4) and (4.5), we can clearly see that the expected utility of running a task depends heavily on variable T, i.e. the time when the task can start. If we can know the execution order and thus the expected starting time for tasks in the ready queue, we will be able to quantify the expected utility density of each task more accurately. In this section, we develop our second utility metric based on a speculated execution order of the tasks in the ready queue.

The general idea to generate the speculated execution order is as follows. We first calculate the expected utility density for each task in the ready queue based on the expected finishing time of the current running task. Then the task with the largest one is assumed to be the first task that will be executed after the current task is finished. Based on this assumption, we then calculate the expected utility

Algorithm 2: Generating the speculated execution order and the expected utility for task in the ready queue

Input : Let $\Gamma = \{\tau_1, \tau_2, ..., \tau_k\}$ be the accepted tasks in the ready queue, and let r_i, $\overline{C_i}$ represent the arrival time and expected execution time of τ_i. Let the current time be t

Output: The new list $\Gamma' = \{\tau'_1, \tau'_2, ..., \tau'_k\}$ with the speculated execution order and their corresponding expected utility density $\widehat{\rho}'_j$ for $\tau'_j, 1 \leq j \leq k$.

if *A task* τ_0 *is being executed* **then**
    $T = r_0 + \overline{C_0}$;
else
    $T = t$;
while Γ *is not empty* **do**
    for *Each task* i *in* Γ **do**
        Calculate $\overline{\rho}_i(T)$ based on Equations (4.3), (4.4), (4.5), and (3.5);
    Select τ_j with the highest $\overline{\rho}_j(T)$;
    Add τ_j to the end of Γ';
    $\widehat{\rho}_j = \overline{\rho_j}(T)$;
    $T = T + \overline{C_j}$;
    Remove τ_j from Γ;

densities for the rest of the tasks in the ready queue and select the next task. This process continues until all tasks in the ready queue are put in the order. While completed, we essentially generate a speculated execution order for the tasks in the ready queue and, at the same time, calculate the corresponding expected utility density for each task. The detailed algorithm is described in Algorithm 2.

The scheduling algorithm based on our speculated utility metric is very similar to Algorithm 1 and is thus omitted. The only difference is that the speculation expected utility, rather than the opportunity cost based utility, for each task in the ready queue is calculated at each scheduling point, including the arrival of a new task, the completion of the current task, and the critical point of the current task.

The complexity of the scheduling algorithm mainly comes from Algorithm 2. It is not difficult to see that the complexity of Algorithm 2 is $O(n^2)$ with n the number of tasks in the ready queue.

3.4 Preemptive Approaches

In the previous section, we introduce two methods to quantify the potential system utility when scheduling a set of real-time requests *nonpreemptively*. Since a preemptive real-time scheduling technique tends to be more responsive for a higher priority request, and can achieve higher schedulability and throughput than its non-preemptive counterpart, we are interested in studying how to schedule a real-time task set preemptively to maximize the total accrued system utility.

When employing the preemptive scheduling method to schedule real-time tasks with the goal of maximizing the accrued utility, a critical issue is to determine when the preemption should occur. An intuitive approach is to define the priority of a task based on its expected utility density (Equation (3.5)). Nevertheless, such an uncon-

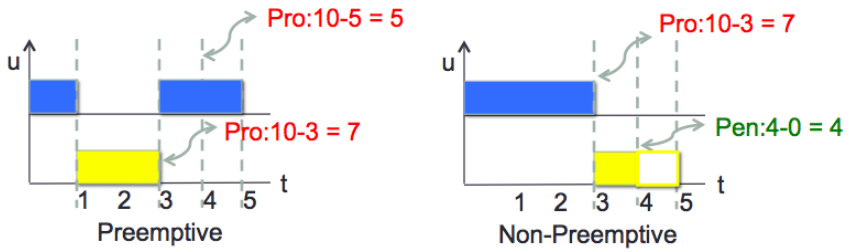
strained preemptive scheduling may or may not improve the system performance, in terms of accrued system utility, when compared to a non-preemptive one.

Consider the two examples in Figure 3.4. Figure 3.4(a) shows two tasks scheduled both preemptively (based on the expected utility density) and non-preemptively. The parameters for both tasks are listed in the figure. In preemptive method, Task τ_1 arrives and starts its execution at arrival time $t = 0$. At time $t = 1$ task τ_2 arrives. Note that, at $t = 1$, we have $\rho_1(t) = 1.3$ and $\rho_2(t) = 1.6$. Therefore, τ_2 comes with a higher expected utility density and preempts τ_1. Task τ_1 continues its execution after task τ_2 completes. The total utility in this method is 12. In the corresponding non-preemptive method, task τ_2 misses its deadline and the total utility in this method is 3. This example shows that by processing the higher *"priority"* job first, the preemption helps increase the total utility of the system.

Now let us consider the example in Figure 3.4(b). For the two tasks in Figure 3.4(b), at time $t = 1$, we have $\rho_1(t) = 1.6$ and $\rho_2(t) = 2.3$. Therefore τ_2 has a higher priority than τ_1. When these two tasks are scheduled in the preemptive manner, task τ_1 misses its deadline and the total utility is 3. Both two tasks can meet their deadlines when they are scheduled in the non-preemptive manner with a total utility of 12.

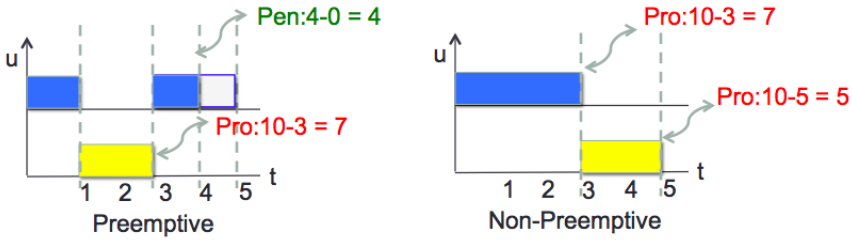
This example illustrates that unconstrained preemption does not always help improve the accrued utility. Note that, since the profit and penalty TUFs of each task vary with time, its "priority" also varies with time. In this case, all tasks in the ready queue need to be checked for priority at every time instance. Hence a perfect preemptive scheduling would be impractical due to its prohibit computational cost, even if it is theoretically possible. In addition, a large number of unconstrained preemptions disrupts task executions, makes them less likely to complete before their deadlines, and leaves alone the large overhead coming with the preemptions. Our

	Arrival	Exec	Best	Worst	G	L	D
Task 1	0	3	2	7	$10-t$	$t-0$	6
Task 2	1	2	1	5	$10-t$	$t-1$	4



(a) Preemptive scheduling is better than the non-preemptive scheduling

	Arrival	Exec	Best	Worst	G	L	D
Task 1	0	3	2	5	$10-t$	$t-0$	4
Task 2	1	2	1	5	$10-t$	$t-1$	6



(b) Non-preemptive scheduling is better than the preemptive scheduling

Figure 3.4: Preemptive vs. non-preemptive scheduling two real-time requests to maximize the accrued system utility.

empirical studies also showed that unconstrained preemptive scheduling can potentially degrade the performance than the corresponding non-preemptive scheduling. To this end, we want to limit the scenarios of when the preemption can occur to improve the performance of the preemptive scheduling.

To constrain the preemptions, we first limit the time instances that at when preemptions can occur. Instead of letting a higher priority task always preempt a lower priority task, we allow that such a preemption can only happen when a new task comes or at a regular checking point, which we call *preemption checking point*. Let the last preemption occurs at time $t = T_0$. A task can be preempted at $t = T$ only if new tasks arrive at $t = T$ or

$$(T - T_0) \; mod \; L_{int} = 0, \tag{3.9}$$

where L_{int} is the length of the *preemption checking point* interval.

At a *preemption checking point*, the higher priority task does not necessarily always preempt the one with lower priority, if the potential gain to execute the high priority task is not significantly higher than the gain achieved by continuously executing the current running task. We define a parameter called *preemption threshold* for this purpose. Let the current running task τ_0's conditional expected accrued utility density be $\hat{\rho}_0(\tau_0, t)$ at time t, and preempting task τ_p's expected accrued utility density be $\overline{\rho_p}$. Task τ_p preempts τ_0 only when the following equation is satisfied

$$\overline{\rho_p}(\tau_p) - \widehat{\rho_0}(\tau_0, t) > \zeta. \tag{3.10}$$

where ζ is the *preemption threshold*.

To further constrain preemptions, we do not allow the current task be preempted if it can complete by its deadline even it requires its worst case execution time. Preempting such tasks can delay the completion of these tasks, and potentially

turn the profit into penalty if these tasks miss their deadlines. This constraint is illustrated by Equation (3.11).

$$S_{\tau_0} + WE_{\tau_0} \leq D_{\tau_0}. \tag{3.11}$$

where S_{τ_0} is the starting time of current running task τ_0. WE_{τ_0} means the worst case execution time of τ_0. D_{τ_0} represents τ_0's deadline.

We summarize our preemption rules and present the preemptive scheduling algorithm in Algorithm 3.

Algorithm 3: PREEMPTION CHECKING

Input : Let τ_0 be the task currently being executed, and τ_p be the task wants to preempt τ_0, current time be t, $\widehat{\rho_0}(\tau_0, t)$ be the conditional expected utility density of τ_0 at time t, $\overline{\rho_p}(\overline{C_p})$ be the expected utility density of τ_p, $\overline{C_p}$ and $\overline{C_0}$ are the expected execution time of τ_p and τ_0, respectively;

```
When a new task arrives or it is the preemption checking point;
if ρ̄_p(C̄_p) − ρ̂_0(τ_0, t) > ζ then
    Check what is τ_0's worst case finish time;
    if S_τ0 + WE_τ0 ≤ D_τ0 then
        Preemption not allowed;
    else
        Preemption allowed;
```

From Algorithm 3, when a preemption checking point is reached or when there is a new task arrives, scheduler first compares the preempting task's expected utility density $\overline{\rho_p}(\overline{C_p})$ with the current running task's conditional expected utility density $\widehat{\rho_0}(\tau_0, t)$. If the preempting task's expected utility density exceeds the current running task's conditional expected utility density by a *preemption threshold*, then the scheduler further checks if the current running task can completes its execution in its worst case or not. If the current running task can be completed even in its worst case, no preemption is allowed in order to protect the current running task, since

this current running task will absolutely contribute positive utility to the system. Otherwise, the preemption may postpone the current running task and result in a penalty because of missing its deadline.

3.5 Experiments

In this section, we use experiments to investigate the performance of our proposed algorithms. The following six representative scheduling approaches were implemented and compared:

- **EDF**: The execution order of tasks are determined based on the EDF scheduling policy;
- **GUS** [70]: The execution order of tasks is determined by the expected utility density, or the accrued utility per unit time;
- **PP**: This is a previous approach developed based on a metric called *Risk Factor* [132]. It adopts similar system models as those used in this paper;
- **RR**: The Risk/Reward approach described in [53]. This is a utility accrual approach that allows the utility value to be negative (e.g. similar to Figure 3.1(b));
- **PPOC**: This is the scheduling approach (i.e. Algorithm 1) built upon the utility metric that is developed based on the opportunity cost;
- **PPS**: This is the scheduling approach built upon the speculated utility based metric as discussed in section 3.3.2.

3.5.1 Experiment Setup

The test cases in our experiments were randomly generated. Specifically, each task $\tau = ([B, W], f(T), G(t), L(t), D)$ was randomly generated as below:

- B, W, and D were randomly generated such that they are uniformly distributed within interval of $[1, 10]$, $[30, 50]$, and $[40, 50]$, respectively;
- The execution time of a task is assumed to be evenly distributed between interval of [B,W], i.e. $f(t) = \frac{1}{W-B}$
- G, L were assumed to be linear functions, i.e. $G(t) = a_g(-t + D)$ in the range of $[0, D]$ and $L(t) = a_l t$. The gradient for $G(t)$ and $L(t)$, i.e. a_g and a_l were randomly picked from the interval of $[4, 10]$ and $[1, 5]$, respectively;
- Task release times follow the Poisson distribution with $\mu = 1$;
- Preemption check interval length L_{int} is set to be 1;
- Preemption threshold ζ is set to be 0;
- The utility density threshold δ is set to 0.

We conducted several different groups of experiments to study and compare the performance of different approaches under different conditions. The results are reported as follows.

3.5.2 Overall Performance Comparison

We first constructed 5 groups of experiments to study the overall performance of our proposed non-preemptive scheduling algorithms. Each group has 1000 task sets, each of which consists of 20 tasks. The six different non-preemptive scheduling algorithms were applied to the same task sets. The overall utility, the total profit,

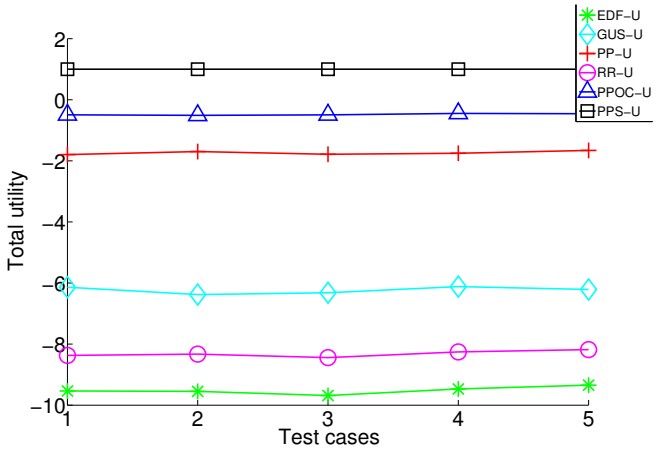
and the total penalty by each scheduling approach were collected and plotted in Figure 3.5(a), Figure 3.5(b), and Figure 3.5(c), respectively. For ease of presentation, the experimental results are normalized to that by PPS.

Figure 3.5(a) clearly shows that both PPOC and PPS can significantly outperform the other scheduling approaches. It is not surprising that, from Figure 3.5(c), we can see the penalty conscious approaches, i.e. PP, PPOC, and PPS, are more effective in control the penalty than the other three, i.e. EDF, GUS and RR. PPOC and PPS are particular effective in penalty control. It is interesting to note from Figure 3.5(b) and 3.5(c) that, while the profits obtained by PPOC and PPS are comparable or even inferior to the other approaches, the penalties are dramatically decreased. This is because tasks that would potentially lead to high penalty are declined or discarded at early stages of their execution. As a result, the overall utilities are significantly higher than other approaches.

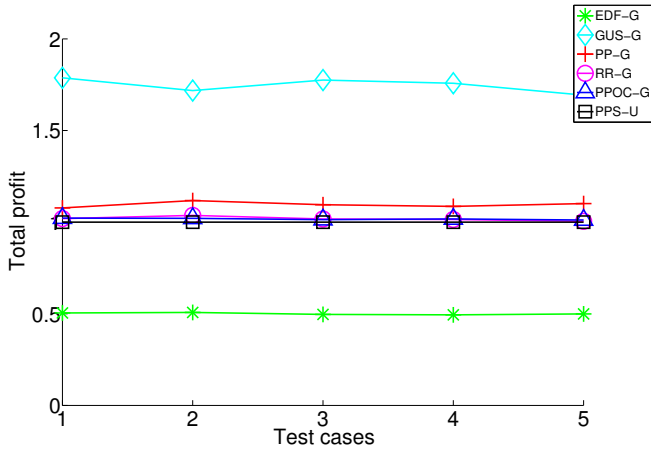
When comparing PPOC non-preemptive and PPS non-preemptive, we can see from Figure 3.5(a) that PPS is slightly better than PPOC. We can tell that the speculation based utility metric predominant the opportunity cost metric in the control of penalty. The speculation order plays the major role in predicting the high risk of penalty.

3.5.3 Arrival Burst Impacts

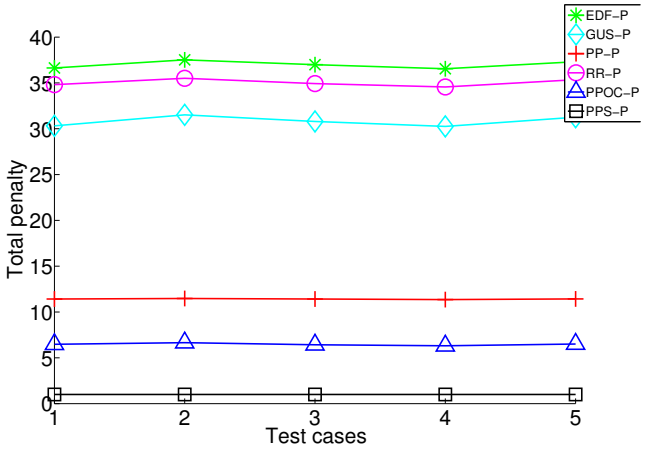
We next studied the performance of our non-preemptive scheduling methods under different burst conditions. In this experiment, we set the number of tasks to 20 and varied the expected number of occurrences within an unit interval, μ, from 1 to 5. By changing μ, we essentially changed the interval length between task arrivals. Figure 3.6 shows the results of the 1000 task sets' total utility with different values



(a) Total utility



(b) Total profit



(c) Total penalty

Figure 3.5: The comparison of total utility, profit, and penalty by different non-preemptive scheduling approaches.

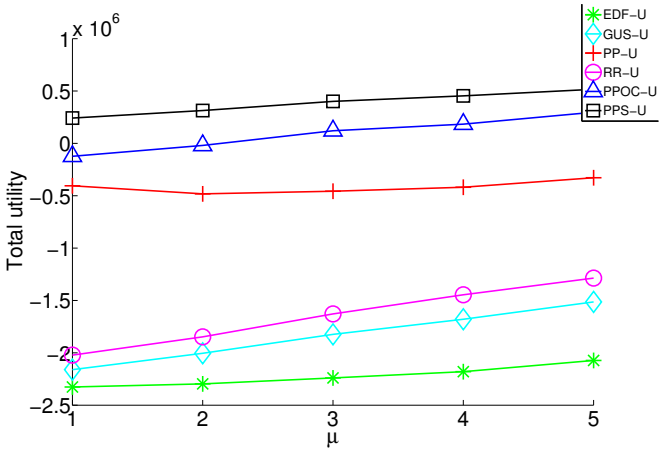


Figure 3.6: The total utility with different μ from non-preemptive scheduling algorithms.

of μ achieved by the non-preemptive scheduling algorithms. From Figure 3.6, we can see that all the methods have a better performance as μ increases, and PPS and PPOC significantly outperform the other approaches.

When μ increases from 1 to 5, the number of task that come within the same length of interval decreases, so the ready queue becomes less crowded and the overall workload reduces. The reduction in workload also helps lower down the deadline miss rate. More tasks can contribute positive profits instead of negative penalties to the system. Therefore the total accrued utility is improved.

3.5.4 Utility Density Threshold Effect

We further studied the impacts of the utility density threshold δ on the scheduling performance. As indicated in Section 3.3, the threshold δ plays an important role in task admission, abortion, and execution. The larger the threshold, the smaller the number of tasks can be accepted and executed, and the smaller the penalty the system will suffer. To study this impact, we conducted another set of experiments. We generated task sets as before, but changed the threshold from -30 to 30, with

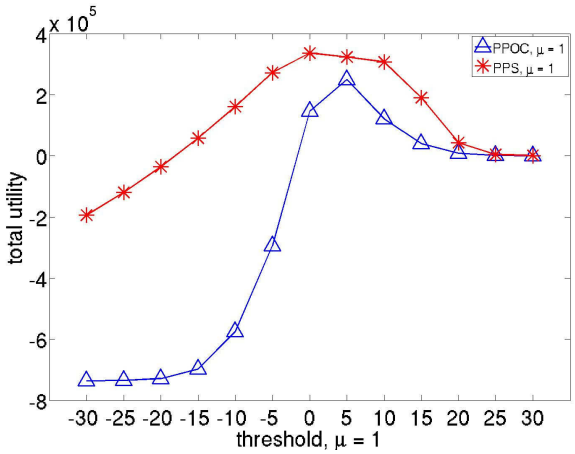


Figure 3.7: The total utility varies with the threshold.

an interval of 5. The total utilities were collected and shown in Figure 3.7. It shows the effects on the 1000 task sets' total utilities at various threshold values.

It is interesting to see that the highest utility does not always occur at the point when the threshold equals zero. With the help from the figure, we can tell that the highest utility seldom occurs at the point with the lowest or the highest threshold value. The lower the threshold, the more tasks can be accepted to the system and get executed. This helps to improve the value of potential total profit. However, having more tasks accepted into a ready queue may potentially increase the potential penalty cost as many jobs can not meet their deadlines. On the contrary, using a higher threshold helps control the potential penalty but may limit the potential total profit that can be obtained. As a result, the total utility is a tradeoff between the two as shown in Figure 3.7. From Figure 3.7 we can see the significant impact that the different threshold values may have on the overall performance. How to choose an appropriate threshold value for a specific task set to strike the balance between the profit and penalty and hence achieve the optimal accrued utility is an interesting problem and needs further study.

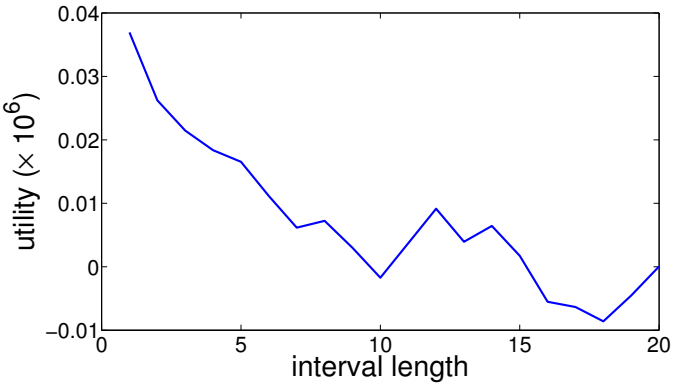


Figure 3.8: Preemption checking interval effect with $\lambda = 1$, $tasknumber = 20$ each group, and $\zeta = 0$.

3.5.5 Effects of Preemption Checking Interval Length L_{int} and Preemption Threshold ζ

In order to design a proper preemptive approach, we studied the preemption effects that come from variables L_{int} and ζ. We want to avoid aggressive preemptions. In Algorithm 3, when there is a high priority task that wants to preempt the current running task, our scheduler first tries to protect the current running task and guarantee the current running task to finish execution without being preempted. The first constraint we added on preemptions is the preemption checking interval. In Figure 3.8, the result shows the effect of L_{int}. Even though there are bumps in the figure, it demonstrates a major trend that the smaller the preemption checking interval, the higher the system utility.

A preemption threshold ζ is another preemption constraint. Its effect is reflected by Figure 3.9. An optimal preemption threshold for a special task set can be hard to find. As shown in this figure, similar to the utility density threshold (δ) effect, the optimal value is seldom achieved at the two extremes. For this particular data set we tested, the best preemption threshold value ζ is around 2

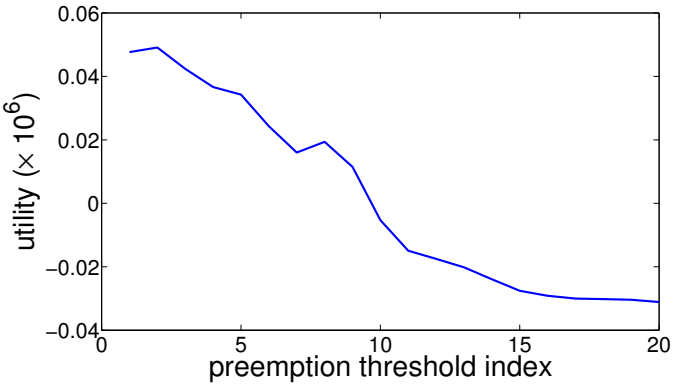
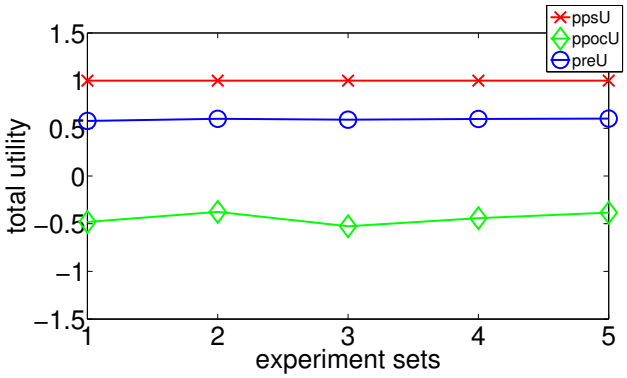


Figure 3.9: Preemption threshold effect with $\lambda = 1$, task number = 20, and $L_{int} = 1$.

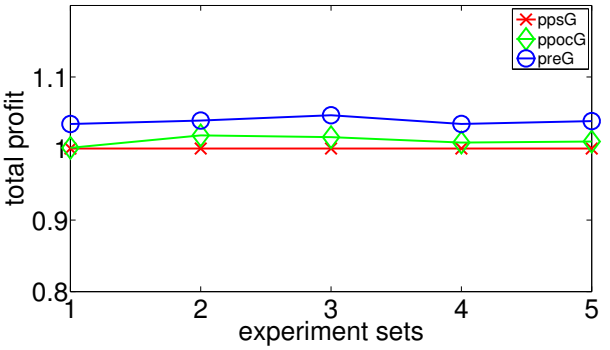
High preemption number does not mean better performance. Besides overheads generated by preemptions, potential penalties caused by preemptions may also be large. A set of carefully designed preemption rules can significantly improve the preemption performance. Results are shown in the next subsection.

