Efficient Resource Management for Cloud Computing Environments

by

Qi Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Cloud computing has recently gained popularity as a cost-effective model for hosting and delivering services over the Internet. In a cloud computing environment, a cloud provider packages its physical resources in data centers into virtual resources and offers them to service providers using a pay-as-you-go pricing model. Meanwhile, a service provider uses the rented virtual resources to host its services. This large-scale multi-tenant architecture of cloud computing systems raises key challenges regarding how data centers resources should be controlled and managed by both service and cloud providers.

This thesis addresses several key challenges pertaining to resource management in cloud environments. From the perspective of service providers, we address the problem of selecting appropriate data centers for service hosting with consideration of resource price, service quality as well as dynamic reconfiguration costs. From the perspective of cloud providers, as it has been reported that workload in real data centers can be typically divided into server-based applications and MapReduce applications with different performance and scheduling criteria, we provide separate resource management solutions for each type of workloads. For server-based applications, we provide a dynamic capacity provisioning scheme that dynamically adjusts the number of active servers to achieve the best trade-off between energy savings and scheduling delay, while considering heterogeneous resource characteristics of both workload and physical machines. For MapReduce applications, we first analyzed task run-time resource consumption of a large variety of MapReduce jobs and discovered it can vary significantly over-time, depending on the phase the task is currently executing. We then present a novel scheduling algorithm that controls task execution at the level of phases with the aim of improving both job running time and resource utilization. Through detailed simulations and experiments using real cloud clusters, we have found our proposed solutions achieve substantial gain compared to current state-of-art resource management solutions, and therefore have strong implications in the design of real cloud resource management systems in practice.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Raouf Boutaba, for his attentive supervision and support throughout my studies. Over the years, he has provided me the freedom, guidance the encouragement to pursue my research goals. His work ethics and professionalism has always impressed me, and will continue to inspire me in my future career.

Secondly, I would like to thank Dr. Joseph L. Hellerstein for being my mentor during my internship at Google. He gave me great inspiration and taught me valuable research skills. My internship experience at Google was quite rewarding, as I gained valuable knowledge on how real cloud data centers operate.

Third, I am grateful my to dissertation committee members: Professor Martin Karsten, Paul Ward, Bernard Wong and Samuel Pierre, for their patience in reviewing and providing valuable comments on this dissertation.

I would like to thank my former and current colleagues at University of Waterloo: Jin Xiao, Maxwell Young, Eren Gürses, Mohamad Faten Zhani, Mosharaf Chowdhury, Muntasir Raihan Rahman, Carol Fung, Khalid Alsubhi, Md. Faizul Bari, Md Golam Rabbani and Arup Raton Roy who has made our lab a pleasant place to work in. I would also like to thank Jakub Truszkowski, Qu Chen, Yuke Yang, Huangdong Meng and many other friends for making life in University of Waterloo a memorable experience. Furthermore, I would also like to thank Quanyan Zhu from University of Illinois at Urbana Champaign for the helpful discussions and collaboration we had over the years, which has helped in the shaping the direction of my research.

Last, but not least, I would like to thank my parents for their continuous encouragement and support that have helped me to attain this achievement.

Dedication

To my parents, Anmin Zhang and Zhaolan Yang, for their love and patience.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the rapid development of Internet technologies, recent years have witnessed the rise of large-scale online service applications such as web search, social networking and content delivery. As these applications often require significant storage, processing and networking capacities, it is a critical challenge to design large-scale computing infrastructures for supporting these applications in a cost-effective manner.

Cloud computing has recently emerged as a computing model for addressing this challenge. Specifically, cloud computing aims at harnessing massive resource capacities of data centers to support applications in a scalable, flexible, reliable and dynamic manner. In a cloud computing environment, the traditional role of service providers is divided into two [126]: The *Cloud providers* (also known as infrastructure providers [22]) package physical resources (e.g. servers) into virtual resources (e.g. Virtual Machines (VMs)), and allocate them to service providers; The *service providers*, on the other hand, use the allocated virtual resources to run their service applications. Compared with traditional private service hosting models, cloud computing offers the following salient features that make it attractive to business owners:

- *Multi-tenancy*: The physical resources in data centers are shared among multiple service providers. By separating virtual resources from physical resources, cloud providers can assign and reassign virtual resources to service providers according to their demand. This also provides management flexibilities to improve resource utilization and to minimize operational cost.

- *Elasticity*: A service provider can rapidly acquire and release resources to scale-up and down in accordance with demand fluctuation and system conditions. As a result,

service providers are given the flexibility to manage their service infrastructures in a dynamic manner, which leads to improved application performance and to reduced operational cost.

- *Ubiquitous network access*: Cloud computing promotes the use of the Internet as the infrastructure for service delivery. As a result, cloud services are easily accessible through a variety of devices with Internet connections.

- *Efficient-support for data intensive applications*: Cloud computing frameworks such as MapReduce [1] and Dryad [121] are designed for large-scale data-intensive computations in a data center environment. By dividing a large computational workload into small and independent tasks, these frameworks can leverage massive processing and storage capacities of data centers to process large volumes of data in very short time.

- *Usage-based pricing*: Cloud computing adopts utility-based pricing models. Specifically, service providers are charged according to their actual resource usage. This model is attractive to service providers as it lowers service operating cost by charging customers on a per-use basis.

Despite the success of Cloud computing and its apparent benefits, the rapid growth in scale and complexity of todays Cloud services and infrastructures also imposes significant challenges on the design of the underlying resource management systems. First, a resource management system must be capable of managing extremely large-scale workloads consisting of vast numbers of applications with diverse performance objectives. Moreover, as cloud computing also advocates on-demand resource provisioning, cloud resource management must be done in a dynamic fashion, according to both demand fluctuation and system conditions. Second, service providers and cloud providers generally have different resource management objectives. The goal of a service provider is to find a resource allocation that best meets its service level objectives (SLOs), while minimizing the total resource usage cost. In contrast, the goal of a cloud provider is to maximize the total resource utilization (which typically translates into revenue from leasing resources to service providers), while minimizing the total operational cost. As service providers and cloud providers face different resource management concerns, separate solution techniques should be developed for each of them. Driven by its importance and difficulties, cloud resource management has attracted significant attention from the research community, with many research topics being actively pursued in recent years [126].

The overarching theme of this thesis is the development of resource management techniques that address several key inefficiencies of today's cloud resource management sys-

tems. In the remainder of this chapter, we will first provide an overview of cloud computing systems, including their architectures and typical workload characteristics. Then we will provide a detailed description of the resource management challenges in cloud computing environments. Lastly, we conclude this chapter by summarizing the main contributions of this dissertation.

1.1 Cloud Computing System Architecture

The general architecture of a cloud computing system is shown in Figure 1.1. Specifically, the physical resources in each data center are organized in racks of physical machines. The racks are connected through data center networks, which offer high bandwidth, low latency connections between physical machines. In order to reduce capital investments, cloud providers often build their data centers with large quantities of commodity machines and switches, as opposed to expensive high-end equipment. However, as commodity equipment may become outdated over-time, it is necessary to upgrade data centers with new equipment once a few years. As a consequence, modern cloud data centers often consist of multiple generations of physical machines and switches with heterogenous processing, storage and networking capacities.

In a cloud computing environment, an application can be divided into one or more components (e.g. servers and processes) that run in separate *virtual containers*. Each virtual container occupies a certain amount of resources (including CPU, memory, disk and network) of the physical machine on which it is scheduled. The most common kinds of virtual containers are *virtual machines* (VMs) [25, 115], although other types of virtual containers can also be used in practice [1]. The applications owned by service providers are then used to provide services to end users across the Internet.

At run-time, each cloud provider runs a *resource management system* (sometimes referred to as a cloud operating system [119] [124]), which is responsible for provisioning physical resources and scheduling virtual containers on physical machines. In a typical usage scenario, a service provider submits *application requests* to the resource management system, specifying the number of virtual containers required for each application as well as the resource requirements for each virtual container. These application requests are handled by the *scheduler*, which is responsible for scheduling virtual containers on

[1] Not all cloud systems use virtualization technology. For example, Google use software containers to run application processes instead of virtual machines. But software containers are analogous to VMs in Google's cloud infrastructure.

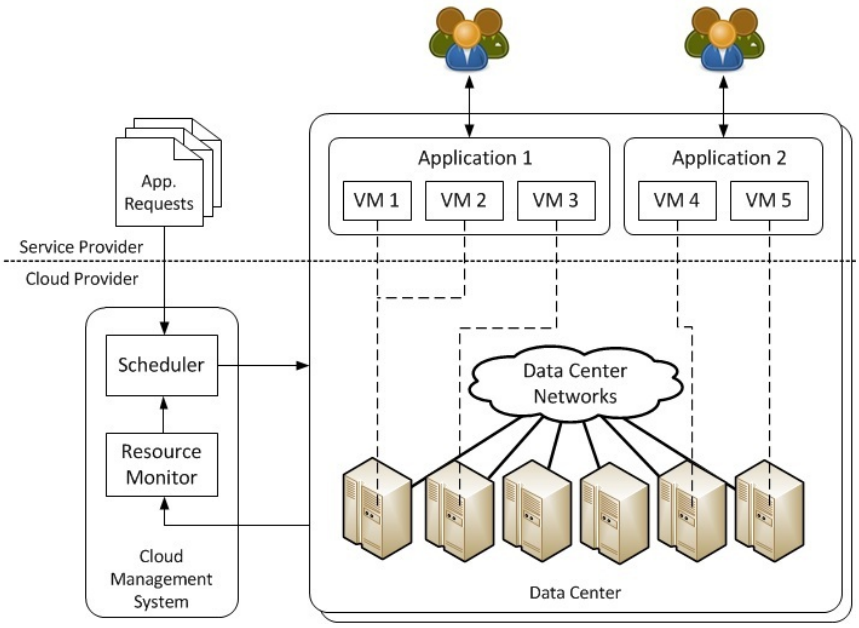


Figure 1.1: Cloud Computing System Architecture

physical machines. Advanced schedulers also provide the feature of dynamic reassigning a virtual container to other physical machines using techniques such as VM migration [85] and process migration [102].

Meanwhile, a *resource monitor* is responsible for collecting run-time statistics of each machine, including machine availability, resource utilization and virtual container status. The information collected by the resource monitor enables the scheduler to construct a global view of the data center, allowing the scheduler to make informed scheduling decisions. For example, when the scheduler needs to schedule a new request and none of the machines has sufficient capacity to schedule new virtual containers, the scheduler places the new request in a scheduling queue, or simply rejects the request.

In addition to racks of physical machines and data center networks, a data center also contains power distribution and cooling systems. The power distribution system is responsible for delivering power to individual physical machines and network switches and routers. As physical machines may generate large amounts of heat over time, the cooling system is responsible for reducing the temperature of data center, in order to prevent hardware failures due to overheating conditions [83]. It has been widely reported that both power generation and cooling requires a significant amount of energy. For example, data centers

consumed about 1.3% of the worldwide electricity supply in 2011 [18]. Such large energy consumption also raises environmental concerns regarding the carbon emissions incurred in the electricity generation process, and their impact on the surrounding environment. Motivated by this observation, cloud providers are constantly looking for ways to reduce energy cost. These include not only better management solutions for power distribution and cooling systems, but resource management techniques as well.

In practice, cloud providers typically build data centers across multiple geographical regions, in order to reduce (1) network latency for accessing cloud services, (2) capital expenditure (CAPEX) and operational expenditure (OPEX) for constructing and operating data centers, respectively [54]. As a result, a service provider is given the freedom of choosing the data centers in which its service application should be placed. At the same time, data centers in different geographical location may be subjected to different electricity costs. In many parts of the U.S., the electricity grid of each region is managed independently by a Regional Transmission Organization (RTO) which operates wholesale electricity markets in order to match supply and demand for electricity. As a result electricity prices in each region can vary independently over-time. This raises interesting challenges regarding the management of workload, power distribution and cooling systems in data centers in order to save energy. We discuss these challenges in Section 1.3.2.

1.2 Cloud Applications

Even though cloud data centers typically run a wide variety of applications, it has been reported that applications in real cloud data centers can be divided into the following two types [127]:

- *User-facing applications*: These applications are responsible for interacting directly with end users. Examples of user-facing applications include web search, content delivery, gaming, and front-end of social networking applications. User-facing applications typically consist of one or more front-end and back-end servers. For example, a typical 3-tier web application consists of a web server, application server and a database server. Each of these servers runs in a dedicated virtual container. All 3 containers may or may not reside in a single physical machine.

- *Batch applications*: These applications do not interact directly with end users during their execution. In a typical scenario, a service provider submits a batch application to the resource management system and waits for its completion. During its

execution, the batch application does not require additional instructions from the service provider. Examples of batch applications include web crawling, data mining and maintenance activities. It has been reported that most of these applications are data-intensive and typically executed using parallel computing frameworks such as MapReduce [127].

To better understand the resource and performance characteristics of batch applications, we now provide a brief description of the MapReduce programming model, as it is one of the most prevalent parallel computing frameworks for batch applications in today's cloud data centers. Originally proposed by Google, MapReduce has been designed for processing large data sets [1]. In a MapReduce system, a batch application is called a *job* that can be sub-divided into multiple *tasks*. A typical MapReduce job consists of two types of tasks: *map* and *reduce*. The input of a MapReduce job is divided into multiple file blocks of equal size (typically either 64MB or 128MB) stored in the underlying distributed file system, such as Google File System (GFS) [52] or Hadoop Distributed File System (HDFS) [2]. At run-time, each file block is processed by a map task to generate a set of intermediate key/value pairs. Each reduce task then merge all the intermediate values associated with the same key to generate the final output. For example, Figure 1.2 illustrates the execution of a `word count` job whose input consists of a collection of text files. A single map task emits a number of intermediate key-value pairs. Each key-value pair contains a word and the number of occurrence of the word in the file block the map task processes. These map tasks are executed independently of each other. Once all the map tasks are finished, the intermediate word counts produced by map tasks are collected (also known as shuffled), sorted, and summed up by reduce tasks to compute the total count for each word in the text collection. As for implementation, each map (or reduce task) can be executed by a mapper (or reducer) process that runs in a dedicated virtual container (e.g. a Java Virtual Machines).

Currently, the most popular implementation of MapReduce is Apache Hadoop MapReduce [8]. A Hadoop cluster is composed of a large number of machines with one node serving as the master and the others acting as slaves. The master node runs a job scheduler (also known as the job tracker [8]) that is responsible for scheduling tasks on slave nodes. Each slave node runs a local scheduler (also known as the task tracker [8]) that is responsible for launching and allocating resources for each task. To do so, the local scheduler launches a Java Virtual Machine (JVM) that executes the corresponding map (or reduce) task.

The original Hadoop adopts a slot-based resource allocation scheme, where the scheduler assigns tasks to each machine based on the number of available slots on that machine.

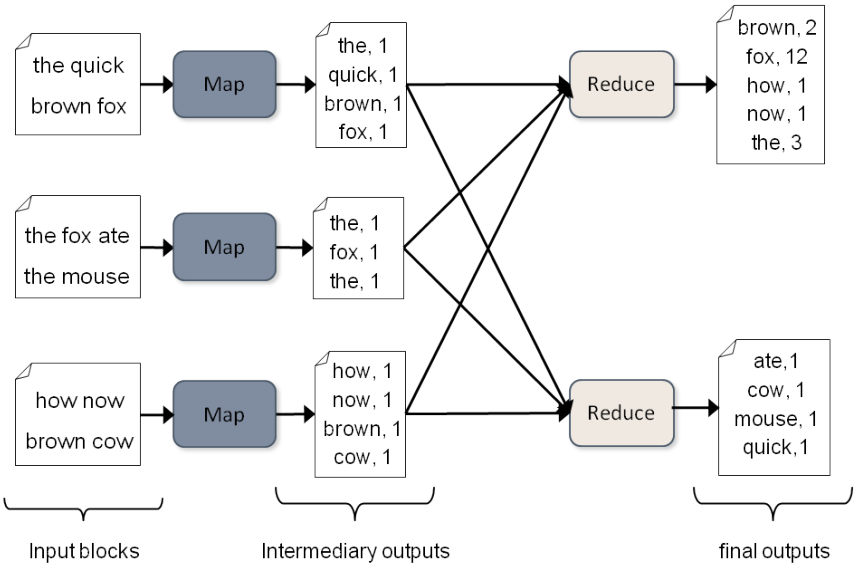


Figure 1.2: MapReduce Wordcount Example

The number of slots on each machine is usually determined by the machines capacity. However, it can also be manually specified in the cluster configuration file.

As a Hadoop cluster is usually a multi-user system, many users can simultaneously submit jobs to the cluster. The job scheduling is then performed by the job tracker, which maintains a queue of jobs to schedule. A slave node runs a task tracker that keeps track of the number of occupied map and reduce slots. The task tracker communicates with the job tracker in the master node by periodically transmitting a heartbeat message containing its state information. The state information contains the current status of each running task as well as the number of unused slots on the machine. The job tracker will use the provided information to make scheduling decisions.

Generally speaking, user-facing applications and batch applications have different performance objectives. The typical Quality-of-Service (QoS) metric for user-facing applications is user-perceived latency (i.e., how long it takes for the application to return the response upon receiving a user request) that is measured in orders of seconds. On the other hand, the typical QoS metric for batch applications is running time (i.e. how long it takes for the application to finish processing the input data set), which ranges from several seconds to several days. Even though both types of applications can be scheduled by the same resource management system, in this dissertation we study the scheduling

problem for each type of application separately, as better scheduling policies can be devised if scheduler is aware of the application type and performance objectives. This is also the typical case in production environments, where dedicated clusters are used to execute MapReduce and Dryad jobs [63, 69]. Public cloud providers such as Amazon also provide a separate service called Elastic MapReduce [7] specifically for MapReduce applications. Nevertheless, recent work such as Mesos [62] has also studied the problem of sharing data center resources among multiple scheduling frameworks.

Lastly, despite limited application categories, workloads in real cloud data center often show significant heterogeneity in terms of resource requirements, running time, arrival rates and priority levels [98]. Specifically, it has been reported that even though most of the applications in Google data centers consume very little resources, a few dozens of application can be very large and consume most of the resources (over 80%) in each data center. At the same time, applications also show diverse resource requirements. Some applications are CPU intensive, while others have high demand for I/O speed and network bandwidth. The running time of applications can also vary significantly: although most of the MapReduce jobs can finish within seconds, a few MapReduce jobs can take very long time to complete. Furthermore, like many other service systems, the arrival rate of applications requests can vary significantly over time, and sometimes can be quite spiky [32]. Lastly, applications have different priority levels. Typically, production jobs (i.e., jobs that generate revenue) are given higher priorities than non-production jobs (e.g., research experiments). Similar characteristics have also been found in Microsoft and Facebook data centers [63] [123] [53]. The heterogenous nature of cloud workload has a profound impact on the design of scheduling policies and resource management solutions, as we describe in the next section.

1.3 Resource Management Challenges

This section provides an overview of the key research challenges pertaining to resource management in cloud computing environments. As both service providers and cloud providers are involved in resource management activities, we summarize the challenges faced by service providers and cloud providers separately in the following subsections.

1.3.1 Challenges for Service Providers

In a cloud computing environment, a service provider leases resources from a cloud provider to run its applications. Therefore, the goal for a service provider is to achieve desired per-

formance objectives as specified in its service level agreement (SLA), while minimizing the total resource rental cost. However, performance objectives vary from application to application. For MapReduce applications, the typical objective is to find the appropriate job configuration (e.g. number of mapper and reducer processes), in order to meet job running time requirements while minimizing the total resource usage for running the mappers and reducers. This problem has been studied extensively in recent research literature. Even though several simple techniques have been shown to provide decent performance in practice [110] [90] [105], more sophisticated schemes, for example using machine learning techniques, have also been proposed in the literature (e.g. [61]).

In addition, the problem for user-facing applications is much more challenging. The typical QoS metric for user-facing applications is user-perceived latency. This metric is not only determined by the processing time and queueing delay of each request, but also the network delay due to carrying traffic over the Internet. The processing time and queueing delay can be controlled by carefully provisioning the resources allocated to the application. However, the network delay can only be minimized by placing the application in data centers close to end users. This problem is known as the *service placement problem* [73] [87], which has been studied for over a decade under many contexts, such as Peer-to-peer (P2P) networks and content delivery networks (CDNs) [92] [133] [42]. A key challenge here is that demand can fluctuate over time. In the context of cloud computing environments, the use of virtualization technologies enables service applications to be dynamically placed or migrated across multiple geographically distributed data centers. As cloud computing also provides support for dynamic (i.e., "elastic") resource provisioning, it is desirable to adjust the placement of applications dynamically. However, existing solutions for dynamic service placement are mostly heuristic driven, and lack a systematic approach for dealing with the dynamic nature of the problem, where demand and system conditions (e.g., resource price and network conditions) can change over time. More importantly, none of the existing solutions has considered the cost of reconfiguration in the optimization model, which may cause system instability under dynamic conditions. Due to these difficulties, the service placement problem has been an active research topic that has been pursued by various research communities [87] [130] [73] [113] [68].

1.3.2 Challenges for Cloud Providers

As the owner of physical data centers, cloud providers are responsible for allocating resources to service providers to maximize the total income, while minimizing operational expenditures. Typical operational expenditures in data center environments include costs

for server and network maintenance, energy (power and cooling) cost, as well as maintenance cost for building, floor, and racks [15]. In order to maximize total revenue from leasing resources while minimizing total operational costs, cloud providers are facing the following management challenges:

- *Workload Scheduling*: As mentioned previously, cloud data centers often consist of machines with heterogeneous resource capacities and performance characteristics. At the same time, cloud workloads show significant diversity in terms of priority, resource requirements, demand characteristics and performance objectives. Therefore, it becomes a challenging problem to determine the optimal assignment of resources to applications in order to satisfy application performance objectives, while maximizing the total utilization of the physical resources. Furthermore, it is advantageous to design separate schedulers for each type of applications. For instance, it is possible to design more effective schedulers for MapReduce jobs by leveraging the knowledge of the characteristics of MapReduce jobs.