3.5.6 Preemption vs. Non-Preemption

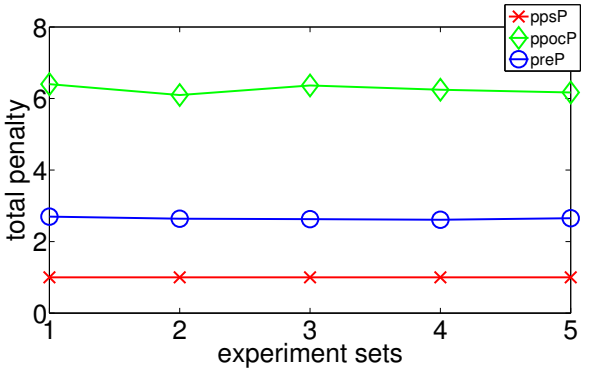
Finally we compare our nonpreemptive and preemptive scheduling approaches. Figure 3.10 shows comparisons in details between non-preemptive and preemptive scheduling approaches with the same task sets. It illustrates that PPS has the highest system utility, followed by the preemptive approach, then PPOC obtains the lowest system utility among these three approaches. Even though from Figure 3.10(b) we can tell that preemptive approach achieves the highest profit among them, it does not have a good control on penalty as PPS does. This results in a lower system utility in the preemptive approach than that in PPS. Nevertheless, Figure 3.10(b) illustrates the value of constrained preemptions for increasing system accrued utility, since our preemptive scheduler always selects high priority tasks to run at proper time.



(a) Comparison of total utility.



(b) Comparison of total profit.



(c) Comparison of total penalty.

Figure 3.10: Comparison between PPS non-preemptive and preemptive scheduling under the burstiness effect.

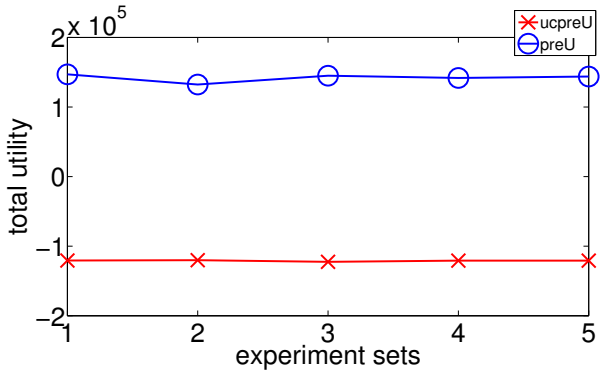


Figure 3.11: Comparison between constrained preemption approach and unconstrained preemption approach.

Figure 3.11 highlights the importance of preemption constraints. Some improper preemption instances postpone the running task's natural execution, in which a task may meets its deadline constraint without being preempted. In addition, it is hard to predict the future condition of the postponed task. Preemptions may help maximize the accrued system utility since the scheduler always runs high priority tasks first, whereas whether to allow preemptions happen needs prudential measures. The observation from Figure 3.11 is that by applying constraints on preemptions, we successfully improve the performance of the preemptive scheduling approach.

3.6 Conclusions

The popularity of Internet has grown enormously, which has presented a great opportunity for providing real-time services over Internet. Considering the tremendously large scale of the internet infrastructure, it is necessary that not only the profit but also the cost of real-time task executions should be taken into consideration during the resource management process. Our experimental results clearly show that the traditional utility accrued approaches become ineffective.

In this paper, we first present two novel non-preemptive utility accrued scheduling approaches upon a metric developed according to the *opportunity cost* concept and a speculation-based metric for expected utility, respectively. Then, a constrained preemptive approach is proposed. Our scheduling algorithms carefully choose highly profitable tasks to execute, and aggressively remove tasks that potentially lead to large penalty. Our extensive experimental results clearly show that our proposed algorithms can significantly outperform the traditional EDF approach, the traditional utility accrued approaches, and an earlier heuristic approach based on a similar profit and penalty aware task model.

In order to accommodate the distributed characteristic of cloud computing, we extend our research in next chapter to study service scheduling for distributed data centers in a multi-electricity-market environment.

CHAPTER 4

SINGLE-TIER SERVICE SCHEDULING IN A DISTRIBUTED ENVIRONMENT

In the previous chapter, we studied the single-tier service request scheduling on a single server. We extend our study to distributed environment in this chapter. We first discuss a TUF based approach, then in order to handle the large amount of service request in cloud environment, we extend our solution to a queuing theory based one.

4.1 Non Queuing Theory Based Approach

In North America market, the electricity prices vary a lot based on the different power generating technology, delivery method and coverage. Even though in the same region, the prices may vary significantly during a day. In order to save the construction and electricity investment, many service provider companies, which provides storage, processing, or other services, build their data centers in different locations where there is less population and near to the power plant. Take Google as an example, it has data centers in Portland, Houston, and Atlanta, etc. This helps to ensure the power supply and at the same time bring down their energy consumption dollar cost.

Several researches have been conducted for guaranteeing QoS with energy consumption minimization on data centers. In [98], the authors modeled the service system based on the Queueing Theory. By optimizing the task allocation and the number of powered on servers in each data center, their method successfully reduces the electricity cost in a multi-electricity-market. The authors in [120] presented a Dynamic Voltage Scaling (DVS) based power control mechanism for heteroge-

neous distributed systems. Their approach properly sets the number of servers that should be powered on and the frequency level at which the working servers should run. Both works model the delay constraint described based on the classic queueing model. While delay constraint can be used to model QoS requirement, it is not as effective for services requests that are sensitive to "timeliness". To this end, TUFs can be used to more accurately describe the timing related QoS requirements.

In this research, we propose a new on-line real-time service allocation and scheduling algorithm for distributed data centers in a multi-electricity-market environment (as shown in Figure 4.1). Service providers gain profit by satisfying service requests to the level identified based on a certain service level agreement (SLA). At the same time, service providers need to pay the cost for energy consumed by transferring and processing requests. We use a model similar to that in [132] to model the processing cost at data centers and data transferring cost in Internet service provider's networks. Based on this model, we developed a scheduling and allocation method to balance the trade off between the system's retained profit and system performance. The experimental results show that our method can effectively reduce the power consumption and, at the same time, increase the system's retained profit. It is worth mentioning that our work is suitable for both homogeneous and heterogeneous data centers and can be easily extended to accommodate more complex service requests.

4.1.1 System Model

Our system architecture consists of a front end server, a task allocator, and several data centers at different locations. Figure. 4.2 shows the overall system architecture. The front end server receives the service requests from the Internet. The task allo-

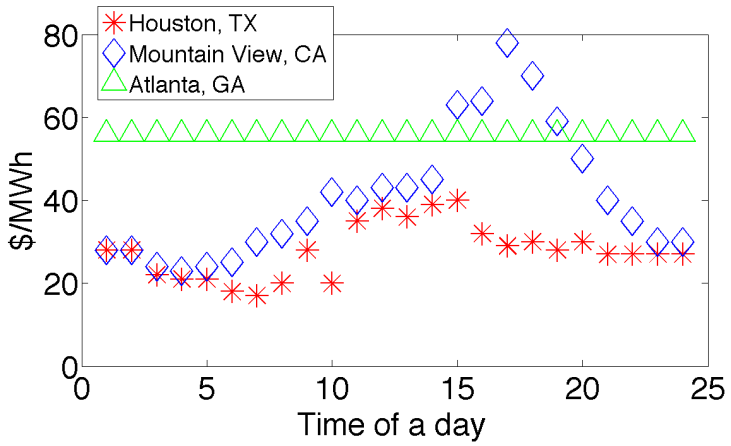


Figure 4.1: Hourly average electricity prices at different locations in a day[120].

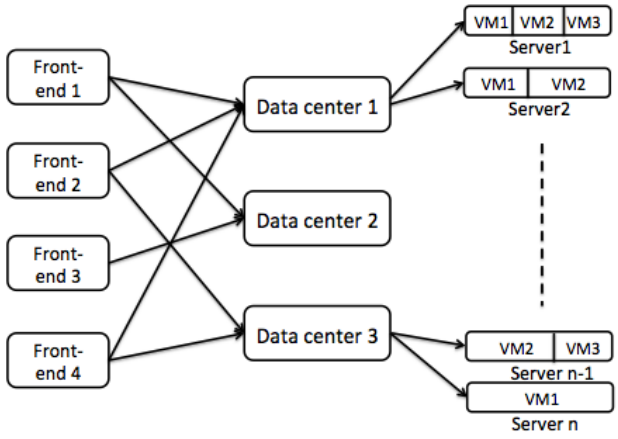


Figure 4.2: System architecture.

cator assigns the tasks according to some metrics to different data center locations. The services are processed and hosted in the data centers. The energy consumption includes the energy consumption for the computation in data centers and that for data transferring during the allocation process.

The real-time tasks, similar to the one in [132], consists of a single sequence of randomly arrived real-time tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, with parameters of τ_i defined as follows.

- S_i: The size of the data that supports the task to be processed;

- $[B_i, W_i]$: The best case execution time B_i and the worst case execution time W_i of τ_i;
- D_i: The relative deadline of τ_i;
- $f_i(T)$: The probability density function for the execution time of τ_i;
- $G_i(t)$: The profit TUF, which represents the profit accrued when a task is completed at time t (relative to its arrival time). We assume $G_i(t)$ is a non-increasing unimodal function before its deadline, i.e. $G(t_i) \geq G(t_j)$ if $t_i \leq t_j$, and $G_i(t) = 0$ if $t > D_i$.
- $L_i(t)$: The penalty TUF, which represents the penalty suffered when a task is discarded at time t (relative to its arrival time). We assume $L_i(t)$ is a non-decreasing unimodal function before its deadline, i.e. $L(t_i) \leq L(t_j)$ if $t_i \leq t_j$, and a task is immediately discarded once it missed its deadline.
- $PC_i(t)$: The processing energy consumption.
- $TC_i(t)$: The data transferring energy consumption.
- $Pr(t)$: The electricity price in a multi-electricity-market, which varies with locations and times.

For the power consumption in processing and data transferring step, we used the model provided in [14], the processing power consumption is given in Equation (4.1):

$$E_S = 1.5 \times T_S \times P_S. \tag{4.1}$$

where T_S is the time for the server to finish the task processing, P_S is the power of the server. The factor of 1.5 is used to account for the cooling energy consumption.

The energy consumed by the data transferring in Internet service provider's network is give in Equation (4.2):

$$E_I = 6 \times (\frac{3P_{es}}{C_{es}} + \frac{P_{bg}}{C_{bg}} + \frac{P_g}{C_g} + \frac{2P_{pe}}{C_{pe}} + \frac{2 \times 9P_c}{C_c} + \frac{8P_w}{2C_w}). \tag{4.2}$$

where P_{es} is the Ethernet switches' power, P_{bg} is broadband gateway routers routers' power, P_g is the power of data center gateway routers, P_{pe} is the power of provider edge routers, P_c and P_w are the power of core routers and Wavelength Division Multiplexed (WDM) fiber links transport equipment, respectively. C_{es}, C_{bg}, C_g, C_{pe}, C_c, and C_w are the capacities of the corresponding equipment in bits per second. The details of this power model can be found in [14]. They per-bit energy consumption of transmission and switching for a public distributed system was estimated to be around $2.7\mu J/b$.

The tasks in the system are associated with a profit function, a penalty function, a processing energy cost function, and an Internet data transferring energy cost function with function values varying with time. Therefore, whenever a task is allocated to the data centers, it incurs a data transferring cost. While the tasks are being executed they have a potential to gain profit, it also has a potential to encounter a penalty at a later time. Once a task starts its execution, a processing energy cost is incurred. The system performance is therefore evaluated by its retained profit, which is the total profit subtracts the penalty and energy costs.

Our problem can be formally defined as: *Given a task set* $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ *as described above, develop an on-line real-time allocation and scheduling method in order to maximize the retained profit for distributed data centers.*

Since the task mode has variable execution times, execution time for a specific task is not known deterministically. We do not know if executing the task will lead to positive profit or loss. In order to make a proper task allocation and schedule policy, we need to generate a metric which is called the *expected retained profit* to help us make decisions.

Given a task τ_i with arrival time of r_i, let its predicted starting time be T, expected execution time is C_i. Then the potential profit ($\overline{G_i}(T)$) to execute τ_i can

be represented as

$$\overline{G_i}(T) = \int_{B_i}^{D_i-(T-r_i)} G_i(t + (T - r_i))f_i(t)dt. \tag{4.3}$$

Similarly, the potential loss ($\overline{L_i}(T)$) to execute τ_i can be represented as

$$\overline{L_i}(T) = L_i(D) \int_{D_i-(T-r_i)}^{W_i} f_i(t)dt. \tag{4.4}$$

Therefore, the expected accrued utility ($\overline{U_i}(T)$) to execute τ_i can be represented as

$$\overline{U_i}(T) = \overline{G_i}(T) - \overline{L_i}(T). \tag{4.5}$$

To account for the power consumed in the process and transfer procedures, we subtract power consumption dollar costs from the expected accrued utility. The expected retained profit is thus given as below

$$\overline{ER_i}(T) = \overline{U_i}(T) - \overline{PC_i}(C_i) - TC_i(r_i). \tag{4.6}$$

where PC_i is the dollar cost for processing. TC_i is the transferring dollar cost. C_i is task τ_i's execution time, and r_i is task τ_i's arrival time.

A task can be accepted or chosen for execution when $\overline{ER_i}(T) > 0$, which means that the probability of to obtain positive retained profit is no smaller than that to incur a loss. We can further limit the task acceptance by imposing a threshold (δ) to the expected accrued utility, i.e. a task is accepted or can be chosen for execution if

$$\overline{ER_i}(T) \geq \delta. \tag{4.7}$$

We call δ as the *expected retained profit threshold*.

Furthermore, since the task execution time is not known a prior, the data centers need to decide whether to continue or abort the execution of a task. The longer the task is executed, the closer the data centers are to the completion point of the

task. At the same time, however, the longer the task executes, the higher penalty the system has to endure if the task cannot meet its deadline. To determine the appropriate time to abort a task, we employ another metric, i.e. the *critical point*.

Assume task τ_i starts its execution at T. Then the potential profit at $T' > T$ (i.e.$\tilde{G}_i(T')$) can be represented as

$$\widetilde{G_i}(T') = \int_1^{D_i-(T-r_i)-(T'-T)} G_i(t+(T-r_i))f_i(t)dt. \tag{4.8}$$

Similarly, the potential loss at $T' > T$ (i.e.$\widetilde{L}_i(T')$) can be represented as

$$\tilde{L}_i(T') = L_i(D)\int_{D_i-(T-r_i)}^{W_i-T'} f_i(t)dt. \tag{4.9}$$

Therefore, the expected accrued utility at $T' > T$ (i.e. $\widetilde{U}_i(T')$) can be represented as

$$\widetilde{U}_i(T') = \widetilde{G_i}(T') - \widetilde{L}_i(T'). \tag{4.10}$$

And the expected retained profit is

$$\widetilde{ER_i}(T') = \widetilde{U}_i(T') - \overline{PC_i}(C_i) - TC_i(r_i). \tag{4.11}$$

We can make $\widetilde{ER_i}(t_0) = \delta$ and solve for t_0. Then when executing task τ_i to time t_0, the expected profit equals its expected loss plus expected power consumption dollar cost. We call t_0 as the *critical point* for executing task τ_i. Due to the non-increasing nature of $\widetilde{G_i}(t)$, the non-negative nature of $\widetilde{P}_i(t)$, and the constant expected process cost $\overline{PC_i}(t)$ and transfer cost $TC_i(t)$, $\widetilde{ER_i}(t)$ is monotonically decreasing as t increases. Therefore, it is not difficult to see that the continuous execution of τ_i beyond the critical point will more likely bring a loss rather than a positive profit.

4.1.2 Our Approach

In Section. 4.1.1, we introduced our system model. Our proposed approach is given in details in this section. We employ a global−local policy for task allocation and scheduling. When a new task arrives, the front end server receives it first. Then, it is sent to the task allocator. The allocator decides which data center is the best one for the new arrival task, and assigns the task accordingly. In each data center, the scheduling will be carried out locally for the tasks assigned to the data centers.

Global Task Allocation

Task allocation part works when a new task arrives. It either assigns the task to one of the data centers, or rejects the task which is estimated that they can not meet the QoS requirement. The details of the task allocation algorithm are shown in Algorithm 4. Suppose we have n data center locations.

Algorithm 4: TASK ALLOCATION

Input : Let τ_i be the new arrival task, and let ta_{ij}, $\overline{C_{ij}}$, ts_{ij} be the arrive time, expected execution time and estimated starting time of τ_i in data center DC_j, respectively. Let current time be t. ER_{ij} is the estimated retained profit of the task at data center j. Let the expected utility density threshold be δ.

```
if A new task, i.e. τ_i arrives then
    for Location j = 1 to n do
        Generate the speculated execution order for τ_i in DC_j, and get its ts_ij;
        Calculate its ER_ij based on its ts_ij for each data center;
    Find the maximum ER_ij;
    if MaxER_ij > δ then
        Assign the task to the location j;
    if MaxER_ij ≤ δ then
        Reject τ_i;
```

When a new task arrives, a speculated execution order of the tasks allocated to a specific processor is generated. Based on this order, the new task has an estimated starting time at this data center. Using this time and the task's arrival time, we can calculate out its potential profit and penalty, its processing energy consumption dollar cost, and the data transferring energy consumption dollar cost during the allocation procedure. The combination of these values gives the expected retained profit of the task. The highest expected retained profit is selected and the task is assigned to the corresponding location. If the highest ER is smaller than the threshold, the task is rejected immediately. Otherwise, it has a high possibility to make the system suffer from a loss. The speculated order is generated by using Algorithm. 5.

Algorithm 5: SPECULATED EXECUTION ORDER

Input : Let $\Gamma = \{\tau_1, \tau_2, ..., \tau_k\}$ be the accepted tasks in the ready queue, and let r_i, $\overline{C_i}$ represent the arrival time and expected execution time of τ_i. Let the current time be t

Output: The new list $\Gamma' = \{\tau'_1, \tau'_2, ..., \tau'_k\}$ with the speculated execution order and their corresponding expected retained profit $\widehat{ER}(\tau'_j)$ for $\tau'_j, 1 \leq j \leq k$.

```
if A task τ_0 is being executed then
    T = r_0 + C̄_0;
else
    T = t;
while Γ is not empty do
    for Each task C in Γ do
        Calculate ER̄_i(T) based on Equations (4.3), (4.4), (4.5), and (4.6);
    Select τ_j with the highest ER̄_j(T);
    Add τ_j to the end of Γ';
    ÊR(τ'_j) = ER̄_j(T);
    T = T + C̄_j;
    Remove τ_j from Γ;
```

When generating the speculated execution order we first calculate the expected retained profit for each task in the ready queue based on the expected finishing time of the current running task. Then the task with the largest one is assumed to be the first task that will be executed after the current task is finished. Based on this assumption, we then calculate the expected retained profit for the rest of the tasks in the ready queue and select the next task. This process continues until all tasks in the ready queue are put in order. When completed, we essentially generate a speculated execution order for the tasks in the ready queue.

Local Task Scheduling

When tasks arrive at the data centers, the scheduling follows the UA criteria, which try to maximize the accrued utility of the system by successfully completing the tasks in time, to schedule the tasks.

The details of our scheduling is described in Algorithm6. The scheduling algorithm works at every scheduling point, which includes new tasks arrival, tasks completion, tasks critical time, and task's deadline missing point. When there is a new job arrives, the scheduler fist checks its expected retain profit at the time when current running task is expected to be finished. If the expected retained profit is larger than the threshold, it is accepted. if not, it is rejected directly. After adding the new task into the ready queue of the data center, the scheduler generates a speculated execution order by using Algorithm 5, and based on this order removes the tasks that can not satisfy the system's requirement. This step is similar to the one in task allocation procedure. The difference is that in local scheduling, the scheduler has to take care of the other waiting tasks to see if some of them need to be removed or not. When a task finishes its execution, it contributes to the system with a positive profit. The finished task also causes processing and transferring cost. At

Algorithm 6: LOCAL TASK SCHEDULING

Input : Let $\{\tau_1, \tau_2, ..., \tau_k\}$ be the accepted tasks in the ready queue, and let $\overline{C_i}$ be the expected execution time of τ_i, and ER_i be the expected retained profit. Let current time be t and let τ_0 be the task currently being executed. Let the expected utility threshold be δ.

if *A new task, i.e.* τ_n *arrives* **then**
    Accept τ_n if $\overline{ER_n}(\overline{C_0}) > \delta$;
    Reject τ_n if $\overline{ER_n}(\overline{C_0}) \leq \delta$;
if τ_n *is added into the ready queue, generate the speculated order for the data center* **then**
    Remove the tasks whose $ER \leq \delta$;
if τ_0 *is completed* **then**
    Choose the highest expected retain profit task τ_i to run;
    Remove τ_j in the ready queue if $\overline{ER_j}(\overline{C_i}) \leq \delta$;
if t *= the critical time of* τ_0*, or* τ_0 *misses its deadline* **then**
    Abort τ_0 immediately;
    Choose the highest expected retain profit task τ_i to run;
    Remove τ_j in the ready queue if $\overline{ER_j}(\overline{C_i}) \leq \delta$;

this time instance, the scheduler selects a new task which has the highest expected retained profit to run. The scheduler works the same when the task reaches to a task's critical time or at deadline missing point. New task selection followed by a scheduling point check. In this step, scheduler further removes the tasks that can not satisfy the system's requirement at the selected task's estimated finishing time.

4.2 Queuing Theory Based Approach

As cloud computing gets more and more prosperous, the number of service requests increases dramatically. In order to handle the large amount of service request, we discuss a queuing theory based approach in this section.

In this research, we present a profit- and cost-aware resource management approach for distributed cloud data centers to optimize a service provider's net profit. By taking advantages of the multi-electricity-market (as shown in Figure 4.1 [98]), our approach has a high efficiency of energy and computing resource usage by judiciously dispatching service requests to different data centers, powering on an appropriate number of servers at different data centers, and adaptively allocating resources to these service requests. Multiple types of services, with no priority difference, are considered in our model. Compared with related work, our contributions in this paper can be summarized as follows:

- We propose a system model that incorporates the multi-electricity-market, SLA, and net profit into a single unified resource management framework. To our best knowledge, this is the first work that deals with multi-electricity-markets, multiple types of requests, and multi-level SLAs, simultaneously.
- We model the profit gained by a service provider as a multi-level step-downward function, which is capable of simulating various scenarios. We formulate our

problem of determining how to dispatch service requests to different data centers, how many servers should be powered on in each data center, and how computing resources should be allocated to service requests as a constrained optimization problem. We also derive a series of constraints to simplify the implementation of our approach.

- The effectiveness of our proposed approach is validated through simulations on both synthetic workload and real data center traces with true electricity price history.

4.2.1 System Model

In this section, we introduce our system model, based on which we develop our time-slotted profit-aware request dispatching and resource management approach for distributed cloud data centers in a multi-electricity-market environment. Our approach periodically runs at the beginning of each time slot T based on the average arrival rates during a slot since job interarrival times are much shorter compared to a slot [73]. Requests arrival pattern forecast is not studied in our work. Existing prediction methods (e.g. the Kalman Filter [124],) or studies (e.g. [40][41]) that have been conducted can be employed if necessary. The length of T is a pre-defined constant that is decided by several factors, e.g. adjusting frequencies of electricity prices (electricity prices stochastically vary over time due to the deregulation of electricity market [101].) We consider that the electricity prices in a time-slot T are constant. Constant prices during a time period are widely implemented in prior work [73][101].

A typical distributed cloud data center system can be illustrated by Figure 4.2. In our system, service requests come from various places and are collected by S

nearby front-end servers, where $S = s_1, s_2, ..., s_S$. Then, requests are dispatched to I capable servers in L data centers via network according to a related metric, where $I = I_1, I_2, ..., I_I$ and $L = l_1, l_2, ..., l_L$.

Virtualization technology, which boosts the realization of the long-held dream of computing as a utility [12], is employed in our architecture to enable server consolidation and simplify computing resource sharing in physical servers. Elasticity of virtualization helps improve the usage efficiencies of computer resources and energy. Different types of services can be held in the same server within their own virtual machines (VMs). The same CPU can be shared by different VMs when necessary. We assume that once a server is powered on, it always runs at its maximum speed. In our scenario, the data centers are heterogeneous, and the servers in a data center are homogeneous. It can be easily extended to heterogeneous data centers with heterogeneous servers.

Task Model

Requests in our system are soft real-time in nature and may encounter both profit and cost. Profit comes from successfully guaranteeing the average delay satisfaction for each type of request [90]. Cost is the dollar cost spent on transferring and processing requests.

Profit:

TUFs are able to precisely specify the semantics of soft real-time constraints [127]. It indicates that in real-time systems, when tasks are completed with respect to their time constraints, the system will be assigned values that vary with the finishing times of the tasks. A TUF can be in any shape. Commonly used TUFs include a constant value before its deadline (Figure 4.3(a)), a monotonic non-increasing function (Figure 4.3(b)), or a multi-level step-downward function (Figure 4.3(c)),

etc. In our scenario, all types of requests desire quick responses. It means that the earlier the tasks are finished, the more utilities they assign to their system. We employ TUFs to represent the profits of processing requests, which are non-increasing functions. Non-increasing TUFs match the SLAs well, since longer delays (beyond some defined time instances) result in lower profits. We will analyze constant value TUFs and multi-level step-downward TUFs in the following sections. These two types of TUFs are representative, especially the multi-level step-downward TUFs. A monotonic non-increasing TUF can be simulated by using a special multi-level step-downward TUF, which has an infinite number of steps. A constant TUF can be simulated as well by using step-downward TUF that only has one step. Consequently, a multi-level step-downward TUF is able to represent a wide range of scenarios and it explains why we mainly focus our study on multi-level step-downward TUFs.

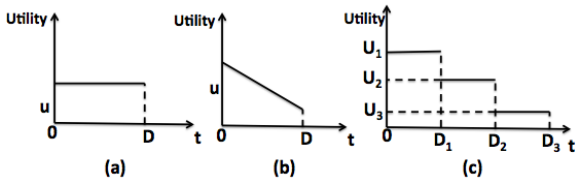


Figure 4.3: Typical TUFs.

Based on the queuing theory, i.e. M/M/1 queue (assuming that the request arriving follows poisson distribution,) it is not difficult to model the expected delay time for k-type requests as [61]

$$R_k = \frac{1}{\phi_k C \mu_k - \lambda_k} \tag{4.12}$$

where in Equation (4.12), C is a server's capacity, and is normalized to 1 in our scenario. In heterogeneous systems, different hardware configurations may have different capacities. μ_k is the processing rate for k-type requests with full capacity.

Note that a server's resource may be shared by many different VMs at the same time. Therefore, its actual processing rate may not be μ_k. ϕ_k indicates the percentage of CPU resource allocated to k-type requests in a single server. λ_k is the arrival rate of k-type requests.

Cost:

Cost consists of two parts. One is the dollar cost for processing requests, the other is the dollar cost for transferring requests.

Processing cost mainly comes from a server's energy consumption. The energy consumption in our work follows the model studied by Google [95] instead of traditional server's energy model. It is based on the energy consumption for processing each single service request. We believe this model is closer to the goal of converting computing ability into one kind of utility in people's daily lives (e.g. electricity.) Then, computing capacity usages are converted into utility consumptions. We assume that energy attributions of the requests are profiled, then the dollar cost on energy consumption for processing requests in a time slot can be expressed as follows:

$$PCost_k = P_k \times \lambda_k \times T \times p \tag{4.13}$$

where $PCost_k$ is the dollar cost for processing k-type requests. λ_k is k-type requests arriving rate. P_k is the energy attribution of k-type requests. Google's study shows that each web search costs $0.0003KWh$ on average. T and p are the length of a time slot (e.g. one hour, which is the same as the electricity prices' changing frequency) and the electricity price at the data center location in a time slot (as shown in Figure 4.1,) respectively.

Dollar cost on transferring requests from a front-end server to a corresponding data center is calculated in a similar method as the one in [96]. As shown in Equation (4.14), it is the product of unit transferring cost ($TranCost_k$) of each type of

request, the distance between the request's origination and destination ($Distance_k$), arrival rate λ_k and the length of a time slot. Since requests may have various characteristics (e.g. sizes,) "$TranCost_k$" is employed to reflect the differences among requests.

$$TCost_k = TranCost_k \times Distance_k \times \lambda_k \times T \tag{4.14}$$

Problem Formulation

With our system architecture and system model defined above, formally, our problem can be formulated as follows:

Problem 4.2.1. Given service requests and data center architectures as described above, develop an efficient on-line profit- and cost-aware workload dispatching and resource allocation approach to maximize net profits for service providers.

4.2.2 Our Approach

In this section, we introduce our approach in detail. For clarity, parameters used in this work are summarized in Table 6.1. We formulate the solution for Problem 6.2.1 as a constrained optimization problem [55][54]. The results are used to decide request dispatching, resource allocation, and the number of servers that should be powered on.

The objective function of Problem 6.2.1 can be mathematically formulated as follows:

$$max \sum_{s=1}^{S} \sum_{l=1}^{L} \sum_{i=1}^{M} \sum_{k=1}^{K} \{U_k(R_{k,i,l})\lambda_{k,s,i,l} - Cost_{k,s,i,l}\lambda_{k,s,i,l}\}T \tag{4.15}$$

After we substitute the factors in Equation (4.15) with Equations (4.12), (4.13) and (4.14), it becomes to Equation (4.16):

$$max \sum_{s=1}^{S} \sum_{l=1}^{L} \sum_{i=1}^{M} \sum_{k=1}^{K} \{U_k(R_{k,i,l})\lambda_{k,s,i,l} - P_{k,l}\lambda_{k,s,i,l}p_l \\ -TranCost_k d_{s,l}\lambda_{k,s,i,l}\}T \tag{4.16}$$

with following constraints:

$$\frac{1}{\phi_{k,i,l}C_{i,l}\mu_{k,l} - \lambda_{k,s,i,l}} \leq D_k, \ \forall k, i, s, l \tag{4.17}$$

$$\sum_{s=1}^{S} \sum_{l=1}^{L} \sum_{i=1}^{M} \lambda_{k,s,i,l} \leq \sum_{s=1}^{S} \lambda_{k,s}, \ \forall k \tag{4.18}$$

$$\sum_{k=1}^{K} \phi_{k,i,l} \leq 1, \ \forall i, s, l \tag{4.19}$$

Constraint (4.17) shows the QoS requirement. The average delay for each type of request cannot exceed its deadline. Constraint (4.18) assures that the number of assigned requests does not exceed the number of total service requests coming from the Internet. Constraint (4.19) bounds the CPU share by various types of services in a single server.

In our constrained optimization formulae, $\phi_{k,i,l}$ and $\lambda_{k,s,i,l}$ are the two variables that need to be solved, representing where to assign and how much workload should be assigned from each front-end server. In addition, as we know how requests are dispatched, we can determine how many servers should be powered on. Clearly, when there is no workload on a server, the server should be powered off.

In our model, we assume that server switching costs and durations are negligible compared to the total energy consumption and time of processing and transferring requests during a time slot (e.g. one hour.)

The complexity of our objective function depends heavily on the format of the utility function used to reflect a request's potential profit. Since multi-level step-downward TUFs are representative and cover a large scenario diversity, in what follows, we discuss three typical multi-level step-downward utility functions, and corresponding solutions for each of them. As stair TUFs need "*if-else*" descriptions, which are unfortunately not well supported by some popular non-linear mathematic programming (or some constraint logic programming) solvers, e.g. Prolog, we hence transform the "*if-else*" into a set of constraints.

One-Level Step-Downward TUF

The first type of TUF has a constant utility before deadline and can be expressed as follows:

$$U_k = TUF(R_k) = \begin{cases} U_{k,1} & 0 < R_k \leq D_k \\ 0 & R_k > D_k \end{cases} \tag{4.20}$$

where, U_k is the utility of k-type requests. $U_{k,1}$ is a constant value. Before delay time R_k exceeds deadline D_k, U_k equals to $U_{k,1}$.

With one-level step-downward TUF, the objective function (Equation (4.16)) is simply a linear function. Even though there is a nonlinear component in Equation (4.17), it can be linearized through simple transformations, i.e. $\phi_{k,i,l} C_{i,l} \mu_{k,l} - \lambda_{k,s,i,l} \geq \frac{1}{D_k}$. The whole problem can be solved by using traditional linear programming solvers [39].

Two-Level Step-Downward TUF

This type of TUF can be expressed as Equation (4.21), where U_k is the utility of k-type requests. R_k is the delay time of k-type requests. $D_{k,q}$ is the relative sub-deadline for each utility level $U_{k,q}$, and q is the index of each level (i.e. the q-th

sub-deadline of k-type requests to achieve the q-th utility level.) We assume that D_k is the final deadline for k-type service requests. Executing a request becomes meaningless once the delay time exceeds D_k.

$$U_k = TUF(R_k) = \begin{cases} U_{k,1} & 0 < R_k \leq D_{k,1} \\ U_{k,2} & D_{k,1} < R_k \leq D_k \\ 0 & R_k > D_k \end{cases} \tag{4.21}$$

Note that when the TUF employs a two-level step-downward function, the objective function is no longer a linear one. Furthermore, with Equation (4.21), it is challenging to formulate the objective in one formula. To solve this problem, we transform Equation (4.21) to a set of extra constraints as follows:

$$U_k \in \{U_{k,1}, U_{k,2}\}, (U_{k,1} > U_{k,2}) \tag{4.22}$$

$$(R_k - D_{k,1}) + \uplus(U_k - U_{k,1}) <= 0 \tag{4.23}$$

$$(D_{k,1} + \delta - R_k) + \uplus(U_{k,2} - U_k) <= 0 \tag{4.24}$$

where, $\uplus$ is a large constant. δ is a constant time value which is small enough. $D_{k,1} + \delta$ indicates the time instance that immediately follows time $D_{k,1}$.

To see the reason that Equation (4.21) can be equivalently transformed to a set of constraints listed in Equations (4.22), (4.23) and (4.24), consider the following two cases:

- When $0 < R_k \leq D_{k,1}$

  Under this condition, we readily have $R_k - D_{k,1} \leq 0$. From Equation (4.22), U_k can be either $U_{k,1}$ or $U_{k,2}$. Therefore, to satisfy Equation (4.24), we must

have $U_k = U_{k,1}$. In the meantime, Equation (4.24) can be easily satisfied as long as $\uplus$ is large enough. To this end, $U_k = U_{k,1}$ is the only solution when $0 < R_k \leq D_{k,1}$.

- When $R_k > D_{k,1}$

  Under this condition, we readily have $D_{k,1} + \delta - R_k \leq 0$. Since U_k can be either $U_{k,1}$ or $U_{k,2}$, to satisfy Equation (4.23), we must have $U_k = U_{k,2}$. In the meantime, Equation (4.23) can be easily satisfied as long as $\uplus$ is large enough. To this end, $U_k = U_{k,2}$ is the only solution when $R_k > D_{k,1}$.

While we can transform Equation (4.21) to a set of constraints listed in Equations (4.22) – (4.24), the problem is not over. Note that Equation (4.22) is still a constraint that is not formulated properly. To formulate the constraint in Equation (4.22), we can define an integer variable x with

$$0 \leq x \leq 1 \tag{4.25}$$

such that

$$U = xU_{k,1} + (1 - x)U_{k,2} \tag{4.26}$$

With the extra constraints listed in Equations (4.22) – (4.24), it is desirable to use traditional integer linear programming solver to solve the problem. Unfortunately, this is not feasible. From Equation (4.12), it is not difficult to see that both Constraints (4.23) and (4.24) are non-linear formulae. To solve this problem, we need to employ the constraint logic programming solvers or nonlinear mathematic programming solvers such as ILOG CPLEX [51] and AIMMS [2] to find the near optimal solutions. With the help of the series of constraints, people may avoid the

difficulty of implementing "$if, else$" statement in some solvers. Similar series can be derived for multi-level step-downward TUFs.

Three or More Level Step-Downward TUF

This type of TUF can be formulated as follows:

$$U_k = TUF(R_k) = \begin{cases} U_{k,1} & 0 < R_k \leq D_{k,1} \\ U_{k,2} & D_{k,1} < R_k \leq D_{k,2} \\ U_{k,3} & D_{k,2} < R_k \leq D_{k,3} \\ \vdots & \\ 0 & R_k > D_k \end{cases} \tag{4.27}$$

Similarly, Equation (4.27) can be transformed into a series of new constraints as listed below:

$$U_k \in \{U_{k,1}, U_{k,2}, U_{k,3}, ..., U_{k,n}\}$$

$$\begin{aligned} (R_k - D_{k,1}) + \uplus(U_k - U_{k,1}) &<= 0 \\ (D_{k,1} + \delta - R_k) + \uplus(U_{k,2} - U_k)(U_k - U_{k,3}) &<= 0 \\ (R_k - D_{k,2}) + \uplus(U_{k,2} - U_k)(U_k - U_{k,1}) &<= 0 \\ (D_{k,2} + \delta - R_k) + \uplus(U_{k,3} - U_k)(U_k - U_{k,4}) &<= 0 \\ (R_k - D_{k,3}) + \uplus(U_{k,3} - U_k)(U_k - U_{k,2}) &<= 0 \\ &\vdots \\ (D_{k,n-1} + \delta - R_k) + \uplus(U_{k,n} - U_k) &<= 0 \end{aligned} \tag{4.28}$$

where, $U_{k,1} \ldots U_{k,n}$, $D_{k,1} \ldots D_{k,n}$, $\uplus$, and δ are the same as those in a two-level step-downward function, and $U_{k,1} > U_{k,2} > \cdots > U_{k,n}$. Take $n = 3$ as an example,

we have:

$$U_k \in \{U_{k,1}, U_{k,2}, U_{k,3}\}, (U_{k,1} > U_{k,2} > U_{k,3}) \tag{4.29}$$

$$(R_k - D_{k,1}) + \uplus(U_k - U_{k,1}) <= 0 \tag{4.30}$$

$$(D_{k,1} + \delta - R_k) + \uplus(U_{k,2} - U_k)(U_k - U_{k,3}) <= 0 \tag{4.31}$$

$$(R_k - D_{k,2}) + \uplus(U_{k,2} - U_k)(U_k - U_{k,1}) <= 0 \tag{4.32}$$

$$(D_{k,2} + \delta - R_k) + \uplus(U_{k,3} - U_k) <= 0 \tag{4.33}$$

Equations (4.30) and (4.33) are very similar to Equations (4.23) and (4.24). The newly added constraints are Equations (4.31) and (4.32). Note that, similar to the analysis above, it is not difficult to see that as long as $\uplus$ is large enough, we must have $U_k = U_{k,1}$ when $R_k \leq D_{k,1}$, and $U_k = U_{k,3}$, when $R_k > D_{k,2}$ to satisfy Constraints (4.30) – (4.33).

Now, consider the situation when $D_{k,1} < R_k \leq D_{k,2}$ (note that $U_{k,1} > U_{k,2} > U_{k,3}$.) Under this condition, similarly, Equations (4.31) and (4.32) can be easily satisfied with any $U_k \in \{U_{k,1}, U_{k,2}, U_{k,3}\}$. From Equation (4.30), we can conclude that, to satisfy Equation (4.32), we must have

$$(U_k - U_{k,1}) < 0$$

that is, we have either

$$U_k = U_{k,2} \quad or \quad U_k = U_{k,3} \tag{4.34}$$

Meanwhile, to satisfy Equation (4.33), we must have

$$(U_{k,3} - U_k) < 0$$

that is, we have either

$$U_k = U_{k,1} \quad or \quad U_k = U_{k,2} \tag{4.35}$$

Therefore, to satisfy Equations (4.30) – (4.33), we must have $U_k = U_{k,2}$ when $D_{k,1} < R_k \leq D_{k,2}$.

We have shown that Equation (4.27) can be transformed equivalently into Equation (4.28). The problem becomes how to formulate U_k (Equation (4.29)) using a general form. Similarly, we can introduce an integer variable x, with

$$1 \leq x \leq n \tag{4.36}$$

where, n is the number of step levels. Then U_k can be formulated as follows:

$$U_k = \frac{\sum_{i=1}^{i \leq n} [\Pi_{j=0, j!=i}^{j \leq n} (j - x)] U_{k,i}}{(-1)^x x!(n - x)!} \tag{4.37}$$

As a result, U_k is successfully transformed into a series of constraints as described in Equations (4.28), (4.36), and (4.37). Same as above, together with our objective function, this constraint series can be solved by using constraint logic programming solvers or nonlinear programming solvers.

4.3 Experiment

We use simulation experiments to investigate the performance of our proposed approach.

4.3.1 TUF Based Approach

Experiment Setup

We evaluate our proposed method's performance based on several known Google's IDC locations with real-life electricity prices (Houston, Texas, Mountain View, California, and Atlanta, Georgia). The test cases in our experiments were randomly generated. Specifically, S, B, W, and D were randomly generated such that they are uniformly distributed within interval of $[1MegaByte, 2MegaByte]$, $[1, 10]$, $[30, 50]$, and $[40, 50]$, respectively. The execution time of a task is assumed to be evenly distributed between interval of [B,W], i.e. $f(t) = \frac{1}{W-B}$. G, L were assumed to be linear functions, i.e. $G(t) = -a_g(t - D)$ in the range of $[0, D]$ and $L(t) = a_l(t - t_a)$, where t_a is task's arrival time. The gradient for $G(t)$ and $L(t)$, i.e. a_g and a_l were randomly picked from the interval of $[0.5, 1]$ and $[0.1, 0.3]$, respectively. The power P of the homogeneous servers is set to be $1KW$ for the computing ease. The per-bit transferring energy cost is 2.7 $\mu J/bit$. $PC(t_e) = 1.5 \times t_e \times P \times Pr(t_s)$, where t_e is the execution time, t_s is the time when the task starts processing. $TC(t_a) = 6 \times (\frac{3P_{es}}{C_{es}} + \frac{P_{bg}}{C_{bg}} + \frac{P_g}{C_g} + \frac{2P_{pe}}{C_{pe}} + \frac{2\times 9P_c}{C_c} + \frac{8P_w}{2C_w}) \times S \times Pr(t_a)$, which can be simplified as $TC(t_a) = 2.7\mu J/bit \times S \times Pr$. We use arrival time here because once the task arrives, it will be immediately allocated or rejected. Task release times' intervals follow the exponential distribution with $\mu = 2$. Pr we used is shown in Figure. 4.1 comes from [98], they simulate their approach with the real multi-electricity-market price. It tracks the prices of three Google's data center locations. The retained profit threshold δ is set to 0. We conducted several different groups of experiments to study and compare the performances of different approaches. The results are reported as follows.

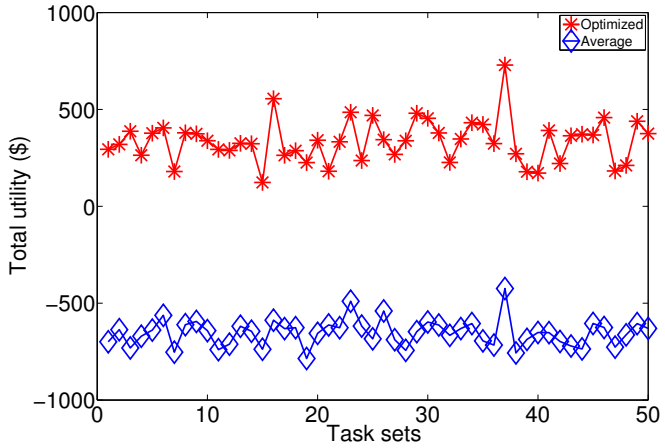


Figure 4.4: Comparison of accrued retained profit.

Experiment Results

We first conducted an experiment using a thousand task sets, each with a hundred tasks. We ran our proposed optimized approach and the naive approach on the same task sets. Our new optimized approach aims at increasing the system's accrued retained profit by successfully completing tasks and reducing the tasks' power consumption dollar costs at the same time.

To show the details clearer, Figure 4.4 displays only the results of the first fifty sets. We can tell that when compared with the naive approach, our optimized approach has a much better performance at attaining higher accrued retained profit.

Figure 4.5 compares the profits gotten by two approaches. Our optimized approach achieves higher profit since the optimized scheduling finishes more tasks in time successfully than naive scheduling does. Figure 4.6 indicates the penalty comparison. From Figure 4.6, we can see that our approach also outperforms the naive approach in the penalty control by discarding less running tasks and missing less deadlines. In our approach, the scheduler carefully accepts potential tasks and ju-

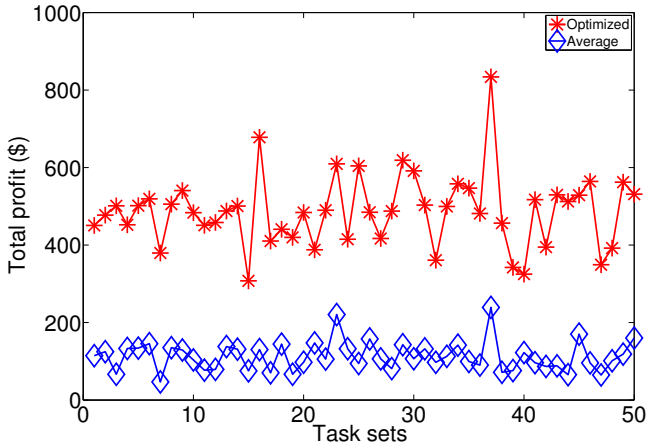


Figure 4.5: Comparison of profit

diciously discards running task if it has high possibility to cause loss to the system. Moreover, in the task allocation process, there is an early screening step at the allocation step. The tasks which can not find a proper data center location will be rejected (even the largest expected retained profit for the task is smaller the the threshold δ). The two approaches transfer different number of tasks. This is the reason that why data centers in our optimized approach host less tasks and results in less processing and transferring cost, which is proven by Figure. 4.7(a) and Figure. 4.7(b). Figure 4.7(a) reflects the higher processing efficiency (less hosting tasks and higher profit) of optimized approach. Because of the scheduler's predictability, the tasks that have potential to bring loss are discarded during execution or removed while waiting in the queue. On the contrary, the naive approach always try to finish every executing tasks or keep waiting tasks until they meet their deadline. The lack of forecast induces low processing efficiency. In Figure. 4.8 we can see our optimized approach completes more tasks than the naive approach does.

In Figure. 4.9, the histogram shows the number of assigned tasks of three different data centers. DC3 has the highest average electricity price. The figure shows

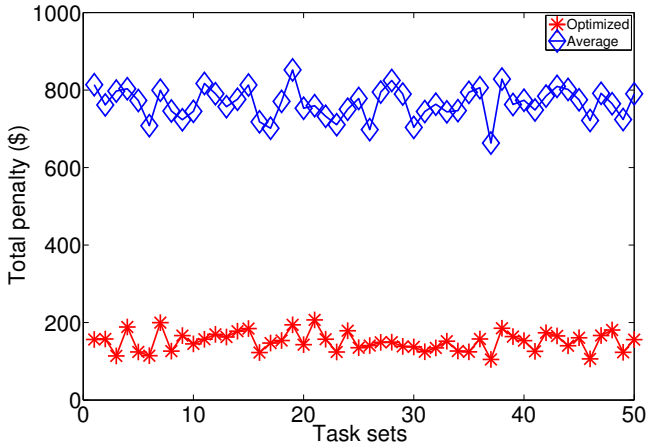


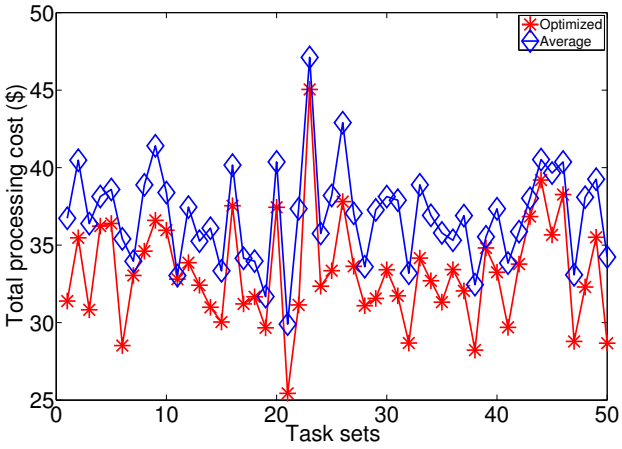
Figure 4.6: Comparison of penalty

the average number of tasks assigned to DC3 in optimized approach is significantly smaller than that in naive approach, and also smaller than the number of tasks assigned into DC1 or DC2 in the optimized approach. As shown in the figure, the total number of tasks allocated in optimized approach is less than the number assigned by naive method. As we have explained previously that in the task allocation step, some tasks, which can not find a proper host data center, are rejected instead of being sent to the destinations. Some certain amount of energy is saved here because less task allocations take place in optimized approach. This has already been shown in Figure. 4.7(b).

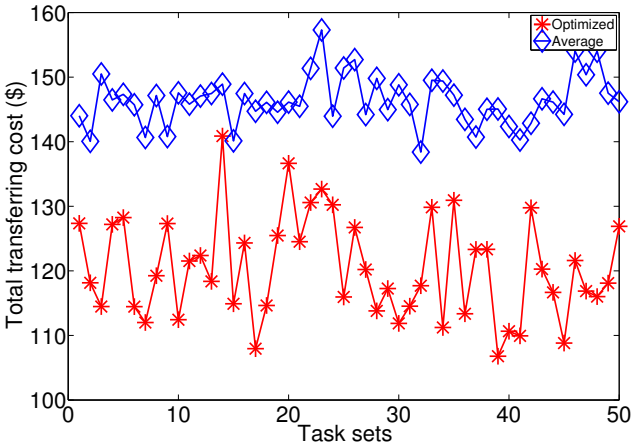
4.3.2 Queuing Theory Based Approach

Study with Real Traces using One-Level Step-Downward TUFs

In this study, we employed a real trace of the 1998 World Cup [10] to generate our service requests. We used a trace that contains requests spanning four different days as the service requests in a day collected by four front-end servers. There are three data center locations providing services to three types of requests dispatched



(a) Processing power consumption



(b) Transferring power consumption

Figure 4.7: Power consumption comparison

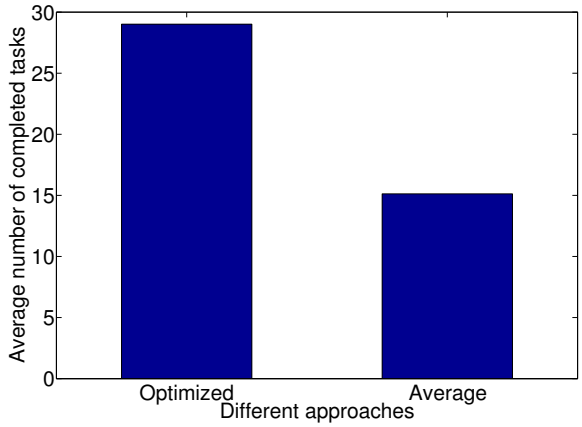


Figure 4.8: Comparison of number of completed tasks

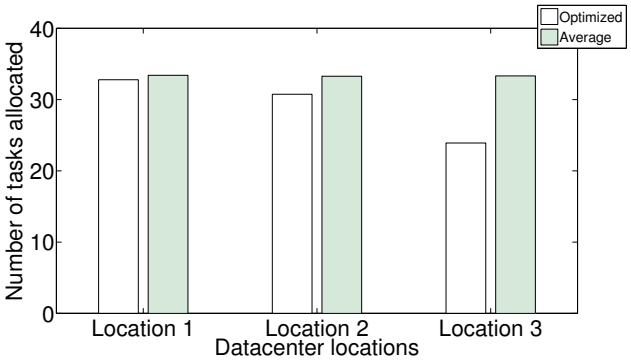
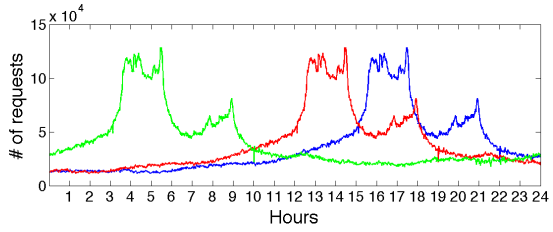
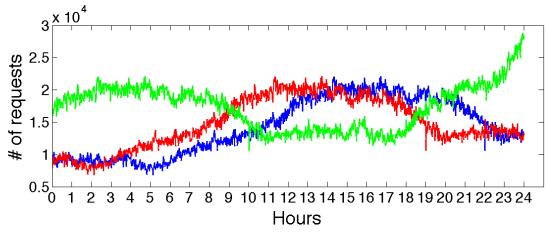


Figure 4.9: Number of task allocation in different data centers.