- *Energy Management*: Data centers consume significant amount of energy. Therefore, cloud providers have strong motivation to improve data center energy efficiency. The ultimate objective of energy management is to make data centers *energy-proportional*, meaning that the energy consumption should be proportional to the actual resource usage in the data center. However, commodity machines today are far from being energy proportional. It has been reported that most of the commodity computers consume more than 50% of the maximum energy consumption even when they are idle [26]. Therefore, the most effective way to save energy is to turn the machines off when they are not being used. Various techniques, such as server consolidation [112] [108], have been shown to be effective for minimizing the number of used machines. More recently, energy efficient hardware architectures that enable slowing down CPU speeds (e.g. Dynamic Voltage Frequency Scaling (DVFS) [114]) has become commonplace. Recent research has also begun to study energy-efficient data center networks. However, in practice, reducing energy consumption often produces a negative impact on application performance. Therefore, it is a challenging issue to find the optimal trade-off between minimizing energy consumption and optimizing application performance in a dynamic cloud environment.

- *Pricing:* Pricing is another important issue not only because it directly affects the revenue of the cloud provider, but also because it can influence the allocation strategy of the service providers, which will eventually affect the effectiveness of a resource allocation scheme. Currently most of the providers such as Amazon EC2 [6],

Rackspace [17] leases virtual resources at a fixed price. However, recent work has suggested that this flat rate charging scheme can lead to inefficient outcomes, due to the mismatch between resource availability and demand fluctuation [129] [56]. A common solution for this problem is to adjust the price according to supply and demand using dynamic pricing schemes such as auctions. However, understanding the influence of dynamic pricing schemes on service providers is still a challenging problem that has not been fully addressed in the literature.

1.4 Research Contributions

In this thesis, we address the following research challenges pertaining to resource management in cloud computing environments:

1.4.1 Dynamic Service Placement in Geographically Distributed Clouds

As mentioned previously, the service placement problem concerns the placement of user-facing applications across multiple geographically distributed data centers. It can be described as follows: Given a geographically distributed and time varying demand from customers, where should a service provider place its service application, in order to minimize the resource rental cost while satisfying customer's SLA requirements? Current solutions to this problem have not fully analyzed the dynamic aspect of the problem, where demand and system conditions (e.g. resource price and network conditions) can change over time. In particular, if the placement configuration needs to be changed, the cost of reconfiguration (such as VM migration) must be considered in the optimization model.

To tackle this problem, we have developed a control-theoretic solution using the Model Predictive Control (MPC) framework [72]. Experiments show our proposed approach can minimize the resource rental cost by adapting the placement decisions according to both demand and resource price fluctuations, while minimize the cost of placement reconfiguration. Using real workload traces, we also demonstrate that a greedy, reconfiguration cost-oblivious algorithm sometimes can cause significant oscillation in service placement configurations, whereas our approach can significantly outperform the greedy solution in terms of both resource cost and reconfiguration cost.

1.4.2 Dynamic Heterogeneity-Aware Resource Provisioning in the Cloud

Data centers today consume a tremendous amount of energy for power distribution and cooling. Dynamic capacity provisioning is a promising approach for reducing energy consumption by dynamically adjusting the number of active machines to match resource demands. However, despite extensive studies of the problem, existing solutions for dynamic capacity provisioning often assume that servers and resource demands are homogeneous. In practice, it has been observed that both resource demands and machine configurations have heterogeneous characteristics, as described before. This raises the question of how to dynamically determine the number of machines of each type to be active at any given time in order to minimize total energy consumption while meeting the Service Level Objectives (SLOs) of heterogeneous workloads.

To answer this question, we designed Harmony, a heterogeneity-aware resource management system for dynamic capacity provisioning in cloud computing environments. We first use the k-means clustering algorithm to divide the workload into distinct task classes with similar characteristics in terms of resources, running times and performance requirements. Then we present a control-theoretic solution for dynamically adjusting the number of machines of each type in order to minimize total energy consumption in the data center while achieving the desired SLO in terms of scheduling delay.

1.4.3 Improving MapReduce Performance Through Predictive Scheduling

MapReduce jobs are the most prevalent type of batch applications in production data centers. The execution of both map and reduce tasks can be divided into phases. For example, a Reduce task can be divided into 3 phases: Shuffle, Sort and Reduce. One interesting observation is that different phases have different resource consumption characteristics. For example, Shuffle is an network-intensive phase with very little CPU usage, whereas the Merge phase consumes primarily CPU power and no network bandwidth. This raises the question of whether the scheduler can effectively consolidate tasks on individual machines, taking into consideration the fine-grained resource characteristics of each phase.

To take advantage of phase-level resource consumption characteristics, we have designed PRISM, a novel scheduling algorithm that effectively consolidates MapReduce tasks on servers based on fine-grained resource usage characteristics of each phase. A salient feature of this scheduler is its ability to schedule individual task execution phases of both map and

reduce tasks. Experiments show PRISM offers high resource utilization and provides up to 1.3× improvement in job running time compared to the current Hadoop resource-aware schedulers.

1.5 Thesis Organization

The remainder of this document is organized as follows. Chapter 2 provides a comprehensive overview of related work, technical preliminaries, and previous results on the problems we address in this thesis. Chapter 3, 4 and 5 are organized around the 3 research contributions described in the thesis, namely, the dynamic service placement in distributed clouds, Harmony, a heterogeneity-aware resource provisioning scheme and PRISM, a fine-grained phase-level resource scheduler for MapReduce. Finally, Chapter 6 provides concluding remarks and outlines potential future research.

Chapter 2

Related Work

This chapter surveys prior work on resource management in cloud computing environments. As discussed previously, cloud resource management concerns both service providers and cloud providers. Therefore, we describe the management challenges faced by each of them and discuss how they are addressed in the current research literature.

2.1 Resource Management Concerns of Service Providers

Given an application to be run in the cloud, the objective of the service provider is to determine how many resources should be leased from each data center in order to ensure the application performance objective is achieved. In practice, the situation differs from application to application. As mentioned before, cloud applications can be divided into two types: user-facing applications and batch applications. For batch applications such as MapReduce jobs, due to the data-intensive nature and high I/O requirement of these applications, all the virtual containers (i.e., tasks) belonging to a single job are executed in the data center where the input data is stored, in order to avoid inter-data center communication. Therefore, the objective of the service provider is to decide the number of mappers and reducers as well as their resource requirements to meet job running time requirement. On the other hand, the situation is more complicated for user-facing applications, where end users can spread over a wide range of geographical locations. In this case, deciding how many instances of the application to be placed in each data center to meet the response time requirement becomes a challenging problem.

In the following subsections, we first discuss the state-of-the-art research on resource allocation for user-facing applications in Section 2.1.1. Then we describe the recent work on resource allocation for MapReduce applications.

2.1.1 Request Specification for User-facing Applications

Placing a service across geographically distributed data centers has received significant attention in the recent research literature. Given a variety of data center locations with different resource prices and service quality in terms of access latency, the service placement problem seeks to determine the optimal placement of applications in data centers, in order to minimize total resource cost while achieving a desired service level objective (SLO) in terms of response time.

Generally speaking, the service placement problem falls into the category of facility location problems (FLP), which is a topic that has been studied extensively in operations research since the 1970's [71] [80]. The problem can be described as follows: Given a graph $G = (V \cup F, E)$, where V denotes the set of demand locations and F denotes the set of candidate locations where service facilities can be placed, let $d_{ij} \in \mathbb{R}^+$ represent the distance between a demand location $j \in V$ and candidate location $i \in F$. Assume there is an opening cost $f_i \in \mathbb{R}^+$ for opening a service facility in $i \in F$, the objective of the problem is to (1) determine the subset of locations $S \subseteq F$ where services should be placed, and (2) assign each demand location $j \in V$ to an open service facility $\sigma(j) \in S$, in order to minimize the total operational cost and total service distance. The exact definition of the total operational cost leads to several different variants of the problem:

- If the objective is to minimize the total facility opening cost and total service distance, (i.e. $C = \sum_{i \in S} f_i + \sum_{j \in V} d_{j\sigma(j)}$), the problem is called the *facility location problem.*
- If the number of facilities is a predetermined constant $K \in \mathbb{N}^+$, and the objective is to minimize the total service distance (i.e. $C = \sum_{j \in V} d_{j\sigma(j)}$), the problem is called the *K-median problem.*
- If there is an upper bound $d \in \mathbb{R}^+$ on the distance between j and its service facility $\sigma(j)$, (i.e. $d_{j\sigma(j)} \leq d$), and the objective is to minimize the total facility opening cost (i.e. $C = \sum_{i \in S} f_i$), the problem becomes the *dominating set problem.*

In addition, the above problems also have their capacitated versions. If each service $i \in F$ has a capacity C_i that specifies the maximum number of clients that can be assigned to i,

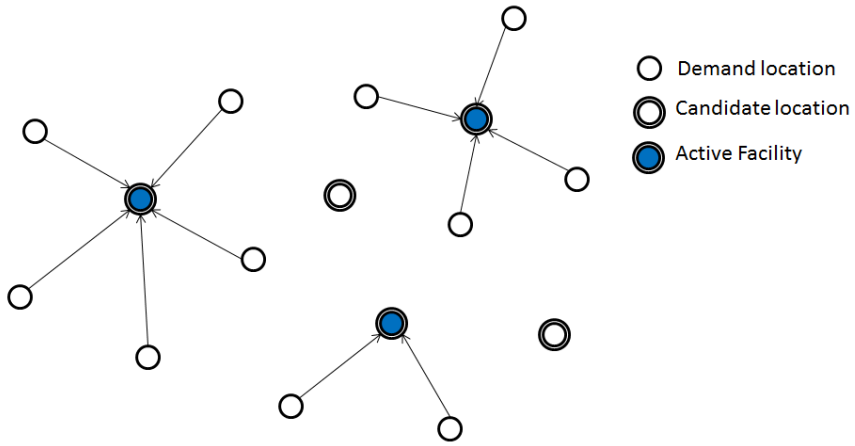


Figure 2.1: Facility Location Problem

(i.e. $|\{j : \sigma(j) = i\}| \leq C_i$), the above problems become the capacitated FLP, capacitated K-median problem and the capacitated dominating set problem, respectively.