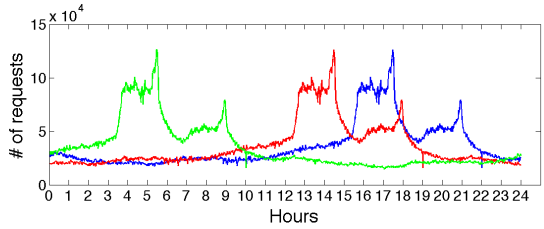
from the four front-end servers. Each data center has six homogeneous servers. We simply shifted the request traces at a front-end server by some time units to simulate the requests of three different service types. The traces generated are shown in Figure 4.10. Electricity prices, as shown in Figure 4.1, are the real data collected from three locations, i.e. Houston, TX, Mountain View, CA, and Atlanta, GA.



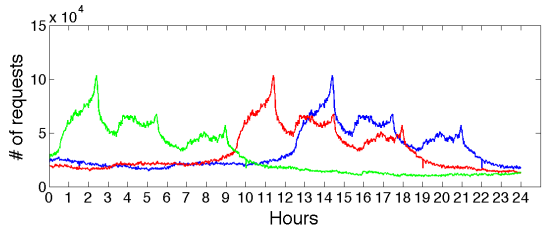
(a) Request at front-end server 1.



(b) Request at front-end server 2.



(c) Request at front-end server3.



(d) Requests at front-end server4.

Figure 4.10: Request traces

A data center's processing capacities and distances among front-end servers and

data centers are generated randomly and given in Tables 4.2 and 4.3, respectively. Processing energy costs (Table 4.4) are given based on the data provided by Google's research blog [95], which are around $0.0003kWh$ for each web search request. TUFs and sub-deadlines for each type of request are collected in Table 4.5. Transferring costs for the three types of requests are 0.003$/mile, 0.005$/mile, and 0.007$/mile. Even though parts of the experiment setup were generated randomly, the experiment does not loss the generality. "Optimized" and "Balanced" approaches are compared.

Net profit:

We first checked the net profits achieved by the two approaches. As explained in previous sections, our new model takes overall factors into consideration and provides a flexible request dispatching and resource allocation strategy to obtain a high net profit. This claim is supported by the results of running real request traces, as shown in Figure 4.11. Our new proposed approach ran the model once an hour (the length of a time slot is an hour). It is obvious that our approach significantly outperforms the static approach in achieving a net profit.

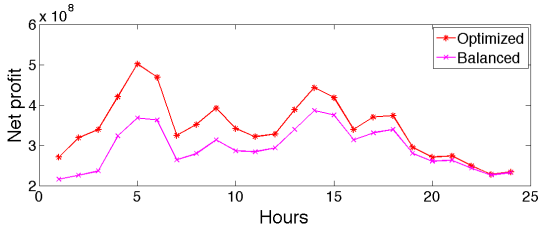
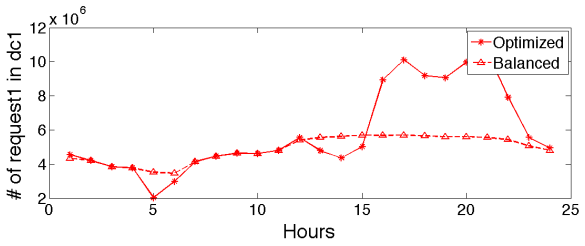


Figure 4.11: Net profits obtained by two approaches.

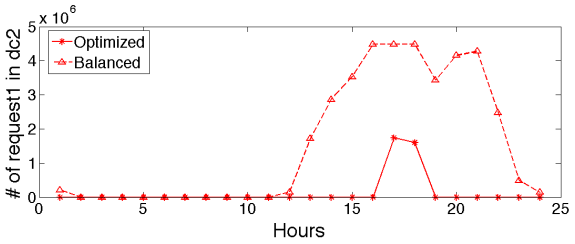
Request dispatching:

We then studied the request dispatching of these two approaches, which are illustrated in Figure 4.12. From the experiment setup, we can see that for Request1, Datacenter1 and Datacenter2 have the same processing capacity, and Datacenter3 has the highest processing rate. In addition, the distances between Datacenter2 and

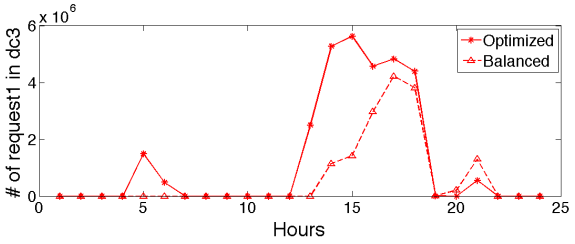
the four front-end servers are the longest. Taking the transferring cost and processing capacities into consideration, Datacenter1 and Datacenter3 were better choices than Datacenter2 for Request1. All things considered, Datacenter2 did process some of the requests to improve the whole system's performance. However, the number of requests dispatched to Datacenter2 was still much smaller than the numbers of requests assigned to Datacenter1 and Datacenter3.



(a) Request1 allocated to datacenter1.



(b) Request1 allocated to datacenter2.



(c) Request1 allocated to datacenter3.

Figure 4.12: Allocations for request 1.

Request2 and Request3 assignments are omitted because of space limit. Similar to the allocation of Request1 shown in Figure 4.12, in *"Optimized"* Datacenter1 and Datacenter3 had a large part of the requests dispatched. Datacenter2 almost had no request assigned. We can see that all request dispatchings in these figures have

similar results at the end of the traces in both *"Optimized"* and *"Balanced."* This explains why *"Optimized"* and *"Balanced"* had similar net profits at the end of the traces, as shown in Figure 4.11.

Study with Real Traces using Two-Level Step-Downward TUFs

Finally, we conducted an experiment to analyze computing resource allocation and request dispatching effect for a real Google workload trace that was recorded in 2010 [48]. This dataset traces over a 7-hour period. It consists of a set of tasks, where each of them runs on a single machine. We duplicated the trace and moved along time scale to simulate two different types of requests. This time we implemented two-step TUFs to represent the possible profits that may be achieved by completing requests successfully. We implemented our equation series derived in Sections 4.2.2 and 4.2.2 in both ILOG CPLEX and AIMMS (constraint logic programming solver and nonlinear programming solver.) In our experiments, we assumed that two types of requests come from one single front-end server, and then dispatched to two data centers. There are six servers in each data center. Electricity prices for the two locations follow the prices of Houston and Mountain View, as shown in Figure 4.1. We selected electricity prices in the time period between 14:00 and 19:00, because the prices in that period are representative in terms of large price vibration. Processing capacities of the two types of requests in each data center were randomly generated and are shown in Table 4.6. Sub-deadlines and TUF values are shown in Table 4.7 and Table 4.8, respectively. The power consumptions of the two types of requests in each data center are summarized in Table 4.9. We simply further assumed that the distances from the front-end server to those data centers are 1000 $miles$ and 2000 $miles$. The transferring costs for the two types of requests are 0.00003 and 0.00005 $\$/mile$.

Net profit:

Net profit achieved from the real trace using our optimized approach is shown in Figure 4.13. The optimized approach outperforms the balanced approach significantly. It clearly illustrates that our optimization efficiently uses electricity price difference to establish its superiority. In Figure 4.13, electricity price differences between Hour2 and Hour4 are larger than those at other times. The advantage of our approach is boosted at those time instances.

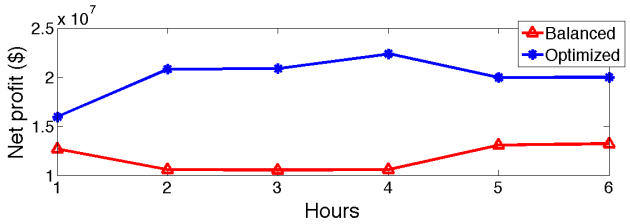


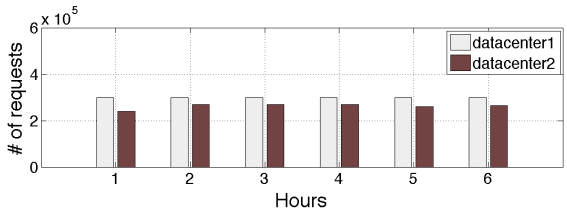
Figure 4.13: Net profits obtained by two approaches with two-step TUFs.

Request dispatching:

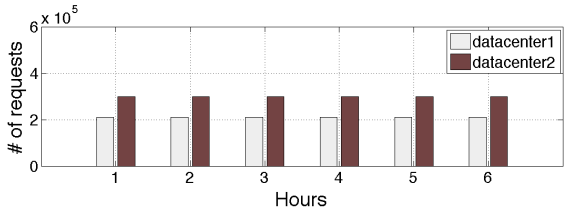
The major difference between these two approaches comes from the request dispatching. As shown in Figure 4.14, it is obvious that "*Optimized*" rationally dispatched requests according to electricity prices, transferring costs, and processing capacities of each data center. All Request1 and Request2 were completed in "*Optimized*." On the contrary, 99.45% request1 and 90.19% request2 were completed in "*Balance*." Even though "*Optimized*" spent 7.74% more on the cost, it achieved a higher net profit. This observation is reasonable, since our optimization approach optimizes the trade off among several target components instead of optimizing each of them.

Workload effect:

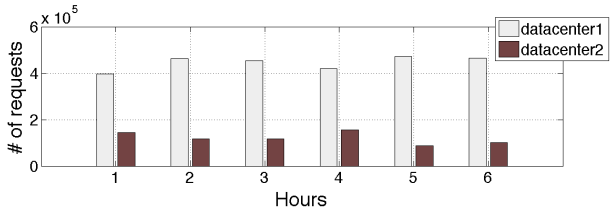
We then increased data center capacities in order to simulate a relatively low workload situation (i.e. all requests in two approaches can be completed successfully.) The result is shown in Figure 4.15(a). A relatively high workload situation was



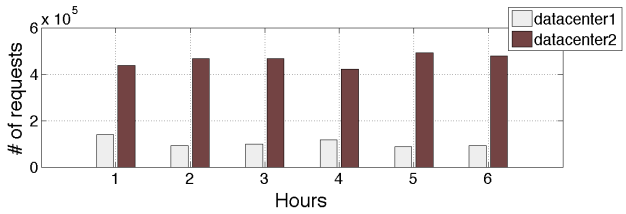
(a) Request1 allocation using balanced approach.



(b) Request2 allocation using balanced approach.



(c) Request1 allocation using optimized approach.



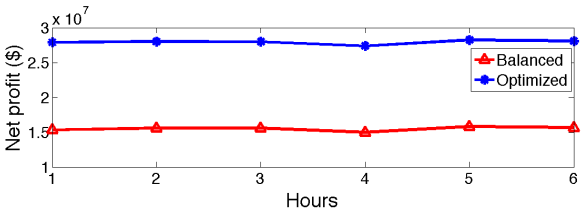
(d) Request2 allocation using optimized approach.

Figure 4.14: Allocations of the requests.

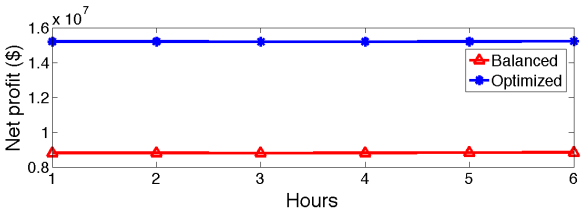
tested as well (i.e. no approach can complete all requests.) The result is shown in Figure 4.15(b). These figures prove that our optimization is superior regardless of workloads.

Computation time:

We kept the experiment setup except changing the number of servers in each data center to service randomly generated number of service requests. We ran each server set five times and their average values were used to represent the computation times.



(a) Net profits comparison with a relatively low workload.



(b) Net profits comparison with a relatively high workload.

Figure 4.15: Low/High workload situations.

Results are shown in Figure 4.16. As can be seen in the figure, the computation time increased exponentially.

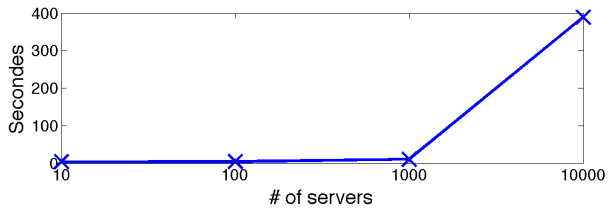


Figure 4.16: Computation times of different server sets.

4.4 Conclusions

Cloud computing systems are proliferating. They provide increasingly diverse network-based services and applications. A large number of requests increases both the scale of data centers and their energy consumptions. Efficient request dispatching and resource allocation algorithms are in urgent need by service providers for achieving high net profits.

In this chapter, we present two optimization-based profit- and cost-aware approaches to maximize the net profits that service providers may achieve when operating distributed cloud data centers in multi-electricity-market environments. We developed a system model to effectively capture the relationship among several factors, such as SLA, cost on energy consumption, service request dispatching and resource allocation. By considering overall factors, our approach judiciously dispatches requests and allocates computing resources. Significant net profit improvement can be achieved by efficiently using energy and computing resources.

Till now, the scheduling approaches we devised are all for single-tier services. Considering the fact that today's cloud applications usually consist of several interdependent services, we move to the study of request scheduling for multi-tier services in next chapter.

Parameters	Definations
K	number of service types in the system.
S	number of front-end servers in the system.
L	number of data centers in the system.
M_l	number of homogeneous servers in data center l.
$C_{i,l}$	capacity of server i in data center l.
μ_k	service rate for k-type requests at a server of capacity 1.
$\lambda_{k,s,i,l}$	k-type requests dispatched to server i in data center l comes from front-end server s.
$\phi_{k,i,l}$	CPU share for k-type requests at server i in data center l.
$R_{k,i,l}$	delay time for k-type requests at server i in data center l.
U_k	utility function for k-type requests. U_k^q corresponds to the utility in q_{th} level.
$D_{k,q}$	relative sub-deadline for the q_{th} utility level.
D_k	relative deadline for k-type requests.
$P_{k,l}$	energy cost for processing k-type requests in data center l.
p_l	electricity price at data center l at time t.
$d_{s,l}$	distance between front-end server f and data center l.
$PCost_{k,l}$	processing cost of k-type requests at data center l.
$TranCost_k$	unit transferring cost of k-type requests.

Table 4.1: Parameter notation.

capacity	datacenter 1	datacenter 2	datacenter 3
$request1(\#/hour)$	3000000	3000000	3600000
$request2(\#/hour)$	3300000	3000000	3600000
$request3(\#/hour)$	3000000	3600000	4200000

Table 4.2: Processing capacities of each data center.

Distance	datacenter 1	datacenter 2	datacenter 3
$front-end1(miles)$	1000	2000	1500
$front-end2(miles)$	800	1500	1000
$front-end3(miles)$	500	1000	1000
$front-end4(miles)$	1000	1500	1000

Table 4.3: Distance among front-end servers and data centers.

Processing cost	datacenter 1	datacenter 2	datacenter 3
$request1(kWh)$	0.0003	0.0004	0.0006
$request2(kWh)$	0.0004	0.0003	0.0003
$request3(kWh)$	0.0006	0.0005	0.0005

Table 4.4: Processing cost at each data center for different types of services.

TUF	Max value	Deadline
$request1$	10 ($)	0.016 (hour)
$request2$	20 ($)	0.023 (hour)
$request3$	30 ($)	0.048 (hour)

Table 4.5: TUFs for each type of request.

Capacity	datacenter1	datacenter2
$request1(\#/hour)$	80000	70000
$request2(\#/hour)$	90000	100000

Table 4.6: Processing capacities of each data center.

Sub-deadline	request1	request2
$sub-deadline1$(hour)	0.001	0.01
$sub-deadline2$(hour)	0.002	0.02

Table 4.7: Sub-deadlines of the request.

TUF values	level1	level2	level3
$request1$($)	20	10	0
$request2$($)	30	10	0

Table 4.8: TUF values at different steps of the requests.

Power	datacenter1	datacenter2
$request1$(kWh)	0.0002	0.0003
$request2$(kWh)	0.0001	0.0003

Table 4.9: Power consumption of the requests in each data center.

CHAPTER 5

MULTI-TIER SERVICE SCHEDULING IN A DISTRIBUTED ENVIRONMENT

As web applications grow tremendously in both scale and scope, more and more applications employ multi-tier computing infrastructures. Scheduling for multi-tier services is more complicated than that for single-tier services. In this chapter, we study the problem of scheduling delay-sensitive multi-tier services with probabilistic performance guarantee.

5.1 Research Problem Introduction

With the prosperity of cloud computing, the application patterns become more and more sophisticated. An application request coming from the Internet usually needs to go through multiple service tiers hosted in different machines at different locations. Different from single-tier applications, multi-tier applications have more intricate inter-tier interactions. A change of a request's execution in a tier may cause the request's (or even other requests') QoS violation.

In addition, there have been an increasing number of time sensitive applications, such as on-line gaming (e.g. Uncharted 3: Drake's Deception [114]) or other streaming multimedia (e.g. Adobe Media Server [1]) deployed on the web (e.g. Rackspace Cloud Media Hosting [97] and AWS [6].) Because of these time sensitive applications, stringent timing constraints are added to the processing infrastructure for desired user experience. Besides the timing constraints, in order to fully satisfy a client's QoS requirements a service provider must ensure that *adequate* requests can be served successfully in time.

In this research, we study the problem of how to efficiently manage a set of multi-tier, time sensitive application requests with probabilistic guarantee of their end-to-

end deadlines. Specifically, the QoS in this paper refers to a constraint described by the probability of dropping an application's request with respect to its end-to-end deadline. While satisfying the QoS, we intend to reduce the average response time for an application. As response time has become an important performance metric [93], and since the non-increasing TUFs are widely employed [68] to analyze the relationship between a request's response time and its processing profit, scheduling approaches that have shorter response times are more desirable.

Different from the traditional dynamic resource provisioning approaches [121][64], we propose a stochastic approach that can judiciously prune the application requests on a given distributed platform to address this challenging problem for the situations with and without resource sharing. Previously, many similar studies employed acceptance control or random deletion to guarantee the end-to-end QoS satisfactions [62][76]. Instead of targeting on the potential mischievous requests, these methods randomly selected the requests to reject or remove. We propose a method that identifies the potential failure requests and terminates these requests. However, how to find the potential failure requests and when to remove them from the system are not trivial problems, especially in the situations where computing resources are shared among different applications.

In our approach, a sub deadline is associated with each service of a time sensitive application. A request is dropped if any of its services miss the associated sub deadline. The rationale of our approach is that when an application request is more likely to miss its end-to-end deadline, and thus is of no use to the system, it is better off to remove the request as early as possible. Dropping a request helps saving the precious computing resources and energy for requests that are more likely to be successfully fulfilled, which would most likely be wasted otherwise. However, requests dropping degrades the QoS of the system. Making the appropriate trade-

off is the key to this problem. To this end, we transform the deadline assignment problem to a queueing problem with reneges [15] and develop an algorithm to determine sub deadlines for the services analytically in the situations with and without resource sharing. Our experimental results demonstrate that our approach is able to statistically guarantee the QoS requirements with higher efficiencies (i.e. achieving required completion ratios with shorter average response times.)

5.2 Preliminary

In this section we introduce our system model and formulate our problem formally.

5.2.1 Service Model

We assume that our application architecture consists of m servers, i.e. $\mathcal{E} = \{E_1, E_2, ..., E_m\}$, each of which provides one dedicated service. The service time of each server, i.e. E_i, is independent from each other and follows an exponential distribution with the average processing time of $\frac{1}{\mu_i}$.

(a) A multi-tier services application.

(b) A queue-based view of an application.

Figure 5.1: Application model.

There is a total of k applications $\mathcal{S} = \{S_1, S_2, ..., S_k\}$ in the system. We assume that the requests arrival pattern for each application follows the Poisson distribution, and each application is modeled with a 4-tuple, i.e. $S_i = \{\lambda_i, D_i, R_i, L_i\}$, where

- λ_i is the arrival rate of requests for S_i;
- D_i is the end-to-end deadline of S_i;
- R_i is the end-to-end deadline satisfaction ratio (aka. the completion ratio) for S_i;
- L_i contains an ordered list of servers that S_i needs to go through, i.e. $L_i = \{E_{i1}, E_{i2}, ..., E_{in}\}$, where $E_{ij} \in \mathcal{E}$.

The multi-tier services in an application and their queuing models are illustrated in Figure 5.1. Note that R_i and D_i together quantify the QoS requirements of S_i, and L_i represents the dependent services that S_i needs to go through. An instance of S_i, triggered by a corresponding service request, is considered successfully completed only if it goes through all the designated servers and finishes its execution before the end-to-end deadline. Otherwise, it is deemed as a failure. For different applications S_i and S_j, $L_i \cap L_j$ may or may not be $\emptyset$, indicating that different applications may or may not share the same servers.

5.2.2 Problem Definition

We assume that, for each application $S_i = \{\lambda_i, D_i, R_i, L_i\}$, a sub deadline (e.g. d_{ij}) is assigned to each server (e.g. $E_{ij} \in L_i$) that S_i needs to go through. A request is dropped if any of its services miss the deadline. Removing a request from the queue helps save system resources for other requests that are more likely to be completed in time. On the other hand, such an action may lead to violating the QoS requirements as identified by the deadline miss ratios. The problem is then how to judiciously determine the sub deadline for each server. These deadlines are local sub deadlines and the interval between any two adjacent sub deadlines is used to indicate how long a request is allowed for being processed in that server.

With the system model and assumptions introduced above, we formulate our research problem as follows:

Problem 5.2.1. *Given an application platform $\mathcal{E}$ and an application set $\mathcal{S}$, determine the sub deadline d_{ij} on each server (e.g. $E_{ij} \in L_i$) for application S_i such that no less than R_i percent of S_i requests can successfully meet their end-to-end deadline D_i.*

5.3 Sub Deadline Assignment

In this section, we present our approach of solving the sub deadline assignment problem. We first discuss our approach for a simple case in which an application has only two services. We then extend our approach to the case for an application with multiple services.

5.3.1 An Application with Two Servers

We first consider the simple case of an application with only two servers. Consider the application $S = \{\lambda, D, R, L(E_1, E_2)\}$ shown in Figure 5.2. Initially, an application request arrives at E_1 following a Poisson process with a rate of λ. If a sub deadline $d_1 \leq D$ is assigned to E_1, not all requests can pass through E_1 due to the deadline violations. We assume that the entire system reaches the stable status and the renege probability is quite small in each server, thus, according to Burke's Theorem [102], the request arrivals for E_2 is assumed to be a Poisson process as well. Assuming that the requests are deleted after the first server at a probability of p_1, then, the arrival rate for the second server E_2 becomes

$$\lambda_2 = (1 - p_1)\lambda_1. \tag{5.1}$$

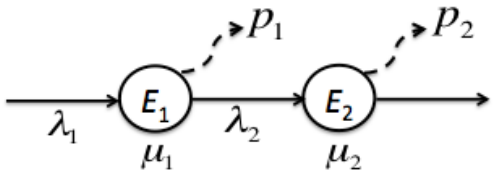


Figure 5.2: A two-tier application with sub deadline assignment.

Similarly, when a sub deadline is assigned to E_2 (i.e. D in this case,) some requests are dropped at a probability of p_2 due to deadline violations. Therefore, the probability for a request to survive both E_1 and E_2 is $1 - (p_1 + (1 - p_1)p_2)$.

Our goal is to determine d_1 such that

$$1 - (p_1 + (1 - p_1)p_2) \geq R. \tag{5.2}$$

The key challenge, however, is to formulate the relations among parameters including $\lambda_1, \lambda_2, p_1, p_2, D$, and R. To this end, we employ the techniques developed for M/M/1 queue with renege [15] to solve our problem. M/M/1 queue with renege is a queueing model that depicts the impatient customers who leave the queue if not fully served within a giving time frame on a server. Specifically, let the arrival rate of requests on a server be λ and the processing rate be μ, and let a customer's maximum waiting length be τ (i.e. the interval between two adjacent sub deadlines in our scenario.) Then the probability that customers renege from the server can be formulated as

$$p = \frac{(1 - \rho)e^{\alpha(\rho-1)}}{1 - \rho e^{\alpha(\rho-1)}} \tag{5.3}$$

where $\rho = \lambda/\mu$ and $\alpha = \mu \cdot \tau$. Similarly, to calculate p_1 in our case, we have

$$p_1 = \frac{(1 - \frac{\lambda_1}{\mu_1})e^{d_1\mu_1(\frac{\lambda_1}{\mu_1}-1)}}{1 - \frac{\lambda_1}{\mu_1}e^{d_1\mu_1(\frac{\lambda_1}{\mu_1}-1)}} \tag{5.4}$$

Based on Equation (6.9), we calculate p_2 by plugging $\lambda = \lambda_2$ and τ_2 into Equation (6.9). However, to formulate the renege probability of E_2, we have to take the

execution status on E_1 into consideration. Since the renege probability on each server will stay small (because of the completion ratio constraint,) we assume that the request arrivals of E_2 still follows a Poisson distribution. Then, another M/M/1 queue can be modeled for E_2. Therefore, let $d_2 = D$, we formulate p_2 as

$$p_2 = \frac{(1 - \frac{\lambda_2}{\mu_2})e^{(D-d_1)\mu_2(\frac{\lambda_2}{\mu_2}-1)}}{1 - \frac{\lambda_2}{\mu_2}e^{(D-d_1)\mu_2(\frac{\lambda_2}{\mu_2}-1)}}. \tag{5.5}$$

Algorithm 7: Sub deadline assignment for a single application with two servers.

Input : **TwoTier**$(\lambda, D, R, E_1, E_2)$
Output: The sub deadline for E_1, i.e. d_1.

```
1 d_1^* = 0;
2 while d_1^* ≤ D do
3     Calculate p_1 and p_2 based on Equations (5.4) and (5.5) using d_1^*;
4     if p_1 + (1 − p_1)p_2 ≤ 1 − R then
5         d_1 = d_1^*;
6         return d_1;
7     d_1^* = d_1^* + δ, where δ is the minimum time interval;
8 print: ("No feasible solution for the given constraints!");
9 exit();
```

The feasible solution of d_1 satisfies Equations (5.2), (5.4), and (5.5). To identify the solution for d_1, we can use sophisticated numerical algorithms when necessary. For ease of presentation, Algorithm 9 employs a simple search method to find the solution of d_1. Note that Algorithm 9 does not always produce a feasible solution of d_1. When λ_1 is extremely large or when μ_1 or μ_2 is too small, there is no possible sub deadline assignment that is able to statistically guarantee the QoS requirements. On the other hand, if Algorithm 9 does produce a solution, we can ensure the probability to successfully fulfill the requests without violating the end-to-end deadlines. This conclusion is summarized in the following theorem.

Theorem 5.3.1. *Given a single application with two servers E_1 and E_2, i.e. $S = \{\lambda, D, R, E_1, E_2\}$, let the processing rates of the two servers be μ_1 and μ_2, respectively. Assume that the relative error when modeling the servers E_1 and E_2 using M/M/1 independent queues is small enough. Then the sub deadline derived from Algorithm 9 can guarantee the deadline meet ratio no smaller than R.*

Proof. Algorithm 9 ensures that the proportion of the total dropped requests is no more than R (line 7). Also, since d_1 is found in such a way that Equations (5.2), (5.4) and (5.5) are all satisfied. According to [15], this ensures that all remaining requests can be processed before d_1 and D when going through the first and second server, respectively. □

In addition, we have made a number of interesting observations that are listed in the following theorem.

Theorem 5.3.2. *For an application S and one of its servers, i.e. E_i, with processing rate of μ_i, let the corresponding request arrival rate be λ_i, the deadline be d_i, and the renege probability at E_i be p_i. Then,*

- *The larger the λ_i, the larger the p_i is;*
- *The larger the μ_i, the smaller the p_i is;*
- *The larger the d_i, the smaller the p_i is.*

Proof. From Equation (6.9), we have

$$\begin{aligned}\frac{\partial p}{\partial \rho} = \{&[(1-\rho)e^{\alpha(\rho-1)}]'[1-\rho e^{\alpha(\rho-1)}]- \\ &[(1-\rho)e^{\alpha(\rho-1)}][1-\rho e^{\alpha(\rho-1)}]'\}/[1-\rho e^{\alpha(\rho-1)}]^2\end{aligned} \tag{5.6}$$

Now let

$$\begin{aligned}F(\rho) \;=\; &[(1-\rho)e^{\alpha(\rho-1)}]'[1-\rho e^{\alpha(\rho-1)}] - \\ &[(1-\rho)e^{\alpha(\rho-1)}][1-\rho e^{\alpha(\rho-1)}]'\end{aligned}$$

Simplify $F(\rho)$ we have

$$\begin{aligned} F(\rho) &= -e^{\alpha(\rho-1)} + [e^{\alpha(\rho-1)}]^2 + (1-\rho)\alpha e^{\alpha(\rho-1)} \\ &= e^{\alpha(\rho-1)}[-1 + e^{\alpha(\rho-1)} + (1-\rho)\alpha] \end{aligned}$$

Let $x = (1-\rho)\alpha$, we have

$$F(\rho) = e^{-x}(-1 + e^{-x} + x) \tag{5.7}$$

According to [3], as long as $x > -1$, $-1 + e^{-x} + x > 0$. Since $x = (1-\rho)\alpha$, and $\rho < 1$, then x is always larger than -1 and Equation (5.7) is positive. Therefore, from Equation (5.6) we have

$$\frac{\partial p}{\partial \rho} > 0. \tag{5.8}$$

To prove the first two bullet points in the theorem, we only need to note that

$$\frac{\partial p}{\partial \lambda_i} = \frac{\partial p}{\partial \rho} \times \frac{\partial \rho}{\partial \lambda_i} > 0. \tag{5.9}$$

and

$$\frac{\partial p}{\partial \mu_i} = \frac{\partial p}{\partial \rho} \times \frac{\partial \rho}{\partial \mu_i} < 0. \tag{5.10}$$

To prove the third bullet point in this theorem, we have

$$\begin{aligned} \frac{\partial p}{\partial d} =& \{[(1-\rho)e^{\mu\cdot d\cdot(\rho-1)}]'[1-\rho e^{\mu\cdot d\cdot(\rho-1)}] \\ & [(1-\rho)e^{\mu\cdot d\cdot(\rho-1)}][1-\rho e^{\mu\cdot d\cdot(\rho-1)}]'\} \\ & /[1-\rho e^{\mu\cdot d\cdot(\rho-1)}]^2 \\ =& \{[(1-\rho)\mu(\rho-1)e^{\mu\cdot d\cdot(\rho-1)}][1-\rho e^{\mu\cdot d\cdot(\rho-1)}] \\ & -\rho\mu(1-\rho)^2[e^{\mu\cdot d\cdot(\rho-1)}]\}/[1-\rho e^{\mu\cdot d\cdot(\rho-1)}]^2 \\ =& -\mu(1-\rho)^2 e^{\mu\cdot d\cdot(\rho-1)}/[1-\rho e^{\mu\cdot d\cdot(\rho-1)}]^2 \\ <& 0 \end{aligned} \tag{5.11}$$

□

5.3.2 An Application with Multiple Servers

We now extend our approach to the case of single application with multiple servers. Similarly, we assume that the system has reached the stable state and each server can be modeled as a M/M/1 queue.

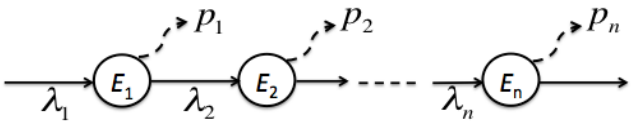


Figure 5.3: Application model with more than two servers.

Take the three-tier application shown in Figure 5.3 as an example. After the first server, assume that there will be p_1 percent of all requests deleted due to the sub deadline violations. The arrival rate for the second server is thus $\lambda_2 = (1 - p_1)\lambda_1$. After the second server, another p_2 percent of the requests among the λ_2 are discarded. Then, only $\lambda_3 = (1 - p_1 - (1 - p_1)p_2)\lambda_1$ are left for the third server. Equivalently, the arrival rate of the requests for tier $n + 1$ from tier n can be formulated as

$$\lambda_{n+1} = (1 - p_n) \cdot \lambda_n. \tag{5.12}$$

The total ratio of requests that have been removed through n ($n \geq 2$) servers can be formulated as

$$\lambda_1 \cdot \{p_1 + \sum_{t_i=2}^{n} (\Pi_{t_j=1}^{t_i-1}(1 - p_{t_j}) \cdot p_{t_i})\} \tag{5.13}$$

Provided that all the remaining requests in the system are able to meet their end-to-end deadlines, in order to satisfy the probabilistic guarantee of R, we only need to require that

$$\lambda_1 \cdot \{p_1 + \sum_{t_i=2}^{n} (\Pi_{t_j=1}^{t_i-1}(1 - p_{t_j}) \cdot p_{t_i})\} \leq (1 - R) \cdot \lambda_1 \tag{5.14}$$

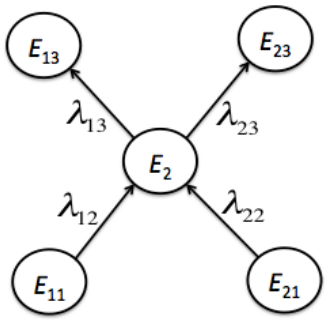


Figure 5.4: Applications with shared tiers.

For an application consists of n servers, based on Equation (6.9) we have

$$p_n = \frac{(1-\frac{\lambda_n}{\mu_n})e^{(d_n-d_{n-1})\mu_n(\frac{\lambda_n}{\mu_n}-1)}}{1-\frac{\lambda_n}{\mu_n}e^{(d_n-d_{n-1})\mu_n(\frac{\lambda_n}{\mu_n}-1)}}. \quad (5.15)$$

Note that in the equation above, p_n is the request removal probability on E_n, and $(d_n - d_{n-1})$ is the allowed processing time of a request on server E_n. We can then formulate the equations recursively for the servers $E_1, E_2, ..., E_n$. The $d_1, d_2, d_3, ..., d_n$ that satisfy Equation (5.14) are the sub deadlines for each server.

5.3.3 Multiple Applications with Shared Servers

Due to issues such as software licenses and resource/cost constraints, different applications usually need to go to the same server for services. We now discuss how to assign sub deadlines for multiple applications in a shared environment.

In this scenario, some tiers that belong to different applications have to share the same function unit (hosted on the same server). The sub deadline calculation is more intricate here. As shown in Figure 5.4, two applications share the same second service tier. Each application has its own QoS required pair $\{D_1, R_1\}$, and $\{D_2, R_2\}$. Algorithm 9 cannot be simply applied to calculate the sub deadlines for the applications. We employ statistical multiplexing to describe the sharing situation. However, it is worth pointing out that our method is not limited to the

statistical multiplexing. Different application requests on the same server share the same processing queue. Since we assume that the arrival rates of both applications on the shared server follow Poisson distribution, the combined workload arrival is still a Poisson process. The processing time of each request on the shared server is exponentially distributed. Therefore, the shared server can be treated as a M/M/1 queue as well. The response time relationship between the two applications and the combined workload can be formulated as following [137]:

$$\frac{1}{U - (\lambda_{12} + \lambda_{22})} = \frac{1}{\mu^*_{12} - \lambda_{12}} = \frac{1}{\mu^*_{22} - \lambda_{22}} \tag{5.16}$$

where U is the shared server's processing rate. λ_{12} and λ_{22} are the arrival rates of the two applications at the second tier. μ^*_{12} and μ^*_{22} are the corresponding simulated processing rates of the two applications under the situation in which applications are processed independently with no resource sharing. The formulations of μ^*_{12} and μ^*_{22} can be derived through simple transformations of Equation (5.16).

$$\begin{aligned} \mu^*_{12} &= U - \lambda_{22} \\ \mu^*_{22} &= U - \lambda_{12} \end{aligned} \tag{5.17}$$

With the derived μ^*_{12} and μ^*_{22} in the shared environment, we can apply Equation (6.9) to obtain the renege probabilities for each of the applications. Due to the fact that they share the same service tier, the sub deadline calculations for the two applications are interdependent. We apply an iterative approach to derive the sub deadlines for each of the applications.

We first assume that no renege happens for application S_2. Then, application S_1 can be treated as a serialized application with a middle tier shared with application S_2. With Equation (5.17), we use the method introduced in the previous section

(Algorithm 9) to calculate the sub deadlines for S_1. Once S_1's sub deadlines are available (e.g. d_{11}), replace the λ_{12} in Equation (5.17) with the derived arrival rate of application S_1 (e.g. λ_{12}^*) in the shared tier using the calculated S_1's sub deadlines (e.g. d_{11}). Then, S_2's sub deadlines are calculated following the same method. These two sub deadline calculation steps for S_1 and S_2 compose an iteration. After each iteration, sub deadline calculations for S_1 and S_2 are conducted again based on the results derived from the previous iteration. The whole procedure stops when the maximum sub deadline difference between two adjacent iterations of both applications are smaller than a pre-defined threshold T (or reach a maximum iteration number.) The details of our approach for multiple applications with shared service tiers is summarized in Algorithm 6.

Algorithm 8: Sub deadline assignment for multiple applications with shared service tiers.

Input : K number of N-tier applications running on M number of servers, $\lambda_i, D_i, R_i, E_1, E_2, \ldots, E_m$. $\forall i \in K$, $\forall m \in M$.
Output: The sub deadline for applications S_1, S_2, ..., S_N.

```
for (i = 1; i ≤ K; i++) do
    for (j = 1; j ≤ N; j++) do
        PreSD_ij = 0;
Initialize Stop to 0;
Initialize Loop_num to 0;
while (!Stop) do
    for (i = 1; i ≤ K; i++) do
        Loop_num = Loop_num + 1;
        Assume no renege for applications S_{i+1}, S_{i+2}, ... S_N;
        Application S_1, ..., S_{i-1} have sub deadlines calculated in the previous
        loops;
        Apply Algorithm 9 for application S_i to get SD_ij;
    if (max[|PreSD_ij − SD_ij|] < T, ∀i ∈ K, ∀j ∈ N) && (Loop_num <= Max_loop_num) then
        Set Stop to 1;
    PreSD_ij = SD_ij;
```

5.4 Simulation Study

In this section, we use simulations with synthetic parameters to investigate the performance of our approach.

We compared our high efficient sub deadline approach ("HESD") with several widely used methods. Acceptance control ("ac"), random deletion ("rd") and First-Come-First-Server ("fifo") have no sub deadline constraints. Then, a local sub deadline calculation method "det" (similar to [49] and [130]) deterministically assigns sub deadlines to servers. Finally, a global sub deadline assignment method "pdf" based on request processing time distribution is employed for comparisons. The summary of the methods is listed as follows:

- **ac**: In the acceptance control method, based on the available computing capacity, a maximum acceptable request arrival rate is provided in order to guarantee that R percent of the accepted requests can be successfully completed before their end-to-end deadlines. If a service request is accepted, it will be processed until it is fully fulfilled.
- **rd**: In the random deletion method, all the request arrivals are accepted into the service chain. However, a bottleneck service tier will be identified. Based on the capacity of the bottleneck tier and the QoS required pair {D, R}, a correspondent amount of requests will be removed randomly from the bottleneck tier.
- **fifo**: The first come first serve method processes each request till completion according to their arrival order.
- **det**: In the deterministic local sub deadline assignment method, each server is assigned a local deadline that is proportional to the server's average response time.

- **pdf**: This is a method that applies the request processing time probability density function (PDF) on each server to find out how likely that the server's follow-up processing can meet the request's end-to-end deadline [136]. We extend their method to a global sub deadline assignment method. The extended version calculates the sub deadlines backward from the exiting point of the application. Thereby, the sub deadline calculated for each server is actually the latest time instance to statistically guarantee that the follow-up servers can finish the request's processing before the request's pre-defined end-to-end deadline.

In order to show the applicability of our method, we implement FIFO and processor sharing (PS) scheduling policies in our simulation with the sub deadline constraints derived by our method.

5.4.1 Single Application without Shared Resources

We tested our method based on a three-tier application. Each tier was assumed to be hosted in an isolated processing unit. No resource or service sharing existed among the tiers. The parameters of the three-tier application were randomly generated. The processing rates of the servers were set to be 120, 115, and 110 requests/second, respectively. The execution times of the requests on the different servers were randomly generated according to each server's processing rate, and the execution times followed exponential distributions. The impacts of different arrival rates and processing rates on the QoS satisfactions were studied. We ran 200000 requests of each application for 5 rounds to average the randomness in each method.

Arrival Rate Effects

We first analyzed the effects of different request arrival rates on the QoS satisfaction.

We set the end-to-end deadline to be $0.4s$ with a completion ratio requirement of 90%, and gradually changed the arrival rate from 99 to 107 request/second with 2 request/second intervals.

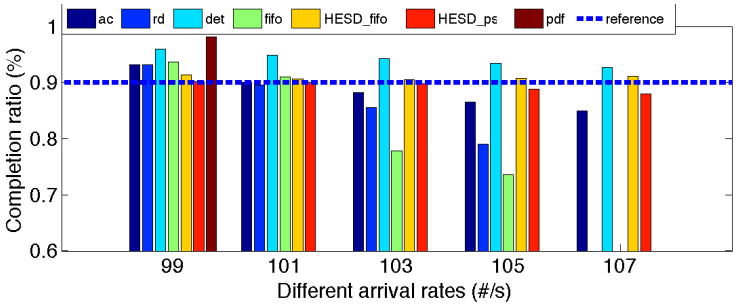


Figure 5.5: Completion ratio comparison.

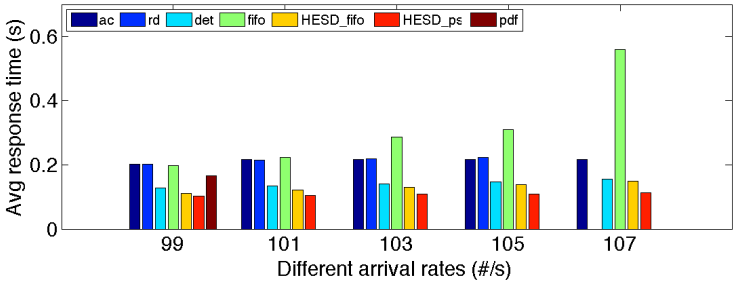


Figure 5.6: Average response time comparison.

Figure 5.5 shows the completion ratios achieved by the methods at different arrival rates. Intuitively, the higher the arrival rate, the worse the completion ratio for most of the methods since the queues are getting more congested with the increasing number of requests. Every request has to experience longer delay in each server in order to be fully served. The results shown in Figure 5.6 clearly supports our inference.

Our proposed method "HESD_fifo" had the lowest average response times while guaranteeing the required completion ratio. Since it is able to statistically guarantee the QoS required pair, its completion ratio kept stable around the required value through out the entire test cases. "HESD_ps" had similar results as "HESD_fifo" when the arrival rate was low. As the arrival rate got higher, its performance degraded slightly. It is interesting to observe that even though "pdf" satisfied the QoS required pair in the first testing case, its derived sub deadlines had less effect as those calculated via our proposed method. The number of completed requests was much larger than the one required to meet the QoS and resulted in significantly higher average response times. Additionally, the "pdf" method is not flexible to the parameter changes. When the arrival rate grew higher, "pdf" struggled and failed to provide sub deadlines for the application. The performance of "det" was surprisingly good in this test. However, it is not able to statistically guarantee the QoS required pair as the resources become stringent. On the contrary, our method is capable of indicating the potential infeasibility when Algorithm 9 could not provide corresponding sub deadlines. "ac" and "rd" satisfied the QoS requirements while the arrival rates stayed low. Nevertheless, as the arrival rate got higher (e.g. 107 requests/second in our case,) in order to guarantee that 90% of the accepted requests could finish before the end-to-end deadline, only around 85% original requests could be accepted for "ac" and 0% for "rd".

Processing Rate Effects

We further studied the processing rate effects on different methods.

The end-to-end deadline was set to be $0.5s$ with a completion ratio requirement of 90%. Arrival rate was 90 requests/second. The processing rates were set to be

120, 115, 110 requests/second initially, and were gradually degraded from 100% to 92% of the full capacities, with 2% degrading intervals.

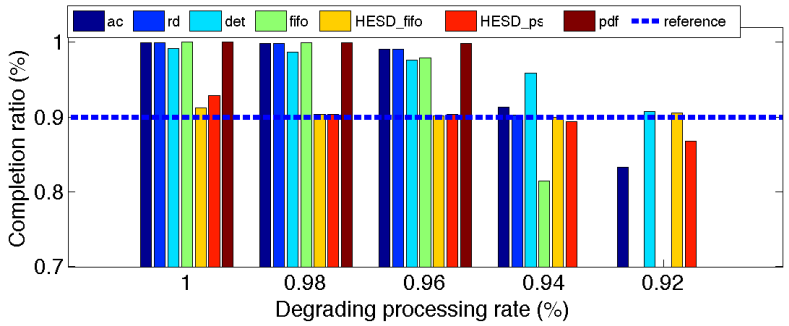


Figure 5.7: Completion ratio comparison.

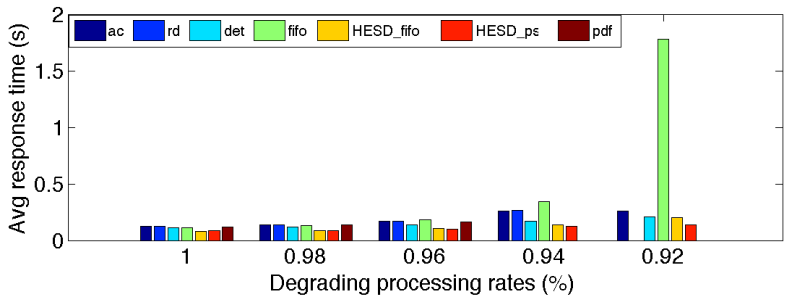


Figure 5.8: Average response time comparison.

Figures 5.7 and 5.8 depict the completion ratios and average response times, respectively. With the full processing capacities, all methods were able to satisfy the QoS requirements. As the processing rate degraded, completion ratios achieved by all the methods except "HESD" dropped. The changes in processing rate did not have significant impacts on the completion ratio in our proposed method ("HESD_ps" degraded slightly as the processing capacity dropped.) However, we can easily tell from the figure that the lower the processing rate, the larger the average response times for all the methods. "pdf" failed to provide sub deadlines for the last two settings.

5.4.2 Multiple Applications with Shared Resources

We finally studied the performance of our proposed method in a shared environment based on the architecture shown in Figure 5.9. Since "ac" and "rd" are similar, "det" has no statistical guarantee, and "fifo" is always the worst on the average response time, we did not include them into the comparisons.

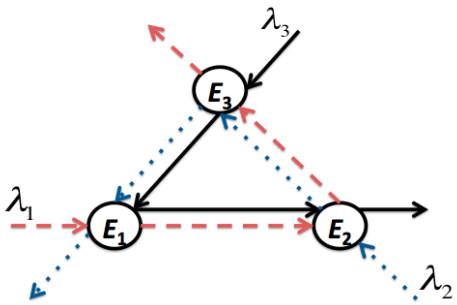


Figure 5.9: Applications with shared servers.

As shown in Figure 5.9, three different applications share three services. The requests of the first application λ_1 enter the system from node E_1. After going through nodes E_2 and E_3, the successfully processed requests leave the system from E_3. Similar to the first application, the requests of the second (third) application enter the system from node E_2 (E_3), and leave the system from nodes E_1 (E_2). We set the processing rate of the three nodes at 350, 360, 370 requests/second, and gradually degraded the processing capacities of the three nodes simultaneously from 96% to 88% with a degrading interval of 2%. The three applications had the same arrival rate at 100 requests/second. The QoS required pairs are {92%, 0.5s}, {90%, 0.4s}, and {92%, 0.5s}, respectively. 10000 number of requests of each application were tested, and the comparisons of the completion ratios and average response times are illustrated in Figure 5.10.

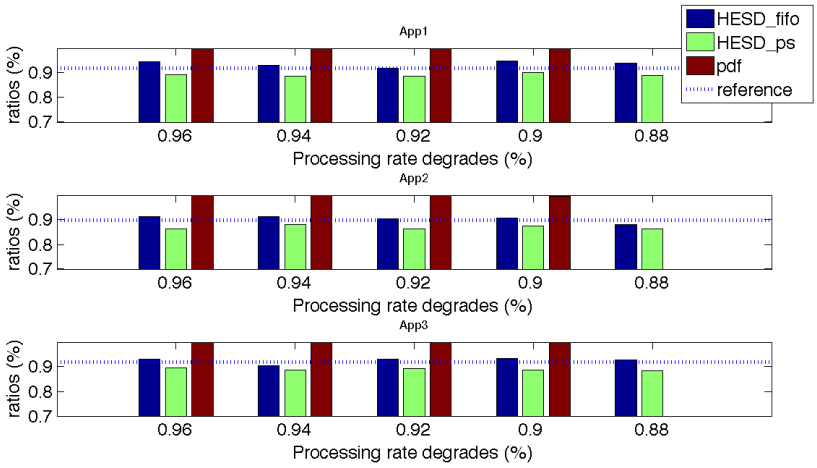
From 5.10(a) we can tell that the "pdf" method had similar performance as in the single application situations. The sub deadlines derived were relatively relaxed

compared to those calculated by "HESD". Not many requests were removed in "pdf". When the processing capacities degraded to 88% in our case, "pdf" failed to assign sub deadlines to the applications subject to the desired QoS required pairs. On the contrary, the completion ratios of "HESD_fifo" kept stable around the required levels at different parameter settings and its average response times were obviously smaller than "pdf" (Figure 5.10(b).) "HESD_ps" had a stable completion ratio as well but failed to reach the required amount (around 4% lower) even though it had the lowest average response times. The average response times reflected a raising trend as the processing capacities degraded in all the methods.