Unfortunately, all of these problems are $\mathcal{NP}$-hard to solve. In fact, they are also APX-hard [58], meaning it is unlikely to find polynomial time algorithms that can find near-optimal solutions in all cases. Therefore, most of the existing solutions for these problems rely on polynomial time heuristic algorithms. From a theoretical perspective, the quality of an algorithm for an $\mathcal{NP}$-hard problem is determined by its *approximation ratio* ρ, which is defined as the ratio between the objective function of the solution produced by the algorithm SOL and the objective function of the optimal solution OPT [107]:

$$\rho = \frac{SOL}{OPT}.$$

Among all the problems defined above, the facility location problem is the simplest, and most widely studied in the literature. It has been shown that a simple greedy algorithm can achieve an approximation factor of $\rho = O(\log n)$, and usually works well in practice (e.g. Internet topology graphs [92]). As depicted by Algorithm 1, the greedy algorithm proceeds in iterations. In each iteration, the algorithm tries to find a facility $i \in F$ and a set of unassigned demands $S \in V$ such that assigning S to i achieves minimum average per-demand cost C_{iS}, which is defined as:

$$C_{iS} = \frac{1}{|S|}(f_i + \sum_{j \in S} d_{ij})$$

Algorithm 1 Greedy Algorithm for Facility Location Problem

1: $U \leftarrow V$
2: **while** $U \neq \{\emptyset\}$ **do**
3: Find (i, S) that minimizes C_{iS} among all $(i, S) \subseteq V \times U$
4: Open facility i, assign demand in S to i
5: $U \leftarrow U \backslash S$
6: **end while**

The algorithm repeats until all demand are assigned.

This greedy algorithm can be easily adapted to solve the other two problems. In fact, it can be shown that this greedy algorithm can be modified to create approximation algorithms for the K-median [65] and the dominating set problem [107]. Furthermore, due to its simplicity and the fact that it usually performs well in practice, it has been commonly used as a baseline for evaluating the quality of placement algorithms.

In the distributed systems literature, the service placement problem has been studied under many different contexts, most notably in content delivery networks (CDNs). In this context, the problem is called the *replica placement problem*, which aims at finding the optimal placement of content replicas across multiple geographically distributed content servers. The original replica placement problem was introduced by Qiu et al. [92], who formulated the problem as a K-median problem, and showed that the greedy algorithm (adaptation of Algorithm 1) is able to find a solution with cost within 1.1-1.6× that of the optimal solution. Subsequently, the replica placement problem has received considerable attention in the distributed systems literature. Radoslavov et al. [94] propose a fan-out heuristic that places replicas at nodes with high node degree in the AS topology graph can perform within $1.1 - 1.2\times$ times the cost of the the greedy solution. Tang et al. [104] give a variant of the replica placement problem similar to the capacitated dominating set problem, where the distance between a client and its replica is upper-bounded by fixed distance. They found that for tree topologies, the replica placement problem can be solved optimally in polynomial time. Karlsson [68] describe a framework for selecting replica placement heuristics, based on the problem objective, constraints and knowledge about the network topology.

Most of the early studies on the service placement have mainly focused on the case where the global network topology is static and can be fully obtained by the placement algorithm. However, this is not necessarily the case in practice. For large-scale or dynamic service infrastructures (e.g. peer-to-peer networks), collecting global information can be expensive. Furthermore, as service demand can change over time, the service placement

needs to be updated over-time. This may incur additional management overhead in terms of replication cost. As a result, recent research on service placement problem has been focusing on two specific aspects of the problem: (1) Enabling distributed service placement, and (2) Dynamic service placement that considers the reconfiguration cost. Distributed replica placement is important when running a centralized algorithm is inefficient in terms of measurement overhead and computational cost. For example, Chen et al. [42] study the problem of placing content replicas in a Tapestry [132] P2P network, and proposed several heuristics for distributed content replication to ensure contents are replicated close to sources of demand. Several researchers have also devised distributed approximation algorithms for the facility location problem [84] [49]. Laoutaris et al. [73] describe a distributed local search heuristic for improving the quality of service placement solutions.

On the other hand, the dynamic service placement problem aims at incrementally adjusting the service placement in order to find a balance between service quality and dynamic reconfiguration cost. In this context, the dynamic reconfiguration cost refers to the cost of dynamically replicating the service (e.g. copy over the data content and updating the service state). The importance of dynamic reconfiguration has been reported in several recent studies. For example, the paper by Oppenheimer et al. [87] studies the problem of placing applications on PlanetLab. The authors discovered that CPU and network usage can be heterogeneous and time-varying among the Planet-Lab nodes. Furthermore, a placement decision can become sub-optimal within 30 minutes, suggesting that dynamic service migration can potentially improve the system performance. A simple load-sensitive service placement strategy is then proposed, and has been shown to significantly outperform a random placement strategy. Presti et al. [91] formulate the dynamic service placement problem as a mixed integer program (MIP), for which a distributed local search algorithm was developed. Our previous work [30] also propose a distributed heuristic for capacitated facility location problem and showed that at steady state, the algorithm is able to achieve a constant performance guarantee. However, these local search algorithms do not minimize the total reconfiguration cost. Rather, the reconfiguration cost is controlled indirectly by limiting the number of local search moves. More recently, Arora et al. [23] provide an approximation algorithm for the Dynamic service problem in the context of virtual networks that explicitly considers the reconfiguration cost, however, the approximation guarantee is weak and not sufficient to guarantee the quality of the solution in the general case.

With the growth of large-scale data center infrastructures, recently the replica placement problem has been adapted to placing online services across geographically distributed data centers. The goal is to minimize the resource rental cost while achieving performance objectives specified in the SLA. Even though the replica placement model is still applicable,

data center environments exhibit several interesting features that can lead to simplified solutions. First, the number of data centers is usually small (in the order of 10s), mainly due to the high investment for building data centers. Furthermore, each data center usually has large free capacity. As a result, a common assumption made for the service placement problem in data centers is that the number of replicas can take fractional values. This allows the problem to be formulated as a convex optimization problem that can be solved optimally in polynomial time. In the implementation, the fractional values can be rounded up to the nearest integer value. This is a reasonable assumption when the number of service instances is large. Using this assumption, Rao et al. [95] study the problem of server placement in a multi-electricity market environment with the goal of minimizing electricity cost. More recently, Liu et al. [79] present a distributed solution for the same problem, taking into consideration both request response time and energy cost. However, both of these works only studied the static case of the problem. As both service and resource price can change independently over time, it is necessary to find a solution that dynamically adjust the placement strategies to optimize both service performance and cost, while minimizing the overhead of reconfiguration.

2.1.2 Request Specification for MapReduce Applications

MapReduce is another type of application that is popular in data center environments. However, different from user-facing applications, MapReduce applications are data intensive and usually scheduled in these data centers that store the input data sets, in order to minimize inter-data center communication. Thus, for each MapReduce job, the objective of the service provider becomes determining appropriate request specification in terms of the minimum number of mappers and reducers, as well as the amount of resources allocated to each mapper and reducer, such that total resource rental cost is minimized while ensuring the job is able to meet performance objective in terms of job completion time.

To achieve this objective, a typical approach employed by service providers is job profiling [110], which typically records the following information about the execution of a job:

- The job specification, which specifies the input size, the number of map and reduce tasks, the number of mappers (i.e. processes used to execute map tasks) and reducers (i.e. processes used to execute reduce tasks) and other job specific input parameters [61].
- The run-time task characteristics, which includes the running time, the consumption

of CPU, memory, disk and network resources of each task. Additional information, such as intermediary file size, can also be recorded in the job profile.

Once a job is profiled, the service provider can use the profile information to decide how much resources are required to minimize total cost while meeting the job running time objective. For example, ARIA [111] is a system that uses profiled information to decide the number of mappers and reducers. The authors first discover that for individual jobs, task running time is stable (within 10% variation) over multiple runs. Based on this observation, an analytic model for task competition time for all three phases: map, shuffle and reduce, is developed. The parameters of the model are determined either through historical logs or using small scale experiments. Using the proposed model, the authors give a polynomial time algorithm for determining the minimum number of required mapper and reducers to ensure the job finishes before its deadline.

Similarly, Tian et al. [105] propose a technique for estimating the completion time of MapReduce jobs using a simple performance model. Based on a few assumptions, their model breaks a map task into 4 phases: Read, Map, Partition, Sort, and Combine. The cost of each phase is modeled as a function of the total number of map tasks in the job and number of slots available for the job. Similarly, a reduce task is divided into 4 phases: Copy, Sort, Reduce, and Write Back. The actual values of parameters used in the cost model can be estimated using small scale experiments. Using this performance model, the authors formulate optimization problems for (1) optimizing job performance subject to budget constraint, and (2) minimizing the budget used subject to completion time constraint. Their initial experiments show the prediction model achieves good accuracy.

More recently Herodotou et al. present a self-tuning system for Hadoop called Starfish [61] which, similar to [105] and [111], uses profiled information to predict job running time and optimize job configurations. Starfish provides a sophisticated profiler that is capable of collecting detailed statistical information about MapReduce task execution, such as the time each task spends in each phase, and the size of intermediary files. Based on the statistical information collected by the profiler, Starfish uses machine learning techniques to answer questions concerning job performance under various input parameters. To date, this is the most sophisticated tool available for helping service providers determining the best specification for MapReduce jobs.

Despite these recent efforts, however, existing solutions still suffer from several limitations. In particular, all of the existing solutions have assumed that run-time task usage is stable over-time.However, this is not true in practice as run-time task usage can vary significantly, depending on the phase it is currently running. For example, the network I/O consumption during the shuffle phase can be significantly higher than during the reduce

phase. As a result, using the profiles produced by task-level profilers can lead to inefficient job schedules that can potentially cause either resource under-utilization or contention.

2.2 Resource Management Concerns of Cloud Providers

As the owner of the physical infrastructure of data centers, the objective of the cloud provider is to schedule resource requests, in order to maximize total revenue, minimize the operational expenditure while satisfying the SLA requirements. In this section, we focus on two important activities, namely workload scheduling and energy management. For workload scheduling, we separately discuss the scheduling of traditional VM workload and MapReduce scheduling.

2.2.1 Scheduling VM Workload in Data Centers

Traditional data centers schedule applications that run in dedicated containers (e.g. VMs). Even though not every cloud uses virtualization technology, the problem of scheduling these applications is commonly studied under the topic of VM scheduling [126]. Given physical machines with various resource capacities and heterogeneous VMs of different size in terms of CPU, memory, disk and bandwidth requirements, the VM scheduling problem aims at finding an assignment of VMs to physical machines that minimizes the scheduling delay (the difference between the time at which a VM request is received and the time it is scheduled) and maximize the utilization of the physical machines used.

In the context of Cloud data centers, the VM scheduling problem can be modeled as a variant of the online vector bin-packing problem [39], where VMs arrive and leave the system over-time. Based on this intuition, it is natural to adopt online bin-packing algorithms, such as first-fit, best-fit and round-robin first-fit (i. e., running first-fit in a round-robin fashion for the purpose of load balancing) for VM scheduling. Indeed, this is the case for many real cloud systems. For example, it has been reported that Microsoft uses a variant of first-fit bin packing algorithm for VM scheduling in its clusters [75]. Google uses a variant of the best-fit algorithm for workload scheduling in its compute clusters [100]. Specifically, a score is computed for each VM-to-machine assignment, and each VM is assigned to the machine with the highest score. In open source cloud platforms, it has been reported that Eucalyptus [11] provides the first-fit and round-robin first-fit algorithm for VM scheduling. To study the effectiveness of bin-packing algorithms for VM scheduling, the work by Lee et al. [75] recently quantitatively evaluated the performance of several variants

of the best-fit and first-fit algorithms using a production workload for the Microsoft Bing product. For the best-fit based algorithms, the authors proposed several ways to combine multiple resource requirements into a single score to measure the "fitness" of a VM-to-machine assignment. Their evaluation result show that all the bin-packing algorithms can produce schedules that are no worse than 20% of the optimal schedule, in terms of the number of physical machines used. Furthermore, "smart" bin-packing algorithms that consider utilization of multiple resource types can reduce the performance gap to less than 10%. These quantitative evaluations support the use of bin-packing algorithms for VM scheduling in data centers.