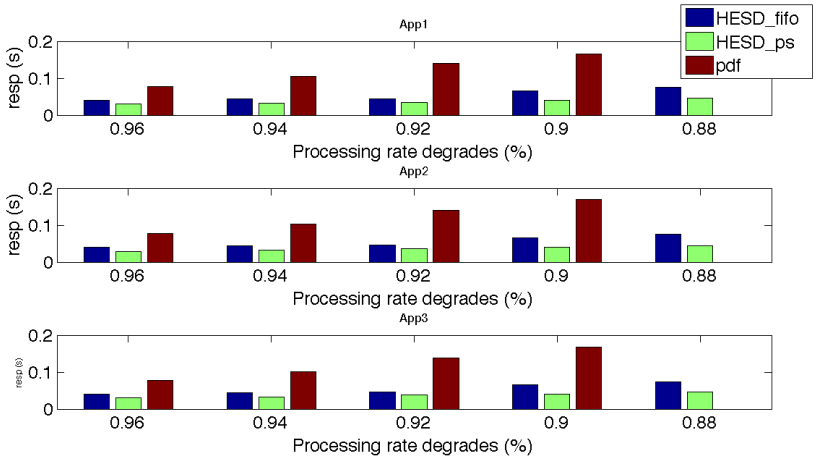
5.5 Conclusion

In this paper, we propose a sub deadline assignment approach for applications with and without resource sharing. We model the multi-tier time-sensitive applications as $M/M/1$ queues with reneges. It is able to statistically guarantee the QoS requirements with high efficiencies by judiciously discarding mischievous failure requests in early stages. Precious computing resources can be used more effectively and efficiently by the promising requests. Our proposed method has the potential to increase the net profit for a service provider.

In order to further explore efficient resource sharing mechanisms, in next chapter we are going to discuss our research on power minimization in an environment where computing resources are shared by the requests of the same service but with different QoS requirements.



(a) Completion ratio comparison.



(b) Average response time comparison.

Figure 5.10: Comparison for multiple applications with shared resources.

CHAPTER 6

POWER MINIMIZATION WITH EFFICIENT RESOURCE SHARING

As the cloud market gets mature, the service level agreements between service providers and customers become more and more fine-grained. Different levels of QoS requirements are defined for various classes of customers. In this chapter, we study the problem of power minimization with guaranteed QoS. Significant power saving can be achieved by efficiently sharing computing and energy resources among requests of the same service but with different QoS requirements.

6.1 Research Problem Introduction

There are extensive studies conducted to improve service provider's power efficiencies (e.g. [116][85][19][119][121]). Server consolidation, for example, has been a common approach to achieve high power/energy efficiency for data centers. The virtualization technology has enabled multiple virtual machine instances be executed on the same physical server. Since a server's power consumption is not exactly proportional to its utilization, and server may consume a significant amount of power even at a very low level of utilization, consolidation helps to increase the utilization of a server and minimize the number of activated physical servers needed. For example, in [116], Verma et al. present consolidation decisions that are made based on the correlations among different workloads. In [85], Mastroianni et al. devise two probability functions to model the effects of the virtual machine allocation and migration. By statistically analyzing the need of a virtual machine allocation or migration, they minimize the number of powered-on servers, and reduce the power consumptions in data centers. Another commonly used approach is to employ the dynamic voltage

and frequency scaling (DVFS) to dynamically adjust the performance and therefore the power dissipation of a server to achieve power efficiency. For example, in [19], Beloglazov et al. propose a two-layer optimization (e.g. global and local layers) to minimize the power consumption in data centers. The local layer monitors each virtual machine's utilization, and dynamically adjusts a physical server's working frequency through dynamic voltage and frequency scaling (DVFS) [119]. The global layer uses bin-packing methods to find a new destination for the virtual machine that has to be migrated. Wang et al. [121] propose an approach for power minimization by determining the appropriate numbers of powered-on servers and their running modes (i.e. execution speeds) for the service requests, which can be modeled using the traditional queuing model.

Power saving techniques usually, if not always, lead to degraded computing performance. However, the quality of service is key to the satisfaction of the client. Service providers usually provide multiple service level agreements (SLAs) regarding different QoS levels. A database may be queried internally by a company's employees or externally by the company's customers. The external customers may further be divided into various priority groups. All of the different "visitors" have their own QoS requirements. The challenges is then how to minimize the power consumption and guarantee these QoS requirements.

With the virtualization technology, it has been a common approach (e.g. [64]) to serve the requests with the same QoS on the same server. Since all requests on the same server has the same QoS requirement, different levels of QoS can be captured by one single variable such as resource provisions (e.g. [116][85][19]), required processing speeds (e.g. [119]) or latency (e.g. [121][64]). While this approach simplifies the resource management problem to guarantee one specified QoS requirements, the overall resource usage can be rather inefficient, as shown later in this paper.

In this chapter, we study the problem of how to minimize the power consumption for a data center with the guaranteed QoS levels. We assume that the data center shall accommodate requests of different kinds of services, as well as requests of the same service but with different QoS requirements. Different from the previous studies that employ an individual server (or virtual machine) for the requests with the same QoS, we develop a method that enables requests of the same service but with different QoS levels to share the same server, and therefore greatly increase the resource utilization. To our best knowledge, this is the first approach proposed that can guarantee different levels of QoS for requests that share the same server. In addition, we devise a novel approach that can judiciously combine various types of requests to the same server, and properly discard potential failure requests in time to minimize the total processing demand and thus the power consumption, while statistically guaranteeing the QoS. Experimental results show that our proposed method greatly outperforms other commonly used methods in terms of both QoS levels and lower power consumptions.

6.2 Preliminary

In this section, we introduce our system model and formulate the problem.

6.2.1 System Model

We assume that a data center provides n different services based on their application purposes, i.e. $S = \{S_1, S_2, ..., S_n\}$. Each service S_i can accommodate different types of requests $\Gamma_i = \{\tau_{i,j}, j = 1, \ldots, r_i\}$, i.e. requests under different service level agreements. Each $\tau_{i,j}$ has its own QoS requirements, denoted as $Q_{i,j}$. We assume that different services need to be hosted on different servers but different types of

Parameters	Definitions
n	total number of services.
$\tau_{i,j}$	the j-type requests in service S_i.
r_i	the total number of request types for service S_i.
$V_{i,k}$	the k-th server that supplies service S_i.
m_i	the total number of servers that support S_i service.
P^d	the dynamic power.
P^s	the static power.
$\lambda_{i,j}$	the arrival rate of $\tau_{i,j}$.
$D_{i,j}$	the deadline for $\tau_{i,j}$.
$R_{i,j}$	the completion ratio for $\tau_{i,j}$.
$Q_{i,j}$	the quality-of-service requirement of $\tau_{i,j}$.
$\mu_{i,j}$	the required processing rate for $\tau_{i,j}$ in order to guarantee $Q_{i,j}$.
$U_{i,k}$	the required processing rate of the k-th server to provide QoS-guaranteed service S_i.
Ω	the pool rate, the sum of all the servers' processing rate in a server pool. $\Omega = \sum_i \sum_k U_{i,k}$.
C	the capacity of the servers.

Table 6.1: Parameter notation.

requests in S_i can be potentially hosted in the same server. We assume that there are n types of servers, i.e. $\{V_1, V_2, ..., V_n\}$, and each type (e.g. V_i) may contain m_i servers (e.g. $V_{i,k}, k = 1, \ldots, m_i$) that support service S_i. Our system model is illustrated in Figure 6.1. In Figure 6.1, τ_{11} τ_{14} share server V_{11}, and τ_{12} τ_{13} share server V_{12}.

We assume that the request arrival patterns follow the Poisson distributions [102], and their response times follow exponential distributions. Different types of requests for the same service may have different arrival times, deadlines, and completion ratio requirements. Specifically, a request is modeled with a 3-tuple, i.e. $\tau_{ij} = \{\lambda_{i,j}, D_{i,j}, R_{i,j}\}$, where

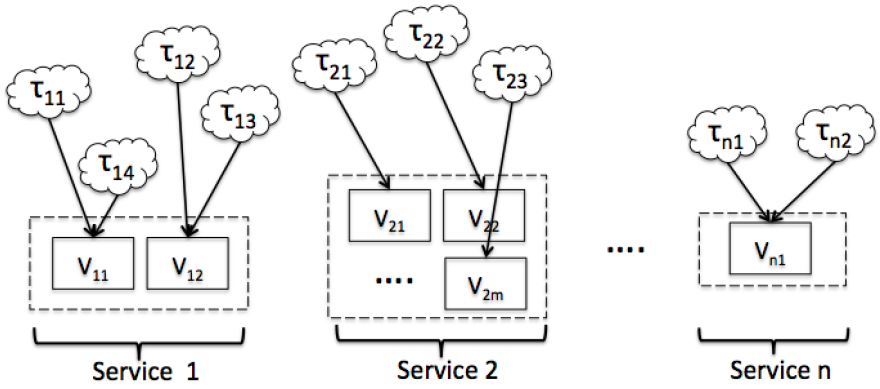


Figure 6.1: System model.

- $\lambda_{i,j}$: the arrival rate of the j-type requests in service S_i.
- $D_{i,j}$: the deadline of the j-type requests in service S_i.
- $R_{i,j}$: the completion ratio requirement of the j-type requests in service S_i.

The $Q_{i,j}$ of $\tau_{i,j}$ is defined by $\{D_{i,j}, R_{i,j}\}$, meaning that at least $R_{i,j}$ percent of the $\tau_{i,j}$ requests have to be served no later than $D_{i,j}$.

A service provider suffers from high power consumptions while providing QoS guaranteed services. We adopt a linear power model to describe a server's power consumption (e.g. [85]), as shown in Equation (6.1):

$$P = P^d\varphi + P^s \tag{6.1}$$

where P^s is the static power, and $P^d\varphi$ is the dynamic power. φ is the utilization of a server, defined as $\varphi = \frac{U}{C}$, where U is the required processing rate of a server, and C is the capacity limit of a server. For ease of reference, we list some commonly used notations in Table 6.1.

6.2.2 Problem Definition

With the system model defined above, the problem we are to address can be formulated as follows.

Problem 6.2.1. *Given service requests* $\Gamma_i = \{\tau_{i,j} : j = 1, ..., r_i\}$ *for* $i = 1, \ldots, n$*, determine the server pool, i.e.* $V = \{V_{i,k} : i = 1, ..., n; k = 1, ..., m_i\}$*, the corresponding processing rate* $U_{i,k}$*, and the allocation of* Γ_i *to* V*, such that the QoS requirements (i.e.* $Q_{i,j}, i = 1, ..., n; j = 1, ..., r_i$*) are guaranteed and the power consumption of the server pool is minimized.*

This problem involves with two intertwined problems: how to judiciously pack service requests to a server and how to determine the proper service rate to minimize the power consumption while guaranteeing the QoS for all types of requests. In what follows, we first introduce three preliminary analyses, and then present our algorithm in details.

6.3 Preliminaries

This section presents three key analyses regarding QoS guarantee, request multiplexing, and request packing. These analyses form the basis of our approach.

6.3.1 Processing Rate Selection for QoS Guarantee

Traditionally, people using M/M/1 queue [102] to represent the request processing procedure [136], as shown in Figure 6.2. Service requests arrive with a rate λ, wait in a queue with an infinite size, and get processed with a rate μ.

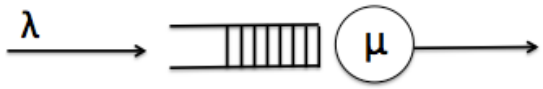


Figure 6.2: M/M/1 queue.

Accordingly, the probability density function (PDF) and the cumulative distribution function (CDF) of the response time are listed below:

$$f(t) = (\mu - \lambda)e^{-(\mu-\lambda)t} \tag{6.2}$$

$$F(t) = 1 - e^{-t\mu(1-\frac{\lambda}{\mu})} \tag{6.3}$$

with a mean response time:

$$E[t] = \frac{1}{\mu - \lambda} \tag{6.4}$$

The q-percentile of the response time t_q (t_q is larger than $q\%$ of all response times) has the following relationship:

$$1 - e^{-t_q\mu(1-\frac{\lambda}{\mu})} = \frac{q}{100} \tag{6.5}$$

with simple transformations, we have:

$$\begin{aligned} t_q &= \frac{1}{\mu - \lambda} ln[\frac{100}{100 - q}] \\ &= E[t] ln[\frac{100}{100 - q}] \end{aligned} \tag{6.6}$$

Given request $\tau_{i,j}$'s arrival rate $\lambda_{i,j}$, deadline $D_{i,j}$, and completion ratio requirement $R_{i,j}$, in order to guarantee $Q_{i,j}$, the required service rate $\mu_{i,j}$ (when $\tau_{i,j}$ is hosted alone) is:

$$\mu_{i,j} = \frac{ln[\frac{1}{1-R_{i,j}}]}{D_{i,j}} + \lambda_{i,j} \tag{6.7}$$

While $\mu_{i,j}$ defined above can guarantee $Q_{i,j}$, as much as $(1 - R_{i,j})\%$ can miss their deadlines and contribute little or no benefit to the application at all. To save

power consumption, it is desirable that we can discard a request if it has a high probability to miss its deadline but save the precious resource for requests that are more likely to be successfully fulfilled. The problem is how to discard these requests without compromising the QoS. To this end, we employ the renege $M/M/1$ queuing model [15], as illustrated in Figure 6.3.

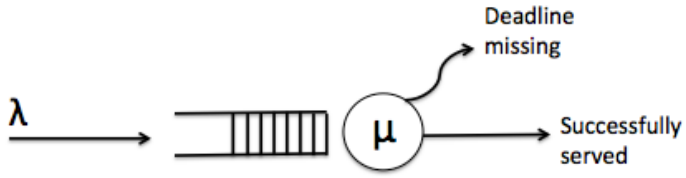


Figure 6.3: M/M/1 with renege.

As shown in Figure 6.3, according to the renege model, each request is associated with a deadline. If a request is not fully served until its deadline, it is removed from the system. According to this model, there exists an interesting relationship among the request's deadline miss probability P_{miss}, the request arrival rate λ, processing rate μ, and deadline D, which can be formulated as:

$$P_{miss} = \frac{(1-\rho)e^{\mu D(\rho-1)}}{1-\rho e^{\mu D(\rho-1)}} \tag{6.8}$$

where $\rho = \frac{\lambda}{\mu}$. Accordingly, for the given $\lambda_{i,j}$, $R_{i,j}$, and $D_{i,j}$ of request $\tau_{i,j}$, we can derive $\mu^*_{i,j}$ that guarantees $Q_{i,j}$:

$$1 - R_{i,j} \geq \frac{(1-\frac{\lambda_{i,j}}{\mu^*_{i,j}})e^{\mu^*_{i,j}D_{i,j}(\frac{\lambda_{i,j}}{\mu^*_{i,j}}-1)}}{1-\frac{\lambda_{i,j}}{\mu^*_{i,j}}e^{\mu^*_{i,j}D_{i,j}(\frac{\lambda_{i,j}}{\mu^*_{i,j}}-1)}} \tag{6.9}$$

It is interesting to note that, the processing rate derived using renege queue $\mu^*_{i,j}$ is smaller than the $\mu_{i,j}$ in Equation (6.7). It means that, with the renege model and judiciously remove the request from the queue, we can guarantee the same QoS with smaller processing rates.

We show the proof of $\mu^* i,j < \mu_{i,j}$ as following:

Proof. In order to proof $\mu^*_{i,j} < \mu_{i,j}$, we substitute $\mu_{i,j}$ derived from Equation (6.7) into the following equation:

$$\begin{aligned} R^* &= 1 - P_{miss} \\ &= 1 - \frac{(1-\frac{\lambda}{\mu})e^{\mu D(\frac{\lambda}{\mu}-1)}}{1-\frac{\lambda}{\mu}e^{\mu D(\frac{\lambda}{\mu}-1)}} \\ &= \frac{1-e^{(\lambda D-\mu D)}}{1-\frac{\lambda}{\mu}e^{(\lambda D-\mu D)}} \end{aligned} \tag{6.10}$$

Then, we have

$$\begin{aligned} R^* &= \frac{1-e^{[\lambda D-(\frac{ln[\frac{1}{1-R}]}{D}+\lambda)D]}}{1-\frac{\lambda}{\mu}e^{[\lambda D-(\frac{ln[\frac{1}{1-R}]}{D}+\lambda)D]}} \\ &= \frac{1-e^{[-ln(\frac{1}{1-R})]}}{1-\frac{\lambda}{\frac{ln[\frac{1}{1-R}]}{D}+\lambda}e^{[-(ln(\frac{1}{1-R})]}} \\ &= \frac{1-(1-R)}{1-\frac{\lambda}{\frac{ln[\frac{1}{1-R}]}{D}+\lambda}(1-R)} \\ &= \frac{R}{1-\frac{\lambda}{\frac{ln[\frac{1}{1-R}]}{D}+\lambda}(1-R)} \end{aligned} \tag{6.11}$$

Since $(1-\frac{\lambda}{\frac{ln[\frac{1}{1-R}]}{D}+\lambda}(1-R)) < 1$, we have:

$$R^* = \frac{R}{1-\frac{\lambda}{\frac{ln[\frac{1}{1-R}]}{D}+\lambda}(1-R)} > R \tag{6.12}$$

Equation (6.12) indicates that using the processing rate μ derived from the traditional method (e.g. Equation (6.7)) generates a completion ratio R^*, which is larger than R. To this end, in order to have the same completion ratio (e.g. R), renege queuing model needs a smaller processing rate, i.e. $\mu^* < \mu$.

□

6.3.2 Request Multiplexing

When service S_i has multiple types of requests, e.g. $\tau_{i,j}$, one common approach is to host each type of requests with one individual server or virtual machine. While this approach simplifies the resource management for guaranteeing specified QoS levels, the efficiency of resource usage can be rather low, not to mention other possible problems, such as the cost of software licenses.

Consider three types of requests of the same service τ_{11}, τ_{12} and τ_{13}. The $R_{i,j}$-th percentile response time $t_{R_{i,j}}$ of $\tau_{i,j}$ can be formulated as:

$$t_{R_{i,j}} = \frac{1}{\mu_{i,j} - \lambda_{i,j}} ln[\frac{1}{1 - R_{i,j}}] \tag{6.13}$$

When hosting each $\tau_{1,j}$ individually on a service pool $V = \{V_{1,1}, V_{1,2}, V_{1,3}\}$, we let the server processing rates as U_{11}, U_{12}, and U_{13}, respectively. Then, to satisfy each $Q_{i,j}$, the processing rates can be calculated as follows:

$$U_{i,j} = \mu_{i,j} = \frac{ln[\frac{1}{1-R_{i,j}}]}{D_{i,j}} + \lambda_{i,j} \tag{6.14}$$

Let us define *the processing rate of the server pool*[1] as the sum of all the servers' required processing rates in a server pool (i.e. V), denoted as $\Omega(V)$. Then, we have:

$$\Omega(V) = \sum_{j=1}^{3} \mu_{1,j} = \sum_{j=1}^{3} [\frac{ln[\frac{1}{1-R_{1,j}}]}{D_{1,j}} + \lambda_{1,j}] \tag{6.15}$$

When the three types of requests are hosted together in a single server $V' = V'_{1,1}$, the processing rate $U'_{1,1}$ of $V'_{1,1}$ has to satisfy the QoS requirements of $\tau_{1,1}$, $\tau_{1,2}$, and $\tau_{1,3}$. For example, assume that the required processing rate of $V'_{1,1}$ is $u^*_{1,1}$ according to $\tau_{1,1}$'s QoS requirements (i.e. $U'_{1,1} = u^*_{1,1}$). Then, based on Equation 6.13, we have

$$\frac{1}{\mu_{1,1} - \lambda_{1,1}} ln[\frac{1}{1 - R_{1,1}}] = \frac{1}{u^*_{1,1} - \sum_{j=1}^{3} \lambda_{1,j}} ln[\frac{1}{1 - R_{1,1}}] \tag{6.16}$$

[1]The processing rate of the server pool, i.e. Ω, is a parameter for reflecting the overall physical server utilization in the server pool.

We transform Equation 6.16 and have:

$$\begin{aligned} U'_{1,1} = u^*_{1,1} &= \mu_{1,1} - \lambda_{1,1} + \sum_{j=1}^{3} \lambda_{1,j} \\ &= \mu_{1,1} + \sum_{j=1, j\neq 1}^{3} \lambda_{1,j} \end{aligned} \tag{6.17}$$

The $U'_{1,1}$ derived by Equation 6.17 guarantees that after the combination, the processing of $\tau_{1,1}$ is the same as it is hosted separately by a single server. Similarly, we can derive the $V'_{1,1}$'s required processing rate $U'_{1,1} = u^*_{1,2}$ ($U'_{1,1} = u^*_{1,3}$, resp.) according to the QoS requirements of $\tau_{1,2}$ ($\tau_{1,3}$, resp.) as follows:

$$U'_{1,1} = u^*_{1,2} = \mu_{1,2} + \sum_{j=1, j\neq 2}^{3} \lambda_{1,j} \tag{6.18}$$

and

$$U'_{1,1} = u^*_{1,3} = \mu_{1,3} + \sum_{j=1, j\neq 3}^{3} \lambda_{1,j} \tag{6.19}$$

Therefore, when $\tau_{1,1}$, $\tau_{1,2}$, and $\tau_{1,3}$ are hosted together in $V'_{1,1}$, in order to satisfy the QoS requirements for all the three types of requests simultaneously, the processing rate $U'_{1,1}$ of $V'_{1,1}$ has to be the maximum among Equations 6.17, 6.18, and 6.19, i.e.:

$$U'_{1,1} = max\{u^*_{1,1}, u^*_{1,2}, u^*_{1,3}\} \tag{6.20}$$

Then, the server pool processing rate is $\Omega(V') = U'_{1,1}$.

We formally formulate the discussion above in Theorem 6.3.1 to conclude our analysis.

Theorem 6.3.1. *For service requests $\tau_{i,1}$, $\tau_{i,2}$, ..., τ_{i,r_i} hosted in a single server $V_{i,1}$, let*

$$u^*_{i,j} = \mu_{i,j} + \sum_{q=1, q\neq j}^{r_i} \lambda_{i,q} \tag{6.21}$$

and let the processing rate of $V_{i,1}$ be $U_{i,1}$. Then $\tau_{i,1}$, $\tau_{i,2}$, ..., τ_{i,r_1} can all meet their QoS if $U_{i,1} \geq max\ \mu^*_{i,j}$.

From Theorem 6.3.1, to meet QoS of $\tau_{1,1}$, $\tau_{1,2}$, and $\tau_{1,3}$ when multiplexing a server, we have $\Omega(V') = \mu_{1,\alpha} + \sum_{j=1,j\neq\alpha}^{3} \lambda_{1,j}$ (assuming $U'_{1,1} = u^*_{1,\alpha}$). Therefore, to compare $\Omega(V)$ and $\Omega(V')$, we have:

$$\frac{\Omega(V)}{\Omega(V')} = \frac{\sum_{j=1}^{3} \mu_{1,j}}{\mu_\alpha + \sum_{j=1,j\neq\alpha}^{3} \lambda_{1,j}} > 1 \tag{6.22}$$

which indicates that to guarantee the same QoS level, request multiplexing requires a smaller processing rate of the server pool (i.e. Ω) than hosting each type of requests on an individual server separately.

6.3.3 Request Packing

From the discussions above, clustering multiple types of requests into the same server helps to improve the resource usage. The question is then how to identify the group of request types for each server that can minimize the server pool processing rate Ω (i.e. $\Omega = \sum_i \sum_k U_{i,k}$) and thus maximize the resource usage and minimize the power consumption. Consider the following example with four types of requests $\tau_1 \ldots \tau_4$:

- $\lambda_1 = 60$, $\mu_1 = 120$.
- $\lambda_2 = 40$, $\mu_2 = 80$.
- $\lambda_3 = 50$, $\mu_3 = 70$.
- $\lambda_4 = 20$, $\mu_4 = 90$.

with μ_i the minimum processing rate to satisfy its QoS when allocated to a server individually. All parameters have the unit of $(request/second)$. With the two request

combinations shown in Figure 6.4, in order to guarantee all the QoS requirements, the required server pool processing rates are $\Omega(V_1, V_2) = 300$ and $\Omega(V_1', V_2') = 280$ (derived based on Equation 6.20). This example clearly shows that different service request allocations can lead to service pools with very different processing rates.

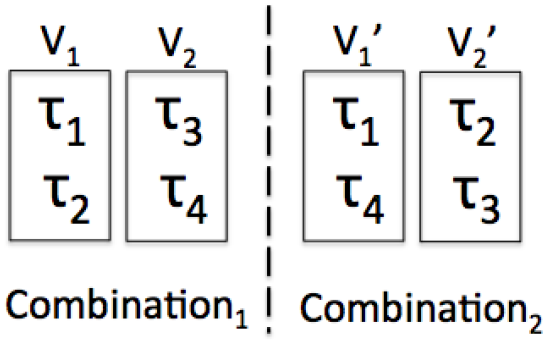


Figure 6.4: Processing rate with different combination.

The service request allocation problem, as we proposed to study here, is $\mathcal{NP}$-hard in nature. Therefore, we focus on the development of effective and efficient heuristic solution for this problem. To this end, we have made a number of interesting observations, which are formulated in the following theorems.

Theorem 6.3.2. *Let $\Gamma_{1,p}$ and $\Gamma_{1,q}$ be two sets of requests of the same service mapped to two servers $V_{1,p}$ and $V_{1,q}$, respectively. Assume that one request type, i.e. $\tau_{1,i} \in \Gamma_{1,p}$, is migrated from $V_{1,p}$ to $V_{1,q}$. Let $U_{1,p}$ ($U_{1,q}$, resp.) be the minimum processing rate for $V_{1,p}$ ($V_{1,q}$, resp.) that guarantees the QoS requirements for requests allocated to the server after $\tau_{1,i}$ is migrated. Then the processing rate for the server pool, i.e. $\Omega(V) = U_{1,p} + U_{1,q}$, is minimized when $\phi_{i,1} = \mu_{i,1} - \lambda_{i,1}$ is the minimum.*

Proof. When migrate request type $\tau_{1,i}$ from $V_{1,p}$ to $V_{1,q}$, there are two cases need to be discussed to derive $\Omega(V)$.