So far the discussion on VM scheduling has been focusing on the assignment of VMs to physical machines. Another important aspect of VM scheduling that has been largely overlooked is the allocation of network resources. As cloud applications can also be sensitive to network performance, recent research proposals have advocated to offer Virtual Data Center (VDCs) instead of VMs [59] [131]. A VDC consists of virtual machines (VMs) connected through switches, routers and links with guaranteed bandwidth. Scheduling VDCs in data centers is more complicated than VM scheduling. In fact, it generalizes the *virtual network embedding* problem, which has been shown to be strongly $\mathcal{NP}$-hard [43]. However, the bin-packing algorithms described above can be adapted for VDC scheduling. In particular, a greedy algorithm that assigns each VM to a machine that minimizes the average distance to all assigned VMs has been proposed in several recent systems including Oktopus [24], CloudNaaS [29] and VDC-Planner [131]. These studies show that this simple greedy algorithm generally performs well for network-aware VM scheduling.

2.2.2 MapReduce Scheduling

MapReduce is another of type of application that is commonly found in data centers. Even though VM scheduling algorithms can also be used for MapReduce scheduling, leveraging the domain knowledge of the MapReduce framework can lead to a better scheduling algorithm for MapReduce jobs.

Generally speaking, the responsibility of a MapReduce job scheduler is to assign tasks to machines with consideration for both efficiency and fairness. To achieve efficiency, job schedulers must reduce resource wastage and maintain high utilization of the cluster. At the same time, fairness is also an important concern since a cluster is often shared by many jobs from multiple users. The most common fairness criteria are [63]: (1) preventing one job from occupying the whole cluster, delaying the completion of everyone else's jobs; (2) ensuring low latency for small or short jobs while maintaining a high overall throughput.

The level of fairness also reflects stability of service quality. Because resources are fairly divided among the running jobs, each job will experience similar speed up or slow down depending on workload conditions. Job response time thus becomes steady and predictable under fair scheduling. A system with reasonable and predictable response time is often considered more desirable than a system that is faster on average but is highly variable in job performance.

The original Hadoop uses a First-In-First-Out (FIFO) scheduler for job scheduling. However, FIFO is not a fair scheduling policy, because a large job can occupy the entire cluster and delay the execution of all subsequent small jobs. Based on this observation, the Hadoop fair scheduler [14] was developed to allow multiple jobs to share the cluster resources [122]. As mentioned previously, in the Hadoop implementation, the resources in Hadoop cluster are divided into multiple slots. At run-time, the Hadoop fair scheduler computes the fair share of resources (e.g., the fraction of the total slots) that each job should receive. By default, the Hadoop fair scheduler uses the max-min fair sharing policy for assigning tasks to slots, even though it also allows job owners to specify the minimum number of slots the job should receive.

Quincy [63] is another fair scheduler that achieves fairness by limiting the number of running tasks belonging to each job at a given time. Specifically, it defines fairness as the objective of ensuring a job j which runs for T_j seconds given exclusive access to a cluster should take no more than $J \cdot T_j$ seconds when there are J jobs concurrently executing on that cluster. The intuition is that when multiple jobs share the same cluster, each job should experience similar performance gain or performance loss in terms of running time. For example, we can consider a cluster that currently has J jobs running, and each job is executing n_j tasks concurrently. Suppose each job j can execute N_j tasks simultaneously when it is given exclusive access to the cluster, then the objective of the fair scheduler is to satisfy

$$\frac{n_1}{N_1} = \frac{n_2}{N_2} = ... = \frac{n_J}{N_J}, \tag{2.1}$$

as doing so can ensure every job receives similar speed-up (or slow-down) in terms of job running time. In this case, we say a job j has the highest deficit if it receives the lowest fair share (i.e. $\frac{n_j}{N_j} \leq \frac{n_l}{N_l}$ for all $1 \leq l \leq j$). Therefore, a simple greedy scheduling algorithm is to schedule a task that belongs to the job with the highest deficit in attempt to equalize their fair share, as shown in Equation 2.1. Furthermore, different from the Hadoop fair scheduler, Quincy considers data locality in addition to job fairness. Data locality is a major concern because transferring large volumes of data across the data center network can be expensive in terms of both latency and bandwidth usage. To achieve both fairness

and data locality, Quincy encodes the scheduling decisions as a flow network where edge weights represent the demands of job scheduling. In this way, the cost of each scheduling decision is quantified in terms of the cost for data transfer and the overhead of terminating tasks. Every time a scheduling decision needs to be made (for example, a new job arrives), a min-cost flow algorithm is used to find a minimum cost assignment of tasks to machines. Quincy then launches new tasks according to the minimum cost assignment. Through experiments, it has been reported that Quincy can significantly outperform a queue-based FIFO scheduler in large MapReduce clusters.

Another line of research on MapReduce scheduling focuses on the issue of mitigating stragglers. A *straggler* (also known as an outlier [21]) is a task that runs much slower than other tasks that belong to the same job. Stragglers can become a major performance bottleneck for MapReduce jobs. For example, if a map task is experiencing a long running time, then the execution of all the reducers will be delayed since they are all waiting for the intermediatory output from the map task. To address this issue, the work by Zaharia et al. proposes a scheduling algorithm called Longest Approximate Time to End (LATE) [125], which monitors the progress of every running task and launches speculative copies of tasks that have longest expected time to finish. More recently, Ananthanarayanan et al. [21] provide an extensive study of root causes of stragglers found in a Microsoft compute cluster, and discovered resource contention, imbalance in input size for reduce tasks and lacking of data locality (i.e. map tasks placed far from the machines hosting the input data) are the major causes of stragglers. They present Mantri, a system that monitors tasks and reduces outliers using cause- and resource-aware techniques. Mantri's strategies include restarting outliers, greedy network-aware placement of tasks and protecting outputs of valuable tasks. Similar to LATE, Mantri preempts and restarts a task elsewhere if doing so can improve the task running time. Compared to LATE, Mantri further considers the impact of network congestion on a task's progress, resulting in much better performance than LATE.

The original Hadoop MapReduce implements a slot-based resource allocation scheme, which does not take run-time task resource consumption into consideration. As a result, several recent studies have reported the inefficiency introduced due to such simple design, and proposed solutions. For instance, Ghodsi et al. [53] propose Dominate Resource Fairness (DRF) as a fairness criterion for determining how much resource of each type should be allocated to each job. DRF is a generalization of the max-min fairness [116] to multiple resource types. As tasks belonging to different jobs have different bottleneck resources (called dominant resources), DRF essentially ensures that the share of the dominant resource is allocated according to the max-min principle. Specifically, consider a cluster of R types of resources whose capacity for each type of resource $r \in R$ is C_r. Assume for

each user (or job) j, the resource consumption of type r is $c_j r$, then the dominant share of user (or job) j can be computed as $\max_{r \in R} \left\{ \frac{c_{jr}}{C_r} \right\}$. At run-time, the DRF scheduler aims at equalize their fair share

$$\max_{r \in R} \left\{ \frac{c_{1r}}{C_r} \right\} = \max_{r \in R} \left\{ \frac{c_{2r}}{C_r} \right\} = ... = \max_{r \in R} \left\{ \frac{c_{Jr}}{C_r} \right\}, \tag{2.2}$$

The authors of [53] further show that compared to many other fairness criteria, DRF is Pareto-efficient, strategy-proof, envy-free, and incentivizes users to share their resources. Subsequently, DRF scheduling algorithm has been extended into a network scheduling algorithm called Dominant Resource Fair Queuing (DRFQ), which is a variant of the Start-time Fair Queuing (SFQ) [55] that achieves flow-level DRF in network components such as routers, firewalls and intrusion detection systems.

Polo et al. propose an Adaptive Resource-aware Scheduler (RAS) [90] that uses job specific slots for scheduling. The size of each slot is determined by the task requirements captured in job profiles, as described in Section 2.1.2. By considering the task resource requirements as well as job deadline requirements, RAS is able to find a better tradeoff between resource utilization and job running time than the current Hadoop Fair Scheduler.

Sharma et al. recently propose a framework called MROrchestrator [101] to shift the resource allocation from a slot-based model to a more flexible and adaptive resource sharing model. In particular, MROrchestrator dynamically detects resource bottleneck on each machine, and reallocates resources among tasks to fairly share the bottleneck resource. However, MROrchestrator is mainly designed for mitigating resource contention among collocated tasks. It does not provide a scheme for resource-aware scheduling.

More recently, Xie et al. have proposed PROTEUS [120], a framework that allocates virtual clusters (similar to VDCs) for MapReduce jobs, taking into consideration the time varying bandwidth requirement of each job. Through simulations and preliminary experiments using a small MapReduce cluster, it can be shown that by considering job bandwidth requirements at a fine-grained level, it is possible to improve the job throughput by up to 40%. However, PROTEUS mainly focuses on allocating network bandwidth among collocated jobs, and ignores other system resources such as CPU and disk. Finally, Next Generation Hadoop [16] represents a major endeavor towards efficient resource allocation in Hadoop MapReduce clusters. It offers the ability to specify the size of the task virtual machine in terms of CPU, memory, disk, and network usage.

One of the key limitations of the above resource-aware scheduling schemes is that they assume task usage does not vary significantly over the course of execution. However, this

is not true in practice. As mentioned in Section 1.2, the execution of a MapReduce task can be divided into phases with significantly different resource requirements. As a result, recent work has begun to study MapReduce scheduling at phase-level. More recently, Lin et al. [78] have studied the problem of optimizing the overlapping of shuffle phases with map phases. Specifically, the MapReduce scheduler is modeled as an overlapping tandem queue and the optimal phase scheduling problem can be shown to be an $\mathcal{NP}$-hard optimization problem. The authors then propose an online scheduling algorithm that has guaranteed competitive ratio compared to the optimal offline solution. However, they have not considered the different resource consumption characteristics of each phase as well as network resource allocation.

2.2.3 Energy Management in Data Centers

Improving energy efficiency is another major concern of cloud providers. In practice, the operational expenditure on energy not only comes from running physical machines, but also from cooling down the entire data center. For large companies like Google, a 3% reduction in energy cost can translate into over a million dollars in cost savings [93]. On the other hand, governmental agencies continue to implement standards and regulations to promote energy-efficient (i.e., "Green") computing [10]. Motivated by these observations, cutting down electricity cost has become a primary concern of today's data center operators.