Case 1. $V_{1,q}$ ***is empty:***

We use $\widehat{\Omega}$ and $\widetilde{\Omega}$ to represent the total processing rate required before and after the migration. With no loss of generality, we assume that $\tau_{1,\alpha}$ has the largest ϕ in $V_{1,q}$. Then,

$$\widehat{\Omega} = \{\mu_{1,\alpha} + \sum_{j \in V_{1,p}, j \neq \alpha} \lambda_{1,j}\} \tag{6.23}$$

$$\widetilde{\Omega} = \{\mu_{1,\alpha} + \sum_{j \in V_{1,p}, j \neq (\alpha,i)} \lambda_{1,j}\} + \{\mu_{1,i}\} \tag{6.24}$$

Equation (6.24) can be simply transformed into:

$$\begin{aligned} \widetilde{\Omega} &= \{\mu_{1,\alpha} + \sum_{j \in V_{1,p}, j \neq \alpha} \lambda_{1,j} - \lambda_{1,i}\} + \{\mu_{1,i}\} \\ &= \{\mu_{1,\alpha} + \sum_{j \in V_{1,p}, j \neq \alpha} \lambda_{1,j}\} + \{\mu_{1,i} - \lambda_{1,i}\} \end{aligned} \tag{6.25}$$

Then,

$$\widehat{\Omega} - \widetilde{\Omega} = \mu_{1,i} - \lambda_{1,i} = \phi_{1,i} \tag{6.26}$$

From Equation (6.26) we can see that Ω increases with the minimum amount only when $\tau_{1,i}$ has the smallest ϕ. Therefore, if $V_{1,p}$ has to migrate $\tau_{1,i}$ to $V_{1,q}$, Ω has the minimum increase by migrating from the request type that has the smallest ϕ.

Case 2. V_2 ***is not empty****:*

With no loss of generality, assume $\tau_{1,\alpha}$ and $\tau_{1,\theta}$ are the request types that has the largest ϕ in $V_{1,p}$ and $V_{1,q}$, respectively. $\tau_{1,i}$ is the migrating request. Then, the case can be further divided into two sub-situations to discuss. We use $\widehat{\Omega}$ and $\widetilde{\Omega}$ to indicate the total required processing rate before and after the migration.

- $\phi_{1,i} < \phi_{1,\theta}$

$$\widehat{\Omega} = \{\mu_{1,\alpha} + \sum_{j \in V_{1,p}, p \neq \alpha} \lambda_{1,j}\} + \{\mu_{1,\theta} + \sum_{k \in V_{1,k}, k \neq \theta} \lambda_{1,k}\} \tag{6.27}$$

$$\begin{aligned}\widetilde{\Omega} &= \{\mu_{1,\alpha} + \sum_{j\in V_{1,p}, j\neq(\alpha,i)} \lambda_{1,j}\} + \{\mu_{1,\theta} + \sum_{k\in V_{1,q}, k\neq\theta} \lambda_{1,k} + \lambda_{1,i}\} \\ &= \{\mu_{1,\alpha} + \sum_{j\in V_{1,p}, j\neq\alpha} \lambda_{1,j} - \lambda_{1,i}\} + \{\mu_{1,\theta} + \sum_{k\in V_{1,q}, k\neq\theta} \lambda_{1,k} + \lambda_{1,i}\} \\ &= \{\mu_{1,\alpha} + \sum_{j\in V_{1,p}, j\neq\alpha} \lambda_{1,j}\} + \{\mu_{1,\theta} + \sum_{k\in V_{1,q}, k\neq\theta} \lambda_{1,k}\}\end{aligned} \tag{6.28}$$

Equations (6.27) and (6.28) show that if $\tau_{1,i}$ has a smaller ϕ than $\tau_{1,\theta}$, the migration causes no change in Ω.

- $\phi_{1,i} \geq \phi_{1,\theta}$

$$\begin{aligned}\widetilde{\Omega} &= \{\mu_{1,\alpha} + \sum_{j\in V_{1,p}, j\neq(\alpha,i)} \lambda_{1,j}\} + \{\mu_{1,i} + \sum_{k\in V_{1,q}, k\neq(\theta,i)} \lambda_{1,k} + \lambda_{1,\theta}\} \\ &= \{\mu_{1,\alpha} + \sum_{j\in V_{1,p}, j\neq\alpha} \lambda_{1,j} - \lambda_{1,i}\} + \{\mu_{1,i} + \sum_{k\in V_{1,q}, k\neq i} \lambda_{1,k}\} \\ &= \{\mu_{1,\alpha} + \sum_{j\in V_{1,p}, j\neq\alpha} \lambda_{1,j}\} + \{\mu_{1,i} - \lambda_{1,i} + \sum_{k\in V_{1,q}, k\neq i} \lambda_{1,k}\}\end{aligned} \tag{6.29}$$

From Equation (6.29) we can tell that as long as τ_i has the minimum ϕ, Ω has the minimum increase.

□

Theorem 6.3.2 implies that when migrating requests from one server to another, selecting the requests with the smallest ϕ helps reduce the overall server pool processing rate. In addition, we have the following theorem.

Theorem 6.3.3. *Let $V_1 = \{V_{1,1}, V_{1,2}, \ldots, V_{1,p}\}$ and $V_1' = \{V_{1,1}', V_{1,2}', \ldots, V_{1,q}'\}$ be two server pools that host the same set of requests Γ_1. Both V_1 and V_1' satisfy Γ_1's QoS requirements. Then $\Omega(V_1) \geq \Omega(V_1')$ if $p \geq q$.*

Proof. Without any loss of generality, we prove our claim with an example in which V_1 has three servers $(V_{1,1}, \ldots, V_{1,3})$ hosting R types of requests $(\{\tau_\sigma, \ldots, \tau_{\sigma+m}\}$ in

$V_{1,1}$, $\{\tau_p, \ldots, \tau_{p+n}\}$ in $V_{1,2}$, and $\{\tau_\alpha, \ldots, \tau_{\alpha+\gamma}\}$ in $V_{1,3}$), where $(m+1)+(n+1)+(\gamma+1)=R$. We further assume that $\phi_\sigma > \phi_p > \phi_\alpha$. Then we have:

$$\widehat{\Omega} = \{\mu_\sigma + \sum_{j \in V_{1,1}, j \neq \sigma} \lambda_j\} + \{\mu_p + \sum_{q \in V_{1,2}, q \neq p} \lambda_q\} + \{\mu_\alpha + \sum_{\theta \in V_{1,3}, \theta \neq \alpha} \lambda_\theta\} \tag{6.30}$$

If $V_{1,1}$ and $V_{1,2}$ have enough residual capacities to host parts of the request set in $V_{1,3}$, then we can migrate x types of requests to $V_{1,1}$ and y types of requests to $V_{1,2}$ ($x+y=\gamma+1,\ 0 \leq x \leq \gamma+1, 0 \leq y \leq \gamma+1$). Then, the server pool becomes $V_1' = \{V_{1,1}', V_{1,2}'\}$.

The total required processing rate $\widetilde{\Omega}$ after migration is formulated as following:

$$\widetilde{\Omega} = \{\mu_\sigma + \sum_{j \in V_{1,1}', j \neq \sigma} \lambda_j + \sum_{\theta_{1,1}' \in V_{1,1}', \theta_{1,1}'=1}^{x} \lambda_{\theta_{1,1}'}\} + \{\mu_p + \sum_{q \in V_{1,2}', q \neq p} \lambda_q + \sum_{\theta_{1,2}' \in V_{1,2}', \theta_{1,2}'=1}^{y} \lambda_{\theta_{1,2}'}\} \tag{6.31}$$

The difference between $\widehat{\Omega}$ and $\widetilde{\Omega}$ is:

$$\begin{aligned}
\widehat{\Omega} - \widetilde{\Omega} &= [\{\mu_\sigma + \sum_{j \in V_{1,1}, j \neq \sigma} \lambda_j\} + \{\mu_p + \sum_{q \in V_{1,2}, q \neq p} \lambda_q\} + \{\mu_\alpha + \sum_{\theta \in V_{1,3}, \theta \neq \alpha} \lambda_\theta\}] \\
&- [\{\mu_i + \sum_{j \in V_{1,1}', j \neq \sigma} \lambda_j + \sum_{\theta_{1,1}' \in V_{1,1}', \theta_{1,1}'=1}^{x} \lambda_{\theta_{1,1}'}\} + \{\mu_p + \sum_{q \in V_{1,2}', q \neq p} \lambda_q + \sum_{\theta_{1,2}' \in V_{1,2}', \theta_{1,2}'=1}^{y} \lambda_{\theta_{1,2}'}\}] \\
&= \mu_\alpha + \sum_{\theta \in V_{1,3}, \theta \neq \alpha}^{\gamma+1} \lambda_\theta - \sum_{\theta_{1,1}' \in V_{1,1}', \theta_{1,1}'=1}^{x} \lambda_{\theta_{1,1}'} - \sum_{\theta_{1,2}' \in V_{1,2}', \theta_{1,2}'=1}^{y} \lambda_{\theta_{1,2}'} \\
&= \mu_\alpha - \lambda_\alpha > 0
\end{aligned} \tag{6.32}$$

Equation (6.32) indicating that hosting the same set of requests with smaller number of server achieves smaller total required processing rate. □

Theorem 6.3.3 indicates that for a server pool that holds the same set of requests, the smaller the number of server is, the smaller the processing rate it needs to guarantee all the QoS requirements. Therefore, to improve the resource usage and minimize the power consumption, it is beneficial to reduce the number of servers as best as we can.

6.4 The Request Allocation Algorithm

We are now ready to discuss our approach for power consumption minimization in data centers with guaranteed QoS.

Since the overall power consumption depends on both the processing rate of the server pool and their static power consumptions (see Equation (6.1)), to solve Problem 6.2.1, we need to minimize the number of employed servers and their utilizations. As discussed above, when multiple requests are hosted in the same server, the processing rate of the server pool can be greatly reduced. Also, when requests with long latency and high possibility of missing its deadline can be judiciously removed, the demanded server processing rate can also be reduced. Therefore, in our approach, we adopt the renege model to enable the guarantee of different QoS levels for requests that share the same server, and also save the power consumption by expunging the requests that have high probability to fail. In the meantime, Theorems 6.3.2 and 6.3.3 provide valuable insights for the development of heuristic to minimize the number of servers as well as the processing rate of the server pool. The detailed algorithm is illustrated in Algorithm 9.

As shown in Algorithm 9, the required processing rate μ_j for each type of requests $\Gamma = \{\tau_j, j = 1, \ldots, r\}$ is calculated using the renege model (line 1). We then sort all requests based on $\phi_j, j = 1, \ldots, r$ in a decreasing order, with $\phi_j = \mu_j - \lambda_j$, where λ_j is the request τ_j's arrival rate, and μ_j is the required processing rate of a server to satisfy Q_j if τ_j is hosted alone (line 2). Then, we employ the traditional first-fit bin-packing algorithm to pack requests in the list to servers with a capacity (i.e. the maximum processing rate) of C (line 3 to line 17). The reason to order the requests according to the value of ϕ is because, according to Theorem 6.3.2, when a server is full and new server needs to be allocated, allocating the requests with the smallest

Algorithm 9: Request allocation

Input : A set of requests $\Gamma = \{\tau_j, j = 1, \ldots, r\}$ in service S, each τ_j has corresponding λ_j and Q_j ($\{D_j, R_j\}$). A set of servers $V = \{V_k, k = 1, \ldots, m\}$ with the same capacity C serve the requests in service S.

Output: The requests allocation.

```
1  Calculate each μ_j to satisfy Q_j based on Equation (6.9);
2  dif_vect ← Sort the φ = μ − λ in the decreasing order;
3  for All requests τ_j, j = 1, 2, ..., r do
4      for All servers V_k, k = 1, 2, ..., m do
5          Add τ_j into server V_k;
6          Calculate V_k's required processing rate U_k based on Equation (6.20);
7          if U_k ≤ C then
8              Remove τ_j from dif_vect;
9              Break the loop and pack the next request;
10         else
11             Remove τ_j from the current server V_k;
12     if Request τ_j did not fit in any available server then
13         Open a new server and pack request τ_j;
```

value of ϕ helps reduce the overall processing rate of the server pool. Furthermore, the bin-packing algorithm minimizes the "bins", or the number of servers. It thus also minimizes the processing rate of the server pool as shown in Theorem 6.3.3. Therefore, Algorithm 1 can potentially achieve high resource usage and thus power efficiency.

6.5 Experimental Results

In this section, we use simulations to study the properties of our proposed approach.

We assume that requests for different services must be assigned to different servers. In our experiments, all requests are of the same service but with different QoS levels. The arrival rate of each request type was randomly generated following a uniform distribution in a range between 120 $requests/second$ and 20 $requests/second$. The deadline of each request type was also randomly generated following a uniform distribution in a range between $100ms$ and $80ms$. The completion ratio was fixed at 95%. The server's capacity was set to be 250 $requests/second$.

We compare our approach to four other approaches. All of them apply renege model to derive the required processing rate for each type of requests.

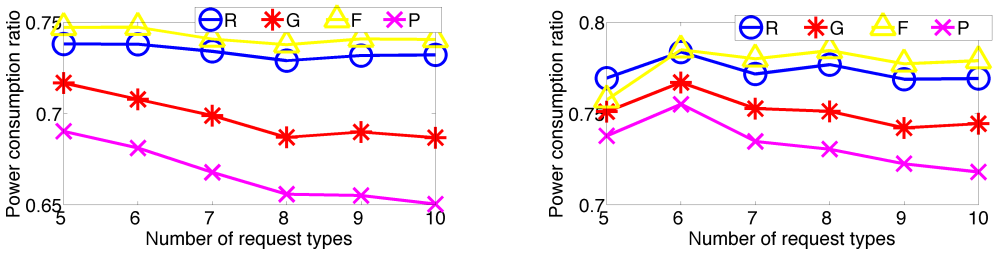
- **split**: denoted as "S", the traditional method that each request type is hosted in an isolated server [116][35].
- **random**: denoted as "R", a method that combines the requests randomly using request multiplexing.
- **first-fit-decreasing**: denoted as "F", a widely employed bin-packing method [86][4]. Requests are combined using request multiplexing in a decreasing order of $\mu_{i,j}$.

- **greedy**: denoted as "G", a greedy packing approach. It packs all the requests into one server, e.g. V_1, using request multiplexing. If the required processing rate of the server exceeds the server's capacity, the request type that has the smallest ϕ will be migrated into a new server, e.g. V_2. The migration continues until V_1's required processing rate U_1 is smaller than or equals to the capacity C, i.e. $U_1 \leq C$. Then, the same procedure works in V_2 and stops when all the servers, e.g. $V_1, \ldots, V_m$ satisfy their capacity constraints.
- **Proposed**: denoted as "P", our proposed method.

6.5.1 Service Utilization Effect

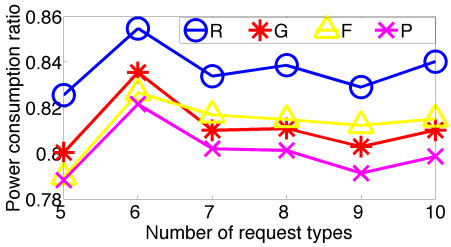
We first conducted a set of tests to study the power saving performance by different approaches under different request utilizations, i.e. $\rho_{i,j} = \frac{\lambda_{i,j}}{\mu_{i,j}}$. From Equation (6.9), when the arrival rate is a constant value, the smaller the deadline, the higher the required processing rate is. Therefore, as the deadline $D_{i,j}$ reduces, the processing rate $\mu_{i,j}$ increases, and then the service utilization increases. We therefore varied request utilization by changing the interval from which we randomly picked the deadlines. Specifically, we modified the upper bound and lower bound of the interval simultaneously, from $100ms$ to $40ms$ and from $80ms$ to $20ms$, respectively, with an interval length of $20ms$. The test cases in each intervals was tested a thousand times. The averaged power consumption results were collected, normalized to that by the approach "split", and are shown in Figure 6.5.

As shown in Figures 6.5(a) to 6.5(d), our proposed method (denoted as "P") has the lowest power consumption ratio across all experiment settings. The figures clearly illustrate the fact that the power consumption ratio decreases as the utilization becomes higher (which indicates an increasing resource saving). When

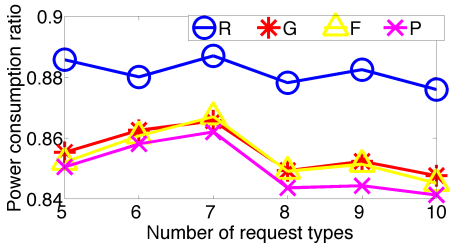


(a) Deadline range $[20, 40]$.

(b) Deadline range $[40, 60]$.



(c) Deadline range $[60, 80]$.



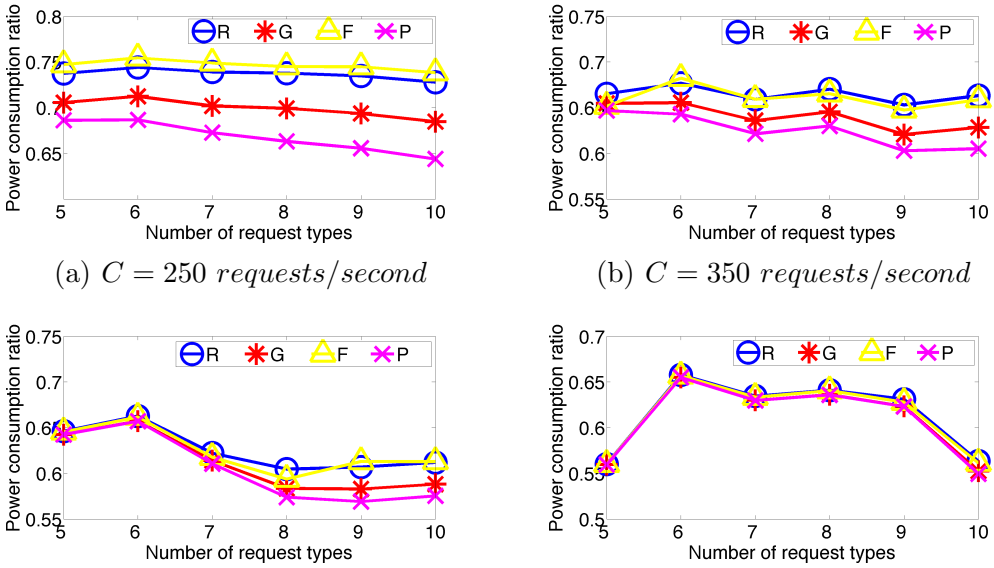
(d) Deadline range $[80, 100]$.

Figure 6.5: Power saving performance for requests with different deadline ranges [lower bound, upper bound].

the number of request types is 10, the power consumption ratio achieved by our proposed approach increased from 65% to 85% as the utilization increases (e.g. the deadline range changes from $[20ms, 40ms]$ to $[80ms, 100ms]$). Since the "greedy" approach uses a similar allocation strategy that is based on the sorting result of $\phi_{i,j}$, its power saving performance is worse than but very close to the results achieved by our proposed approach. The "first-fit-decreasing" (ffd) derives the request allocations according to the sorting of $\mu_{i,j}$, which does not have direct effects on the processing rate of the server pool (i.e. Ω) and thus has a poor power saving performance, especially when the utilization is high (as shown in Figure 6.5(a)). However, when the utilization is low (as shown in Figure 6.5(d)), "ffd" has a similar performance as our proposed approach because the sorting of $\phi_{i,j}$ is similar to the sorting of $\mu_{i,j}$.

6.5.2 Capacity Effect

Next, we study the server's capacity effect on power savings. In this experiment, we kept the request deadline's upper and lower bounds at $40ms$ and $20ms$, and gradually changed the server's capacity from 250 $requests/second$ to 550 $requests/second$ with an interval length of 100 $requests/second$. The completion ratios for all requests were set at 95%. We ran each setting a thousand times.



(a) $C = 250$ $requests/second$ (b) $C = 350$ $requests/second$

(c) $C = 450$ $requests/second$ (d) $C = 550$ $requests/second$

Figure 6.6: Server's capacity effects on power saving.

The averaged results are shown in Figure 6.6. The server's capacity increases from 250 $requests/second$ in Figure 6.6(a) to 550 $requests/second$ in Figure 6.6(d). Our proposed approach outperforms the others throughout all of the test settings. However, the improvement of our proposed approach diminishes as the server's capacity increases. This is because when the server's capacity increases, more types of requests can be hosted together in the same server. All the approaches that employ request multiplexing can achieve much better energy saving results than that when each request type is hosted solely on an individual server. If the capacity

is large enough that all the requests can be packed into a single server, then all the approaches that employ request multiplexing have the same results.

6.5.3 Completion Ratio and Average Response Time

Finally, we compare the request completion ratios and average response times achieved by our proposed approach and the split method. The test was conducted with 5 request types. The deadline's upper and lower bounds were set to $40ms$ and $20ms$. The arrival rate's upper and lower bounds were set as 120 $requests/second$ and 20 $requests/second$. Completion ratios were set to 95% for all requests. Our proposed method provided a combination, indicating that $\{\tau_4, \tau_1, \tau_5\}$ and $\{\tau_3, \tau_2\}$ were hosted separately in two servers. Each request type had $10,000$ requests. The averaged simulation results are shown in Figure 6.7.

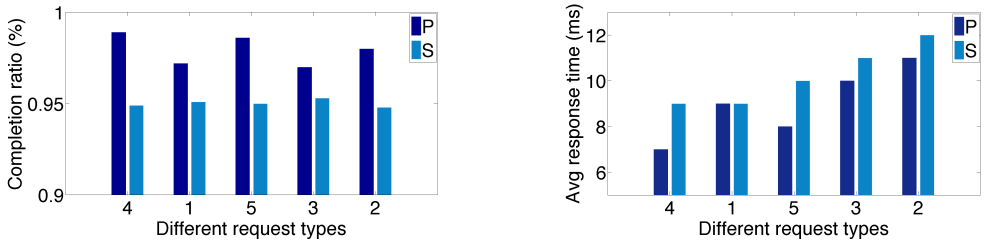


Figure 6.7: Completion ratios and average response times.

As reflected in Figure 6.7(a), our proposed method not only guarantees the completion ratios (actually our method achieves even higher completion ratios), but also obviously reduces the average response times as shown in Figure 6.7(b). The reason for this phenomenal improvement is because of the request multiplexing technique, which efficiently utilizes computing resources among different types of requests.

6.6 Conclusion

The expansion of web services in both scope and scale make efficient resource management extremely challenging for a service provider's sustainable development. In this paper, we propose an approach based on request multiplexing and renege queueing techniques to judiciously combine different types of requests for each server and discard potential failure requests in time. A data center's power consumption can be reduced significantly without compromising QoS satisfactions.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this chapter, we first summarize our contributions presented in this dissertation. Then, we discuss the possible directions for our future research work.

7.1 Summary

Today, cloud computing has permeated many aspects of our daily lives. When people are enjoying the convenience brought by cloud services, service providers are suffering from tough resource management challenges (e.g. high operational costs, delay-sensitive service requests, huge heterogeneities from both hardware and customers, large dynamics in service demands and electricity prices, etc.).

In this dissertation, we studied the problem of delay-sensitive cloud service scheduling for the sustainable development of cloud computing.

We first developed a set of profit and penalty-aware request scheduling methods on a single server to maximize a service provider's net profit. Two non-preemptive scheduling methods were proposed based on the "opportunity cost" concept and the speculated execution order. Considering the fact that preemptive techniques are more responsive to higher priority requests, and more schedulable than their non-preemptive counterparts, we then extended the non-preemptive methods to a preemptive one. Our methods carefully chose the ready task to run, judiciously discard pending requests or abort task executions, and cautiously preempt current running tasks. They achieve better performance than traditional scheduling methods. Among the three newly developed methods, speculated order based scheduling has the best performance. Our preemptive method is 50% worse than the speculated order based approach but outperforms the opportunity cost based method.

Then, in order to accommodate the distributed characteristic of cloud computing, we studied the request scheduling problem for distributed data centers in a multi-electricity-market environment. We first devised a request mapping and scheduling approach to judiciously allocate requests to their corresponding data centers and decide the execution sequence for the allocated requests. Considering the huge service demands in today's cloud environment, we extended our method by employing the queuing model. Compared to a static request dispatching method, our approach achieves 130% more net profits in the best case.

Next, we focused on the request scheduling problem for multi-tier cloud services. We developed an algorithm to assign sub deadlines to each service tier in order to statistically guarantee the QoS requirements. Our method is able to discover potential failure requests and remove them at early stages. The precious computing resources can be saved for other requests that are more likely to be successfully fulfilled. Compared to other traditional methods (e.g. fifo), our method can statistically guarantee the QoS while achieving a maximum 80% shorter average response time. We further extended our method to accommodate the case with resource sharing among different services. With the statistically guaranteed QoS, our method achieves a maximum 50% shorter average response time than a traditional sub deadline calculation method.

Finally, we studied the power minimization problem in an environment where computing resources are shared by different types of service requests. By employing the renege queuing model and the statistic multiplexing technique, the efficiency of resource utilization is significantly improved. Our solution achieves a 55% maximum power savings compared to traditional methods with statistically guaranteed QoS.

7.2 Future Work

As cloud computing evolves, the cloud environment is becoming more and more complex. The complexities come from two perspectives: computing infrastructure and service patterns.

The continuously rising service demands require a tremendous amount of computing nodes (e.g. physical servers) to host the huge workload. Additionally, with the rapid advancement in hardware technologies, more heterogeneities will be presented among the various computing nodes. The gigantic computing infrastructures and their heterogeneities pour a significant amount of complexities into cloud environments. Even though the probability of a hardware or software component failure during its lifetime (typically 3-5 years in industry) can be small, it could be magnified across all computing nodes in a complex environment. At such a large scale, a hardware/software failure is "the norm rather than an exception" [117].

Additionally, today's cloud services are usually hosted by multi-tier service architectures. Since the service tiers have close inter-tier interactions, the success of a cloud service heavily depends on the execution in every single service tier. A malfunction in a sub-service leads to a failure of the whole service architecture. The intricate inter-tier dependencies aggravate the complexities in the cloud.

Therefore, it is extremely hard to avoid service outages because of hardware/software or operator failures.

7.2.1 Cost of Failures

Service downtimes attract more and more attention. Significant amount of effort has already been devoted into the fault tolerance design for service providers [81][80][117][47]. However, the problem is still one of the immediate challenges that

service providers are facing. In 2010, 95% of data centers experienced service downtimes. The number dropped to 91% in 2013. Similarly, the length of downtime incidents dropped from 97 minutes in 2010 to 86 minutes in 2013. Nonetheless, the cost of downtime per incident raised from $\$5,211$ (per minute) in 2010 to $\$8,023$ (per minute) in 2013 [118]. The growing trend in downtime cost asks for more effective and efficient fault tolerance techniques to guarantee a service provider's sustainable development in a competitive business environment.

7.2.2 Failure Classifying

The failures that lead to service downtimes can be classified into three groups in general: hardware failures, software failures, and operator failures.

- Hardware Failure

  Among the downtimes experienced by data centers, around 33% were caused by IT equipment failures [118]. Provided by Vishwanath et al. [117], most (78%) of the faults were found in hard disks, followed by a few failures (5%) due to raid controllers. Memory accounted for only 3%. Thus, hard disks are the most dominant reason behind server failures.

- Software failure

  Software failures take different proportions in different applications. They may cause around 30% of service downtimes in software oriented applications and only around 8% in hardware focused applications [92]. Many malfunctions may lead to software failures, among which hangs and loops are two main types. Sub-service components become unresponsive during a hang waiting for a resource that never becomes available, or during a loop endlessly repeating series of instructions [80].

- Operator Failure

  Patterson et al. [92] conducted two surveys to analyze the causes of service downtimes, based on the U.S. Public Switched Telephone Network (PSTN) and three Internet sites. According to the two survey results, around 50% of service outages are due to operator faults. The researcher claims that as long as human operators are involved in system operations, they will make errors, even when they truly know what to do.

The three failure groups discussed above cover most of the service outage reasons. Even though people are able to categorize the failures with much more fine-grained metrics, it is impossible for us to avoid the failures. To this end, we would like to study the problem of how to effectively and efficiently back up the delay-sensitive cloud services in order to weaken the failure impacts on QoS satisfaction.

7.2.3 Research Problem

There are two possible perspectives from which we can tackle the failure impacts: check pointing and redundancy.

- Check Pointing

  Check pointing is a technique used to help efficiently handle failures by inserting checking points into an application's execution. At each checking point, all execution information will be stored in case of a failure in the near future. Once an application encounters a failure, its execution rolls back to the nearest checking point for retrieving all the stored execution records. The execution resumes from the nearest checking point instead of rolling back to the beginning of the execution. Therefore, the processing efficiencies (in terms of short response time or low power consumption) can be potentially improved.

However, the efficiency improvement relies on the number of inserted checking points. A large number of checking points helps reduce roll back recovering time. On the contrary, it increases the computing overhead because of the execution status validations, and the increased overhead can be a burden for delay-sensitive services.

- Redundancy

  Redundancy is a technique to tackle failures by adding redundant components into the system. In virtualized environments, redundant virtual machines are configured in anticipation of host server failures [81]. A widely employed approach is the $N+M$ redundant configuration, which prepares M redundancies and guarantees that N components work when any M components fail. The redundancies can be either active backups (i.e. work simultaneously with the original copies) or passive backups (i.e. work after the original copies fail).

  Both redundant backup configurations help QoS satisfactions when failures occur. However, the price that service providers have to pay is the space and energy costs (e.g. more storage spaces and power will be consumed). Under certain space/power budgets, how to efficiently design fault tolerance configuration for QoS satisfaction is not a trivial problem.

In order to make our designed resource management methods accommodate failures, we intend to incorporate the fault tolerance concept into our methods from the two perspectives mentioned above. By balancing the trade-off between QoS satisfaction and time or space complexity, we can help service providers improve their service efficiencies and thus achieve sustainable developments.

BIBLIOGRAPHY

[1] Adobe media server. http://www.adobe.com/products/amazon-web-services.html.

[2] Aimms. http://www.aimms.com/.

[3] V. R. Aiyar. On the exponential inequalities and the exponential function. *The Mathematical Gazette*, 4(61):pp. 8–12, 1907.

[4] Y. Ajiro and A. Tanaka. Improving packing algorithms for server consolidation. In *Int. CMG Conference*, pages 399–406. Computer Measurement Group, 2007.

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

[6] Amazon web services. http://aws.amazon.com.

[7] M. Andrews. Probabilistic end-to-end delay bounds for earliest deadline first scheduling. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 603–612. IEEE, 2000.

[8] H. Arabnejad and J. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *Parallel and Distributed Systems, IEEE Transactions on*, 25(3):682–694, March 2014.

[9] D. Ardagna, M. Trubian, and L. Zhang. Sla based resource allocation policies in autonomic environments. *Journal of Parallel and Distributed Computing*, 67(3):259–270, 2007.

[10] M. Arlitt and T. Jin. 1998 world cup web site access logs, August 1998.

[11] Arm-the architecture for the digital world. http://www.arm.com/.

[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. *UC Berkeley*, 2009.

[13] I. D. Baev, W. M. Meleis, and A. E. Eichenberger. Algorithms for total weighted completion time scheduling. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 852–853, 1999.

[14] J. Baliga, R. Ayre, K. Hinton, and R. Tucker. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 99(1):149 –167, jan. 2011.

[15] D. Y. Barrer. Queuing with impatient customers and ordered service. *Operations Research*, 5(5):pp. 650–656, 1957.

[16] Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, and L. Stougie. Multiprocessor scheduling with rejection. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, 28-30 January 1996, Atlanta, Georgia.*, pages 95–103, 1996.

[17] S. K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1-2):9–20, 2006.

[18] A. Beloglazov. *Energy-efficient management of virtual machines in data centers for cloud computing.* PhD thesis, Department of Computing and Information Systems, The University of Melbourne, February 2013.

[19] A. Beloglazov and R. Buyya. Energy efficient resource management in virtualized cloud data centers. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 826–831. IEEE Computer Society, 2010.

[20] Z. Bodie, R. Merton, and D. Cleeton. *Financial Economics.* Prentice Hall, New York, 2008.

[21] C. Boeres, A. Lima, and V. E. Rebello. Hybrid task scheduling: Integrating static and dynamic heuristics. In *Computer Architecture and High Performance Computing, 2003. Proceedings. 15th Symposium on*, pages 199–206. IEEE, 2003.

[22] K. G. Brill. The economic meltdown of moore's law and the green data center. In *Proceedings of the 21th Large Installation System Administration Conference, LISA 2007, Dallas, Texas, USA, November 11-16, 2007*, 2007.

[23] M. Cardosa, M. R. Korupolu, and A. Singh. Shares and utilities based power consolidation in virtualized server environments. In *Integrated Network Management*, pages 327–334. IEEE, 2009.

[24] F. Casati and M.-C. Shan. Definition, execution, analysis, and optimization of composite e-services. *IEEE Data Eng. Bull.*, 24(1):29–34, 2001.

[25] J. S. Chase, D. C. Anderson, P. N. Thakar, and A. M. Vahdat. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles SOSP*, pages 103–116, 2001.

[26] M. Chen, H. Zhang, Y.-Y. Su, X. Wang, G. Jiang, and K. Yoshihira. Effective vm sizing in virtualized data centers. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pages 594–601. IEEE, 2011.

[27] S. Chen, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and W. H. Sanders. Blackbox prediction of the impact of dvfs on end-to-end performance of multitier systems. *SIGMETRICS Performance Evaluation Review*, 37(4):59–63, 2010.

[28] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing server energy and operational costs in hosting centers. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 303–314, New York, NY, USA, 2005. ACM.

[29] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 30, 2002.

[30] R. K. Clark. *Scheduling dependent real-time activities*. PhD thesis, Carnegie Mellon University, 1990.

[31] Demands of the new data center: Consolidate, virtualize, cool. http://www.crn.com/features/data-center/229301293/demands-of-the-new-data-center-consolidate-virtualize-cool.htm.

[32] Dropbox. https://www.dropbox.com/home.

[33] Recounting ec2 one year later. http://www.jackofallclouds.com/2010/12/recounting-ec2/.

[34] Facts and stats of world's largest data centers. http://storageservers.wordpress.com/2013/07/17/facts-and-stats-of-worlds-largest-data-centers/.

[35] E. Feller, L. Rilling, and C. Morin. Energy-aware ant colony based workload placement in clouds. In *Grid Computing (GRID), 2011 12th IEEE/ACM International Conference on*, pages 26–33, Sept 2011.

[36] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4):397–413, 1993.

[37] S. K. Garg, C. S. Yeo, A. Anandasivam, and R. Buyya. Environment-conscious scheduling of hpc applications on distributed cloud-oriented data centers. *J. Parallel Distrib. Comput.*, 71(6):732–749, 2011.

[38] A. Ghiasi and R. Baca. Overview of largest data centers. http://www.ieee802.org/3/bs/public/14_05/ghiasi_3bs_01b_0514.pdf.

[39] Glpk. http://www.gnu.org/software/glpk/.

[40] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Capacity management and demand prediction for next generation data centers. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 43 –50, july 2007.

[41] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, IISWC '07, pages 171–180, Washington, DC, USA, 2007. IEEE Computer Society.

[42] T. Gonzalez and S. Sahni. Preemptive scheduling of uniform processor systems. *Journal of the ACM (JACM)*, 25(1):92–101, 1978.

[43] Google app engine. https://developers.google.com/appengine/docs/whatis googleappengine.

[44] Google docs. http://www.google.com/google-d-s/b1.html.

[45] Report: Google uses about $900,000$ servers. http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers/.

[46] Microsoft and ibm chase amazon while google falls off the pace. https://www.srgresearch.com/articles/microsoft-and-ibm-chase-amazon-while-google-falls-pace.

[47] A. Haeberlen. A case for the accountable cloud. *Operating Systems Review*, 44(2):52–57, 2010.

[48] J. L. Hellerstein. Google cluster data, Jan 2010. Posted at http://googleresearch.blogspot.com/2010/01/google-cluster-data.html.

[49] S. Hong, T. Chantem, and X. S. Hu. Meeting end-to-end deadlines through distributed local deadline assignments. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 183–192. IEEE, 2011.

[50] Forget application response time "standards"–it's all about the human reaction. http://www.riverbed.com/blogs/human-reaction-drives-application-response-time-standards.html.

[51] Ilog. http://www-01.ibm.com/software/websphere/ilog/.

[52] Intel atom processor. http://www.intel.com/content/www/us/en/processors/atom/atom-processor.html.

[53] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 160–169, 2004.

[54] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 111–119, New York, NY, USA, 1987. ACM.

[55] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[56] R. Jejurikar. Energy aware non-preemptive scheduling for hard real-time systems. In *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 21–30. IEEE, 2005.

[57] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *RTSS*, volume 85, pages 112–122, 1985.

[58] A. Kamthe and S.-Y. Lee. A stochastic approach to estimating earliest start times of nodes for scheduling dags on heterogeneous distributed computing systems. *Cluster Computing*, 14(4):377–395, 2011.

[59] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 428–437. IEEE, 1993.

[60] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *Parallel and Distributed Systems, IEEE Transactions on*, 8(12):1268–1274, 1997.

[61] L. Kleinrock. *Queueing systems, volume I: theory*. Wiley Interscience, 1975.

[62] K. Konstanteli, T. Cucinotta, K. Psychas, and T. Varvarigou. Admission control for elastic cloud services. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 41–48. IEEE, 2012.

[63] H. Kuno. Surveying the e-services technical landscape. In *Advanced Issues of E-Commerce and Web-Based Information Systems, 2000. WECWIS 2000. Second International Workshop on*, pages 94–101. IEEE, 2000.

[64] P. Lama and X. Zhou. Efficient server provisioning with end-to-end delay guarantee on multi-tier clusters. In *Quality of Service, 2009. IWQoS. 17th International Workshop on*, pages 1–9. IEEE, 2009.

[65] K. Le, R. Bianchini, J. Zhang, Y. Jaluria, J. Meng, and T. D. Nguyen. Reducing electricity cost through virtual machine placement in high performance computing clouds. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 22:1–22:12, New York, NY, USA, 2011. ACM.

[66] K. Le, O. Bilgir, R. Bianchini, M. Martonosi, and T. D. Nguyen. Managing the cost, energy consumption, and carbon footprint of internet services. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '10, pages 357–358, New York, NY, USA, 2010. ACM.

[67] G. Lee. *Resource Allocation and Scheduling in Heterogeneous Cloud Environments*. PhD thesis, EECS Department, University of California, Berkeley, May 2012.

[68] Y. C. Lee, C. Wang, A. Y. Zomaya, and B. B. Zhou. Profit-driven service request scheduling in clouds. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 15–24. IEEE Computer Society, 2010.

[69] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171. IEEE, 1989.

[70] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms*. PhD thesis, Virginia Polytechnic Institute and State University, 2004.

[71] P. Li, H. Wu, B. Ravindran, and E. Jensen. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. *Computers, IEEE Transactions on*, 55(4):454–469, April 2006.

[72] W. Li, K. M. Kavi, and R. Akl. A non-preemptive scheduling algorithm for soft real-time systems. *Computers & Electrical Engineering*, 33(1):12–29, 2007.

[73] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. *IEEE/ACM Trans. Netw.*, 21(5):1378–1391, Oct. 2013.

[74] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[75] J. W.-S. Liu. *Real-time systems.* Prentice Hall, 2000.

[76] X. Liu, J. Heo, L. Sha, and X. Zhu. Adaptive control of multi-tiered web applications using queueing predictor. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 106–114, 2006.

[77] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. L. Andrew. Greening geographical load balancing. In *Proceedings of the ACM SIGMETRICS joint international conference on measurement and modeling of computer systems*, SIGMETRICS '11, pages 233–244, New York, NY, USA, 2011. ACM.

[78] Z. Liu, M. S. Squillante, and J. L. Wolf. On maximizing service-level-agreement profits. In *EC '01: Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 213–223, New York, NY, USA, 2001. ACM.

[79] C. D. Locke. *Best-effort decision making for real-time scheduling*. PhD thesis, Carnegie Mellon University, 1986.

[80] S. Loveland, E. M. Dow, F. LeFevre, D. Beyer, and P. F. Chan. Leveraging virtualization to optimize high-availability system configurations. *IBM Systems Journal*, 47(4):591–604, 2008.

[81] F. Machida, M. Kawato, and Y. Maeno. Redundant virtual machine placement for fault-tolerant consolidated server clusters. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 32–39. IEEE, 2010.

[82] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 49:1–49:12, New York, NY, USA, 2011. ACM.

[83] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. *SIGARCH Comput. Archit. News*, 35(2):46–56, June 2007.

[84] M. Marzolla, O. Babaoglu, and F. Panzieri. Server consolidation in clouds through gossiping. In *Proceedings of the 2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, WOWMOM '11, pages 1–6, Washington, DC, USA, 2011. IEEE Computer Society.

[85] C. Mastroianni, M. Meo, and G. Papuzzo. Probabilistic consolidation of virtual machines in self-organizing cloud data centers. *IEEE T. Cloud Computing*, 1(2):215–228, 2013.

[86] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th international conference on Autonomic computing*, pages 11–20. ACM, 2010.

[87] A. K.-L. Mok. Fundamental Design Problems of Distributed Systems for the Hard–Real–Time Environment. Technical Report MIT-LCS-TR-297, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1983. Ph.D. Thesis.

[88] R. R. Muntz and E. G. Coffman Jr. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the ACM (JACM)*, 17(2):324–338, 1970.

[89] S. Muppala and X. Zhou. Cosac: Coordinated session-based admission control for multi-tier internet applications. In *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, pages 1–6, 2009.

[90] A. Nair, Jayakrishnan Wierman and B. Zwart. Provisioning of large scale systems: The interplay between network effects and strategic behavior in the user base. under submission.

[91] H. Oh and S. Ha. A static scheduling heuristic for heterogeneous processors. In *Euro-Par'96 Parallel Processing*, pages 573–577. Springer, 1996.

[92] D. A. Patterson. Recovery oriented computing: A new research agenda for a new century. In *HPCA*, page 247. IEEE Computer Society, 2002.

[93] Applications performance equals response time, not resource utilization. http://www.virtualizationpractice.com/applications-performance-equals-response-time-not-resource-utilization-9916/.

[94] F. I. Popovici and J. Wilkes. Profitable services in an uncertain world. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 36, 2005.

[95] Powering a google search. http://googleblog.blogspot.com/2009/01/powering-google-search.html.

[96] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the electric bill for internet-scale systems. *ACM SIGCOMM Computer Communication Review*, 39(4):123–134, 2009.

[97] Rackspace cloud media hosting. http://www.rackspace.com/cloud/media/.

[98] L. Rao, X. Liu, M. Ilic, and J. Liu. Mec-idc: joint load balancing and power control for distributed internet data centers. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 188–197, New York, NY, USA, 2010. ACM.

[99] L. Rao, X. Liu, L. Xie, and W. Liu. Minimizing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 1145–1153, Piscataway, NJ, USA, 2010. IEEE Press.

[100] J. Regehr and J. A. Stankovic. Hls: A framework for composing soft real-time schedulers. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 3–14. IEEE, 2001.

[101] S. Ren, Y. He, and F. Xu. Provably-efficient job scheduling for energy and fairness in geographically distributed data centers. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 22 –31, june 2012.

[102] T. G. Robertazzi. *Computer Networks and Systems: Queueing Theory and Performance Evaluation*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[103] Amazon's cloud on track for $2bn in revenue in 2013. http://www.theregister.co.uk/2013/04/26/aws_revenue_analysis/.

[104] O. Sarood and L. V. Kale. A 'cool' load balancer for parallel applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 21:1–21:11, New York, NY, USA, 2011. ACM.

[105] L. Sha, T. F. Abdelzaher, K.-E. rzn, A. Cervin, T. P. Baker, A. Burns, G. C. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[106] K. Shin and P. Ramanathan. Real-Time Computing: A New Discipline of Computer Science and Engineering. *Proc. IEEE*, 82(1):6–24, Jan. 1994.

[107] Y. Song, Y. Zhang, Y. Sun, and W. Shi. Utility analysis for internet-oriented server consolidation in vm-based data centers. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[108] B. Speitkamp and M. Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE T. Services Computing*, 3(4):266–278, 2010.

[109] 7 statistics you didn't know about cloud computing. http://blog.nskinc.com/IT-Services-Boston/bid/118077/7-Statistics-You-Didn-t-Know-About-Cloud-Computing.

[110] C. Subramanian, A. Vasan, and A. Sivasubramaniam. Reducing data center power with server consolidation: Approximation and evaluation. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[111] Synergy research group. https://www.srgresearch.com/.

[112] X. Tang, K. Li, G. Liao, K. Fang, and F. Wu. A stochastic scheduling algorithm for precedence constrained tasks on grid. *Future Gener. Comput. Syst.*, 27(8):1083–1091, Oct. 2011.

[113] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.

[114] Aws case study: Naughty dog. http://aws.amazon.com/solutions/case-studies/naughty-dog/.

[115] R. Urgaonkar, U. C. Kozat, K. Igarashi, and M. J. Neely. Dynamic resource allocation and power management in virtualized data centers. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 479–486. IEEE, 2010.

[116] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 28–28, Berkeley, CA, USA, 2009. USENIX Association.

[117] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204. ACM, 2010.

[118] Voice of Uptime. How Downtime Impacts the Bottom Line 2014. http://www.stratus.com/blog/uptime/?p=1369.

[119] G. von Laszewski, L. Wang, A. J. Younge, and X. He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *CLUSTER*, pages 1–10. IEEE, 2009.

[120] P. Wang, Y. Qi, X. Liu, Y. Chen, and X. Zhong. Power management in heterogeneous multi-tier web clusters. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 385 –394, sept. 2010.

[121] S. Wang, W. Munawar, J. Liu, J.-J. Chen, and X. Liu. Power-saving design for server farms with response time percentile guarantees. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 273–284. IEEE, 2012.

[122] X. Wang, D. Lan, X. Fang, M. Ye, and Y. Chen. A resource management framework for multi-tier service delivery in autonomic virtualized environments. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 310–316, 2008.

[123] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, pages 328–335. IEEE, 1999.

[124] G. Welch and G. Bishop. An introduction to the kalman filter, 1995.

[125] What is cloud computing? http://aws.amazon.com/what-is-cloud-computing/.

[126] H. Wu. *Energy-Efficient utility Accrual Real-Time Scheduling*. PhD thesis, Virginia Polytechnic Institute and State University, 2005.

[127] H. Wu, U. Balli, B. Ravindran, and E. D. Jensen. Utility accrual real-time scheduling under variable cost functions. In *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*, pages 213–219. IEEE, 2005.

[128] H. Wu, B. Ravindran, and E. D. Jensen. Energy-efficient, utility accrual real-time scheduling under the unimodal arbitrary arrival model. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 474–479. IEEE Computer Society, 2005.

[129] J. Yu and R. Buyya. A budget constrained scheduling of workflow applications on utility grids using genetic algorithms. In *Workflows in Support of Large-Scale Science, 2006. WORKS '06. Workshop on*, pages 1 –10, june 2006.

[130] J. Yu and R. Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Sci. Program.*, 14(3,4):217–230, Dec. 2006.

[131] J. Yu, R. Buyya, and C. K. Tham. Cost-based scheduling of scientific workflow application on utility grids. In *Proceedings of the First International Conference on e-Science and Grid Computing*, E-SCIENCE '05, pages 140–147, Washington, DC, USA, 2005. IEEE Computer Society.

[132] Y. Yu, S. Ren, N. Chen, and X. Wang. Profit and penalty aware (pp-aware) scheduling for tasks with variable task execution time. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 334–339, New York, NY, USA, 2010. ACM.

[133] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. *ACM SIGOPS Operating Systems Review*, 37(5):149–163, 2003.

[134] L. Zhang and D. Ardagna. Sla based profit optimization in autonomic computing systems. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 173–182, New York, NY, USA, 2004. ACM.

[135] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Trans. Computers*, 36(8):949–960, 1987.

[136] M. Zivkovic, J. Bosman, J. L. Van den Berg, R. Van der Mei, H. Meeuwissen, and R. Nunez-Queija. Dynamic profit optimization of composite web services with slas. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pages 1–6, Dec 2011.

[137] M. Zukerman. Introduction to queueing theory and stochastic teletraffic models. *arXiv preprint arXiv:1307.2968*, 2013.

VITA

SHUO LIU

2006	B.S., Electrical Engineering Beihang University Beijing, China
2009	M.S., Electrical Engineering Utah State University Utah, USA
2014	Ph.D., Electrical Engineering Florida International University Florida, USA

PUBLICATIONS

Shuo Liu, Soamar Homsi, Ming Fan, Shaolei Ren, Gang Quan, Shangping Ren, (2015). *Power Minimization for Data Center with Guaranteed QoS*, 2015 Design Automation and Test in Europe (DATE). (accepted)

Shuo Liu, Soamar Homsi, Ming Fan, Shaolei Ren, Gang Quan, Shangping Ren, (2014). *Scheduling Time-Sensitive Multi-Tier Services with Probabilistic Performance Guarantee*, The 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS). (accepted)

Ming Fan, Qiushi Han, Shuo Liu, Shaolei Ren, Gang Quan, Shangping Ren, (2014). *Enhanced Fixed-Priority Real-Time Scheduling on Multi-core Platforms by Exploiting Task Period Relationship*, Journal of Software Systems. (accepted)

Ming Fan, Qiushi Han, Shuo Liu, Gang Quan, (2014). *On-Line Reliability-Aware Dynamic Power Management for Real-Time Systems*, 16th International Symposium on Quality Electronic Design (ISQED). (accepted)

Ming Fan, Shuo Liu, Gang Quan, (2014). *Energy Calculation For Periodic Multi-Core Scheduling in System Thermal Steady-State With Consideration Of Leakage-AndTemperatureDependency*, Journal of Supercomputing. (under review)

Shuo Liu, Shaolei Ren, Gang Quan, Ming Zhao, and Shangping Ren, (2013). *Profit Aware Load Balancing for Distributed Cloud Data Centers*, 2013 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 611–622.

Shuo Liu, Gang Quan, Shangping Ren, (2012). *On-Line Real-Time Service-Oriented Task Scheduling Using TUF*, ISRN Software Engineering, doi:10.5402/2012/681985.

Shuo Liu, Gang Quan, Shangping Ren, (2011). *On-Line Real-Time Service Allocation and Scheduling for Distributed Data Centers*, 2011 8th IEEE International Conference on Services Computing (SCC), 528 – 535.

Shuo Liu, Gang Quan, Shangping Ren, (2011). *On-Line Preemptive Scheduling of Real-Time Services with Profit and Penalty*, 2011 Proceedings of IEEE Southeastcon, 287–292.

Shuo Liu, Gang Quan, Shangping Ren, (2011). *On-Line Scheduling of Real-Time Services with Profit and Penalty*, 2011 26th ACM Symposium on Applied Computing (SAC), 1476–1481.

Shuo Liu, Gang Quan, Shangping Ren, (2010). *On-Line Scheduling of Real-Time Services for Cloud Computing*, 2010 6th IEEE World Congress on Services (Services), 459–464.