In the research literature, a large body of recent work tries to improve energy efficiency of data centers. A plethora of techniques have been proposed to tackle different aspects of the problem. In the context of resource management, one of the simplest yet most effective approach for reducing energy cost is to dynamically adjust the data center capacity by turning off unused machines, or to set them to a power-saving (e.g., "sleep") state. This is supported by the evidence that an idle machine can consume as much as 50% of the power when the machine is fully utilized [40] [50]. Unsurprisingly, a number of efforts are trying to leverage this fact to save energy by minimizing the number of active machines using a combination of server consolidation and dynamic capacity provisioning:

Server consolidation aims at finding an assignment of workload to machines in order to minimize the number of used machines. Typically, server consolidation is achieved through (1) resource-aware workload scheduling, as described in Section 2.2.1, and (2) dynamically adjusting workload placement using migration. The former approach relies on the scheduler to find a good initial placement of workloads, whereas the second approach realizes that the initial workload placement can become sub-optimal over time, and improves workload placement using techniques such as VM migration.

On the other hand, *dynamic capacity provisioning* aims at dynamically controlling the number of active machines by switching machines on and off (or in and out of "sleep" state), based on various factors such as the workload arrival rate, workload performance requirement and the electricity price. While over-provisioning the data center capacity can lead to sub-optimal energy savings, under-provisioning the data center capacity can cause significant performance penalty in terms of scheduling delay, which is the time a resource request has to wait before it is scheduled in the data center. Furthermore, there is usually a reconfiguration cost associated with switching machine state. For example, it has been reported that frequently switching a machine on and off will reduce the lifetime of the machine (often known as the "wear-and-tear" effect) [77]. Therefore, it is necessary to consider both the scheduling delay and the reconfiguration cost while carrying out a dynamic capacity provisioning procedure.

There is a large body of literature that has investigated one or more aspects of server consolidation and dynamic capacity provisioning. For example, *pMapper* [109] is a dynamic, migration-aware server consolidation scheme that aims at finding a tradeoff between application performance and power consumption. As the server consolidation problem generalizes the bin-packing problem, several greedy heuristics are then proposed for placing new VMs. As workload placement can become stale over time, local search heuristics are used to incrementally convert the current workload placement to the desired workload placement. Through simulations using realistic workload traces, it has been shown that pMapper is capable of achieving significant energy savings. Similarly, *Mistral* [67] is a workload consolidation framework that dynamically adjusts VM placement in order to achieve the optimal tradeoff between power consumption, application performance, and adaptation costs (which includes the cost of adjusting VM capacity, live migrating VMs and shutting down/restarting a physical machine). Mistral relies on offline measurements to estimate run time adaptation costs, and a workload predictor to estimate the periods during which the workload for each application is stable. As the optimal VM consolidation problem is $\mathcal{NP}$-hard, Mistral proposes a self-aware A^* search algorithm to prune the search space. Experiments show that Mistral can significantly reduce the cost of energy and performance penalty due to VM migration.

Another energy reduction technique that has been proposed in the literature is temperature management. The goal is to minimize the total cooling cost while preventing the occurrence of overheating conditions. The overheating conditions are undesirable as they have a negative impact on hardware reliability and often cause higher server energy consumption due to increased fan speed. To understand why temperature management is important, recently El-Sayed et al. [48] have analyzed traces from several Google data centers to identify the impact of temperature on machine reliability and energy consumption.

They found the machine failure rate tends to grow linearly with the average machine temperature. On the other hand, high temperature can cause significant increase in machine power consumption due to increased fan speed. Furthermore, machine temperature may vary across data centers. In order words, certain machines may become "hot spots" of the data center. In this context, Bash et al. [28] have studied the problem of dynamically controlling the air conditioning system based on the measured temperature in data centers. Moore et al. [83] have studied the problem of temperature-aware workload placement. They discovered that, by placing workload according to the temperature profile of each physical machine, it is possible to reduce the cooling cost substantially. Through experiments, the authors show that in the best case, temperature-aware workload placement can reduce cooling cost by almost 50%.

It has been reported that data center networks are responsible for 10%-20% energy consumption of a typical data center. As a result, recent studies have proposed techniques for saving energy in data center networks. For example, ElasticTree [60] is a network power manager that finds a subset of network components (links and switches) where network traffic can be feasibly routed using these components. This allows the unused components to be turned off to save energy. The authors first present a formal model for minimizing energy consumption in data center networks, and showed that optimally solving this problem is intractable in practical settings. Then the authors propose two heuristics, including a greedy bin-packing algorithm and a topology-aware heuristic that takes advantage of the symmetry of the recently proposed fat-tree topology. Experiments using real data center traffic traces show ElasticTree is able to reduce data center energy consumption by up to 50%. Similarly Abts et al. [20] describe several ways for designing energy-proportional data center networks, including topology design and dynamic switch link-rate adjustments. Clearly, as physical servers become more energy efficient, the energy efficiency of network equipment is becoming an increasingly important field for research innovation.

So far all the discussion on energy efficiency has been focusing on the case of a single data center. However, most cloud providers build data centers across multiple geographical regions. Interestingly, the energy cost of data centers in different regions can differ significantly from time to time. In many parts of the U.S., the electricity grid of each region is managed independently by a Regional Transmission Organization (RTO) which operates wholesales electricity markets in order to match supply and demand for electricity. Furthermore, the source of energy can differ from region to region. Even though traditional energy sources such as fossil fuel are still being harvested nowadays, they generally have negative impact on the environment in terms of carbon emission. As result, there is a strong incentive to use green energy as a source of electricity to power up data centers.

However, green energy such as solar can be easily obtained in areas such as California, and tidal energy can only be harvested in coastal regions. Therefore, it is a challenging problem to find ways to leverage differences in energy source and price to reduce total energy consumption and carbon emissions across multiple data centers. In this context, Rao et al. [95] have studied the problem of service placement and request routing in order to minimize energy cost by leveraging the electricity price difference across data centers. Gao et al. [51] have studied the problem of achieving a 3-way tradeoff among service access latency, energy cost and carbon emission for CDNs, using request routing and dynamic data placement. However, a limitation of all existing approaches is that as service demand can fluctuate significantly over-time (e.g. flash crowd effect), it is necessary to reconfigure the placement configuration at a rapid rate in order to ensure high service performance at all times. However, none of the existing works has considered the cost of dynamic reconfiguration.

Chapter 3

Dynamic Service Placement in Geo-Distributed Clouds

3.1 Introduction

Large-scale online service providers have been increasingly relying on geographically distributed cloud infrastructures for service hosting and delivery. In this context, a key challenge faced by service providers is to determine the locations where service applications should be placed such that the hosting cost is minimized while key performance requirements (e.g. response time) are assured. This involves solving two problems jointly: (1) deciding on the number of servers placed in each data center, and (2) routing each request to appropriate servers to minimize response time. As cloud providers typically offer on-demand and elastic resource access, it is possible to adjust the number of servers to match service demand in a dynamic way. Furthermore, the cost of reconfiguration (i.e., the cost of adding and removing servers) must be taken into account. The consideration of reconfiguration cost is important for ensuring the system stability and minimum management overhead and costs. In particular, these operations have costs for setup (e.g., VM image distribution) and tear-down (e.g., data fetching / state transfer). For example, it has been reported that starting up a VM in Amazon EC2 cloud can take between 20 seconds to more than 13 minutes, depending on the VM size and OS running in the VM [81]. Thus, the time it takes to scale up the service needs to be considered when making scaling decisions, as under-provisioning servers during the scaling up process can cause revenue loss. Thus, it is in the interest of service providers to reduce such reconfiguration cost.

On the other hand, the price of resources offered by cloud providers is also subject to

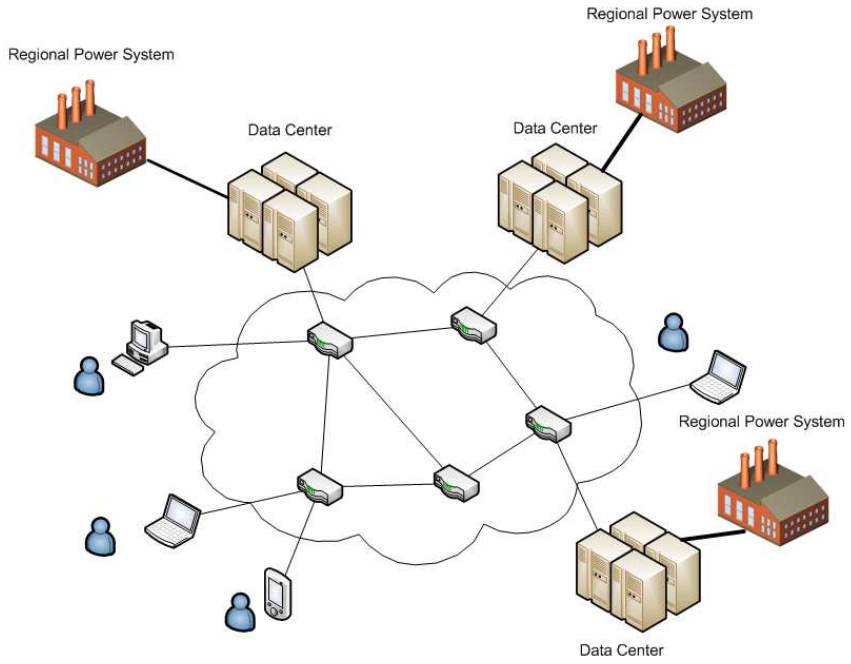


Figure 3.1: Model of service placement in geographically distributed data centers

change. In particular, energy consumption is a major contributor to the operational cost of a data center. In many regions of the U.S., the electricity grid of each region is managed independently by a Regional Transmission Organization (RTO) which operates wholesales electricity markets in order to match supply and demand for electricity, as illustrated in Figure 3.1. As a result, electricity prices in each region can vary independently over time. Based on this fact, recently there have been several studies on dynamic server placement [79,95] and request dispatching [93] in private clouds, taking into account fluctuating energy costs. The same benefit can be achieved in public clouds by introducing some degree of dynamic pricing, such as the one being used by Amazon EC2 [5].

Combining the above observations, a service provider is facing the problem of how to dynamically control the number of servers placed in each data center to minimize the total resource cost while satisfying SLA requirements, taking into consideration the fluctuation of both demand and resource price. We call this problem *dynamic service placement problem (DSPP)*. This problem shares many similarities with traditional replica placement problem in [68, 92, 113]; however, the price fluctuation is often neglected in the existing literature. Recently, there have been several studies on this problem (e.g., [79]). However, the dynamic aspect of the problem, particularly the cost of dynamically starting and shutting down

servers, is still largely unaddressed.

In this chapter, we study the DSPP problem using both control and game theoretic methods. In our solution, we first propose a control framework based on Model Predictive Control (MPC) approach to provide an online adaptive control mechanism which aims at reducing service provider costs, namely, resource allocation and reconfiguration costs. We further extend this framework to a game-theoretic model to consider the competition among multiple service providers, taking into consideration the capacity constraint of each data center. This model is realistic for several reasons: (1) on-demand resource allocation mechanisms can often lead to situations where resource demand exceeds the capacity available in a data center (e.g., during holiday seasons). (2) Recently, there are numerous proposals that advocate for small-scale data centers (e.g., [45], [64]). In both cases, limited data center capacity can result in some service providers to fail to obtain the desired resources. In this case, we analyze the outcome of resource competition and show that there exists an optimal outcome, and provide algorithms to compute this outcome. Finally, using simulations based on realistic topologies, demand and prices, we demonstrate the effectiveness of our proposed approach and analyze various properties of the resource competition game.

The remainder of the chapter is organized as follows. Section 3.2 presents the proposed framework for a single service provider. Section 3.4.6 describes the problem formulation of DSPP for a single service provider. The design of our controller for DSPP is provided in Section 3.5. In Section 3.6, we extend our framework to a multi-provider scenario and analyze the outcome of the resource competition game. Section 3.7 presents our experimental results. Finally, we conclude the chapter in Section 3.8.

3.2 System Architecture and Design

We consider a multi-regional cloud environment that consists of multiple data centers situated at different geographical locations. Our system architecture consists of 4 components as depicted in Figure 3.2: (1) request routers, (2) monitoring module, (3) analysis and prediction module, and (4) the resource controller. Both the request routers and the monitoring module can be directly owned by the service provider, or leased from other service providers who offer them as services. In particular, the service provider controls *request routers* (also known as redirectors) which are responsible for redirecting the requests to the appropriate servers [89, 118]. In practice, request redirection can play a key role in improving server accessibility through load balancing, latency minimization and content replication. For instance, Amazon EC2 Elastic Load Balancing service [6] is an example

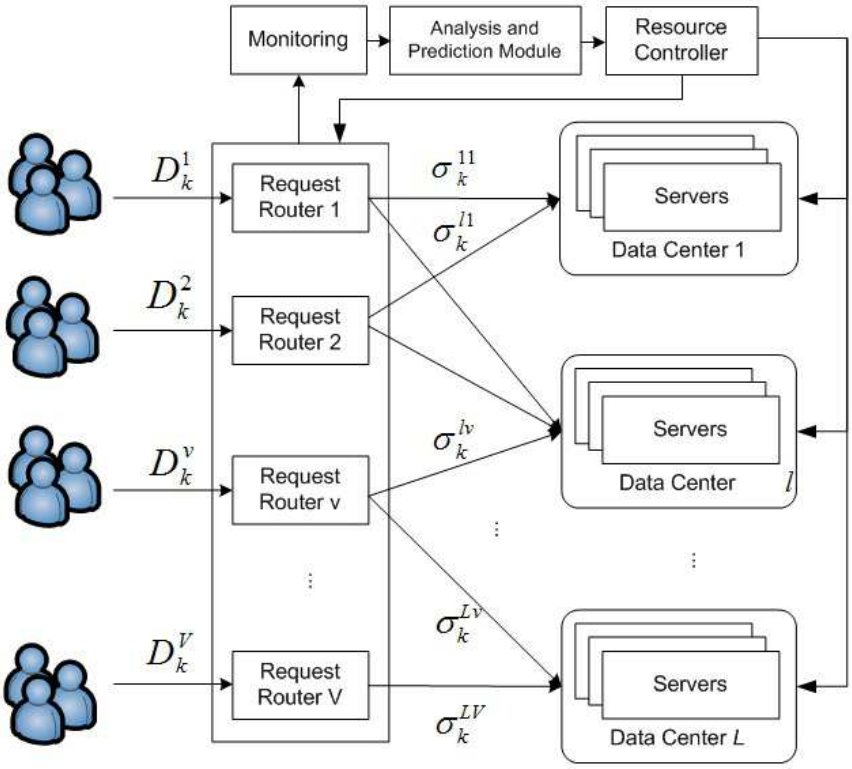


Figure 3.2: System architecture for a single service provider

of a simplified request router. More sophisticated designs (e.g. DONAR [118]) have also been studied in the literature. The *monitoring module* is responsible for collecting statistics, including the amount of requests received (i.e. the demand) at the different request routers and the prices offered by each data center. The *analysis and prediction module* models the dynamics of demand and price fluctuations, and forecasts the future values of both demand and resource prices. In practice, it has been shown that both demand and price in production data centers generally exhibit daily fluctuation patterns [79], [93]. In this case, the demand can be reasonably predicted using historical traces. However, there are occasions where both demand and resource price can behave in an unexpected manner, e.g., flash-crowd effect or system failure. Alternative prediction models such as autoregressive (AR) models [35], and demand characterization models [31] may be used. It is important to point out that our control-theoretic model is generic and can work with any demand prediction technique.

Finally, the *resource controller* is responsible for solving the DSPP and making online

control decisions at run time. It dynamically adjusts the number of servers leased in each data center in order to satisfy the SLA requirements (in terms of latency), while minimizing the resource rental cost. Furthermore, it informs the request routers about the number of servers allocated in each data center. The request routers must then find appropriate assignment of demand to the allocated servers. In our system architecture, each request router adopts a simple strategy which is to split demand proportionally among the servers that satisfy the SLA requirements. The formal demand assignment model is discussed in Section 3.13.

3.3 Understanding Server Reconfiguration Cost

One of the key differences between our framework and previous work is the modeling of server reconfiguration cost, which includes both the cost of scaling up and scaling down servers within a data center. Modeling server reconfiguration cost is important for several reasons. First, starting up a new server running in a VM takes time. Several recent papers [88] [81] have examined the VM start-up time and shut-down time. The VM start-up time is defined as the time it takes for the VM to become accessible through `ssh` after the acquisition request is submitted. Similarly, the shut-down time is defined as the time it takes for the cloud to release the resources occupied by the VM. Mao et. al. measured the VM start-up and shut-down time for several major cloud providers including Amazon, Windows Azure and Rackspace [81]. The measurements show that VM startup time is independent of the time of day at which the request is issued. However, it is typically dependent on the cloud provider, the data center that schedules the VM as well as the VM image size. For example, it has been shown that on average it takes roughly 100 seconds to start a single VM of instance types M1.SMALL, C1.MEDIUM and M1.LARGE running Linux operating system. This number can be up to 9 times greater for Windows instances. Furthermore, the VM start-up time grows linearly with VM image size. A VM with $4GB$ image can take up to 400 seconds to start up, which is almost $4\times$ the start-up time of a VM with $1GB$ image. Moreover, VM start-up time can also vary from data center to data center. It has been found that VM start-up time for newly constructed data centers sometimes can be much worse than other data centers. Finally, VM shut-down time is usually very short (typically less than 10 seconds) compared to VM start-up time. These results are consistent in the sense that similar observations have also been found in Rackspace and Windows Azure data centers [81].

The main implication of the above observations is that dynamic server capacity provisioning cannot be achieved instantaneously. It takes time for a virtual server to start-up,

depending on the exact type of virtual server as well as the clould provider and data center locations. During busy periods when provisioned server capacity is unable to fulfill demand with the desired QoS, scaling up the service infrastructure may take several minutes to complete, during which service quality continues to suffer. Thus, we consider the revenue loss during the service scaling-up periods as part of the reconfiguration cost. Therefore, service providers need to find a trade-off between service agility (for minimizing resource cost) and reconfiguration cost when making service placement decisions.

3.4 Problem Formulation

We model the network as a bipartite graph $G = (L \cup V, E)$, where L denotes the set of data centers, V denotes the location of customers. For instance, V can be the set of access networks to which customers are connected. Denote by $E \subseteq L \times V$ the communication paths between customers and data centers. We also assign constant weights d^{lv} to denote the network latency between a data center $l \in L$ and a client location $v \in V$.

In our framework, we consider a discrete-time system model where time is divided into multiple time periods called *reconfiguration periods* corresponding to the timescale at which server placement and routing decisions are made. We assume that there is an interval of interest $\mathcal{K} = \{0, 1, 2, ..., K\}$ that consists of $K + 1$ periods. Let $N = \{1, 2, ..., n\}$ denote the set of service providers. We assume that at time $k \in \mathcal{K}$, each customer location $v \in V$ has demand D_k^v in terms of average arrival rate of requests from location v at time k. For simplicity, we assume that all the servers rented by each service provider have identical size and functionality. For instance, a server can be a virtual machine (VM) that runs a specific application image. We define the state variable $x_k^l \in \mathbb{R}_+$ as the number of servers owned by the service provider at location $l \in L$ at time k. To simplify the model, we assume that x_k^l can take continuous values rather than discrete values. This assumption is reasonable for large-scale services that require tens or hundreds of servers, where the weight of each individual server in the overall solution is small. In this case, we can always obtain a feasible solution by rounding up the continuous values to the nearest integer values. Based on this assumption, we can further decouple x_k^l by defining $x_k^{lv} \in \mathbb{R}_+$ as the number of servers at location l serving demand from $v \in V$:

$$x_k^l = \sum_{v \in V} x_k^{lv}, \quad \forall l \in L, 0 \leq k \leq K. \tag{3.1}$$

Let $u_k^{lv} \in \mathbb{R}$ denote the change in the number of servers in x_k^{lv} at time k, we then have:

$$x_{k+1}^{lv} = x_k^{lv} + u_k^{lv}, \quad \forall l \in L, v \in V, 0 \leq k \leq K. \tag{3.2}$$

Table 3.1: Table of Notations

Symbol	Meaning
x_k^l	Num. of servers at DC l at time k
x_k^{lv}	Num. of servers at l serving demand from v at time k
D_k^v	Avg. demand arrival rate originated from v
σ_k^{lv}	Avg. arrival rate of demand from v to DC l at time k
u_k^{lv}	Change in the number of servers at DC l at time k
λ_k^{lv}	Avg. arrival rate to each server from v to l at time k
d_{lv}	Network latency between location v and data center l
μ	Request process rate of a single server
p^l	Price of each server at DC l
r	Reservation ratio
C^l	Capacity of DC l
H_k	Resource allocation cost at time k
G_k	Reconfiguration cost at time k
J	Total operational cost

3.4.1 Modeling the Server Allocation and Reconfiguration Cost

To model the cost of server allocation, we assume that there is a price p_k^l for running a server at data center $l \in L$ at time k. The total resource cost R_k for service hosting at time k is

$$R_k = \sum_{l \in L} x_k^l p_k^l \qquad \forall 0 \leq k \leq K \tag{3.3}$$

We also assume that there is a convex function $g : \mathbb{R} \to \mathbb{R}_+$ that computes the cost of reconfiguration. A possible reconfiguration cost function is

$$G_k = \sum_{l \in L} g^l(u_k^l) = \sum_{l \in L} c_{on}^l (\sum_{v \in V} u_k^l)^+ - c_{off}^l (\sum_{v \in V} u_k^l)^-, \forall 0 \leq k \leq K. \tag{3.4}$$

where c^l_{on} and c^l_{off} are the average monetary costs for adding an additional VM and removing a VM, respectively. For example, the c^l_{on} can measure the performance penalty during the startup time of a server, as mentioned in 3.3. Our framework can also support other reconfiguration cost functions as well.

3.4.2 Modeling the Performance Objectives

While minimizing the total operational cost, the allocation of servers and demand assignment must satisfy a set of constraints, including (1) demand constraint and (2) SLA performance constraint. Define σ^{lv}_k as the demand arrival rate from v assigned to data center l at time k, the demand constraint ensures that all demands are satisfied:

$$\sum_{l \in L} \sigma^{lv}_k = D^v_k, \quad \forall v \in V,\ l \in L,\ 0 \leq k \leq K. \tag{3.5}$$

In addition, there is a SLA performance constraint that specifies a maximum delay $\bar{d}_{lv}$ that the service provider tries to achieve between a location v and a data center l. We focus on modeling this constraint in the rest of this subsection. For data center $l \in L$, we assume that there is a load balancer placed in data center l such that demand $\sigma^l_k = \sum_{v \in V} \sigma^{lv}_k$ arriving from location v is equally split among the local servers x^l_k. Assuming each server has mean service time μ, the utilization of a server can be defined as $\rho^l_k = \frac{\sigma^l_k}{x^l_k \mu}$. We assume the queuing delay of a server is a convex function $f(\cdot)$ of the server utilization ρ^l_k:

$$q^l_k = f(\rho^l_k) \tag{3.6}$$

This is true for most of the queuing systems. For example, if each server can be modeled as a $G/G/1$ queue, then the queueing delay can be approximated by [70]:

$$q^l_k = \left(\frac{\rho^l_k}{1 - \rho^l_k}\right)\left(\frac{c_a^2 + c_s^2}{2}\right) \cdot \frac{1}{\mu} \tag{3.7}$$

where c_a and c_s are the coefficient of variation of arrival rate (i.e. $\frac{\sigma^l_k}{x^l_k}$) and service time, respectively. It is evident that this is convex in ρ^l_k. This also holds for other queueing systems such as $M/M/1$ queues.

We aim to ensure that for any $(v, l) \in E$ with $\sigma_k^{lv} > 0$, the average delay (i.e., the sum of propagation and queuing delay) is less than $\bar{d}$[1]:

$$d^{lv} + q_k^l \leq \bar{d}, \quad \forall v \in V, l \in L, 0 \leq k \leq K. \tag{3.8}$$

By defining the constant

$$a^{lv} = \begin{cases} \frac{1}{f^{-1}(\bar{d}-d^{lv})\mu}, & \text{if } \bar{d} - d_{lv} > 0, \\ \infty, & \text{otherwise,} \end{cases} \tag{3.9}$$

we can rewrite the constraint (3.8) as:

$$x_k^{lv} \geq a^{lv}\sigma_k^{lv}, \qquad \forall v \in V, l \in L. \tag{3.10}$$

We can combine constraints (3.5) and (3.10) to eliminate σ_k^{lv}:

$$\sum_{l \in L} \frac{x_k^{lv}}{a^{lv}} \geq D_k^v, \qquad \forall, v \in V, 0 \leq k \leq K. \tag{3.11}$$

Alternatively, we can model the performance objective as a SLA penalty function. Assume there is a convex penalty function $h(\cdot)$ that measures revenue loss due to exceeding the delay requirement. Thus the total performance penalty can thus be measured as:

$$P_k = \sum_{v \in V} h(\sum_{l \in L} \frac{x_k^{lv}}{a^{lv}} - D_k^v) \tag{3.12}$$

3.4.3 Demand Assignment Policy

We can define the demand assignment policy for each request router as:

$$\sigma_k^{lv} = D_k^v \cdot \frac{\frac{x_k^{lv}}{a^{lv}}}{\sum_{l \in L} \frac{x_k^{lv}}{a^{lv}}}. \tag{3.13}$$

Imposing constraint (3.13) implies that SLA requirement is met by all request routers. In practice, each request router for location $v \in V$ can implement the policy by splitting the demand D_k^v proportionally according to equation (3.13) using any standard load balancing technique.

[1]It should be pointed out that even though our model focuses on guaranteeing the average delay, it is straightforward to extend it to handle more general cases, such as ϕ-percentile delay (where ϕ is typically 95%). For example, the ϕ-percentile delay of an $M/M/1$ queue can be computed as $q = \frac{\ln(\frac{1}{1-\phi})}{\mu - \sigma_k^l}$, where $\ln\left(\frac{1}{1-\phi}\right)$ is a constant.

3.4.4 Modeling the Capacity Constraint

We also assume each data center has finite resource capacities, this implies the number of VMs that can be launched in each data center is upper-bounded by its capacity. In the simplest case where all servers have the same size, we can specify the capacity constraint as $x_k^l \leq C^l$, $\forall l \in L$ where C^l is the number of launchable servers in data center l. In our model, we adopt a more general model that captures the capacity constraint as a penalty function. In particular, assuming each data center possesses R types of resources, let s^r denote the size of a server for a resource type $r \in R$, and C^{lr} is the capacity of data center l for resource type r, we can use a convex penalty function $\pi^r(s^r x_k^l)$ to measure the penalty due to violation of capacity constraint. For example, we can define

$$C_k = \pi^r(s^r x_k^l) = \zeta(s^r x_k^l - C^r)^+ \forall l \in L, k \in \mathcal{K}, r \in R$$

where ζ is a very large constant that severely penalizes the violation of capacity constraints. The purpose of using a penalty function is that it admits more general forms of capacity constraints, as we shall further elaborate in Section 3.6.2.

3.4.5 DSPP formulation

Given the system model described above, the goal of DSPP is to minimize the total cost of operating a given service. In other words, the goal of DSPP is to minimize the following objective function:

$$J := \sum_{k=0}^{K} H_k + G_k + P_k + C_k$$

subjects to constraints (3.2) and the requirement that $x_k^{vl} \geq 0 \, \forall l \in L, k \in \mathcal{K}$.

Define $\mathbf{x}_k = [x_k^{11}, ...x^{L1}, ..., x_k^{lv}, ..., x_k^{LV}]^\top \in \mathbb{R}_+^{LV}$, $\mathbf{p}_k^{'} = [p_k^1, p_k^2, ..., p_k^L] \in \mathbb{R}_+^L$, $\mathbf{p}_k = [\mathbf{p}_k^{'}, \mathbf{p}_k^{'}, ...\mathbf{p}_k^{'}]^\top \in \mathbb{R}_+^{LV}$, $\mathbf{u}_k = [u_k^{11}, ...u^{L1}, ..., u_k^{lv}, ..., u_k^{LV}]^\top \in \mathbb{R}^{LV}$, $\mathbf{a}_k^v = [\frac{1}{a_k^{1v}}, \frac{1}{a_k^{2v}}, ..., \frac{1}{a_k^{Lv}}]^\top$, $\mathbf{a}_k = \text{diag}^{-1}\{\mathbf{a}_k^1, ..., \mathbf{a}_k^V\} \in \mathbb{R}_+^{LV \times V}$, $g_k : \mathbb{R}^L \to \mathbb{R}^L$ that captures the reconfiguration cost, $\mathbf{R} = [1, 1, ..., 1]^\top \in \mathbb{R}_+^L$, $\mathbf{B} = \text{diag}^{-1}\{\mathbf{R}, ..., \mathbf{R}\} \in \mathbb{R}_+^{LV \times V}$, $\mathbf{D}_k = [D_k^1, ..., D_k^V]^\top$, $\mathbf{C} = [C^1, C^2, ...C^L]^\top \in \mathbb{R}_+^L$, $\mathbf{s} = [\mathbf{I}^{L \times L}, ..\mathbf{I}^{L \times L}]^\top \in \mathbb{R}_+^{LV \times L}$, we can rewrite DSPP as:

$$\begin{aligned}
\min_{\{\mathbf{u}_0, .., \mathbf{u}_{K-1}\}} \quad & J = \sum_{k=0}^{K} \mathbf{p}_k^\top \mathbf{x}_k + \mathbf{R}^\top g_k(\mathbf{B}\mathbf{u}_k) \quad + P_k(\mathbf{a}_k \mathbf{x}_k - \mathbf{D}_k) + \pi(\mathbf{s}^\top \mathbf{x}_k) \\
\text{s.t} \quad & \mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{u}_k \qquad \forall k \in \mathcal{K}, \\
& \mathbf{x}_k \in \mathbb{R}_+^{LV} \qquad \forall k \in \mathcal{K}
\end{aligned}$$

3.4.6 Example: An Extension to Multi-Tiered Applications

The DSPP model presented in previous section is generic enough to capture many different service architectures. In this section, we present one particular architecture: multi-tiered applications. Particully, in previous sections we assume that each service is hosted in a single VM that can operate independently of other VMs. However, this model may be over simplified in many cases. For example, many online services may consist of multiple tiers of applications, where different tiers may be placed in different data centers and use VMs of different size. Furthermore, these applications may or may not be placed in a single data center. For example, for many online services where data is partitioned and stored across multiple data centers, in many cases it is necessary for an application to retrieve the data remotely from a different data center. To deal with these issues, in this section we show how DSPP formulation can model the placement problem of multi-tiered applications across geographically distributed data centers.

We assume the service to be placed consists of n tiers of components, each tier $1 \leq i \leq n$ can be assigned to a subset of data centers $L_i \subseteq L$. We shall use a 3-tier web application shown in Figure 3.3 to illustrate our model. As is the typical case, the 3-tier web application consists of (1) front-end web servers, (2) application servers that run the business logic, and (3) database servers who have access to data storage. Let D_k^v denote the arrival rate of type v service request from access network $v \in V$ during time period k. Each service request will invoke internal requests among service applications. In the context of multi-tiered services, the service invocation for a single request can be modelled as a flow along a path from the front-end to back-end servers. Define $P_s \subseteq V \times L_1 \times L_2 \times ... \times L_n$ as set of possible path requests are routed . Let the demand along each flow path p be $\sigma_k^p \in \mathbb{R}_+$. Furthermore, define $P(n, l) \subseteq P$ as the set of flow paths that has tier n service routed through data center $l \in L_n$, to simplify our notation, we assume V represents the 0th tier of the service, as shown in Figure 3.1. Let x_k^{pnl} represent the number of servers at tier n in data center $l \in L_n$ at time k, and u_k^{pnl} as change in x_k^{pnl} at time k respectively, we thus have the following state equation:

$$x_k^{pnl} = x_k^{pnl} + u_k^{pnl} \qquad \forall n \in N, l \in L, p \in P, k \in \mathcal{K}$$

The total resource cost R_k for service hosting at time k is

$$R_k = \sum_{n=1}^{N} \sum_{l \in L} \sum_{p \in P} x_k^{pnl} p_k^{nl} \qquad \forall 0 \leq k \leq K \tag{3.14}$$

We also assume that there is a convex function $g : \mathbb{R} \to \mathbb{R}_+$ that computes the cost of reconfiguration. A possible reconfiguration cost function is

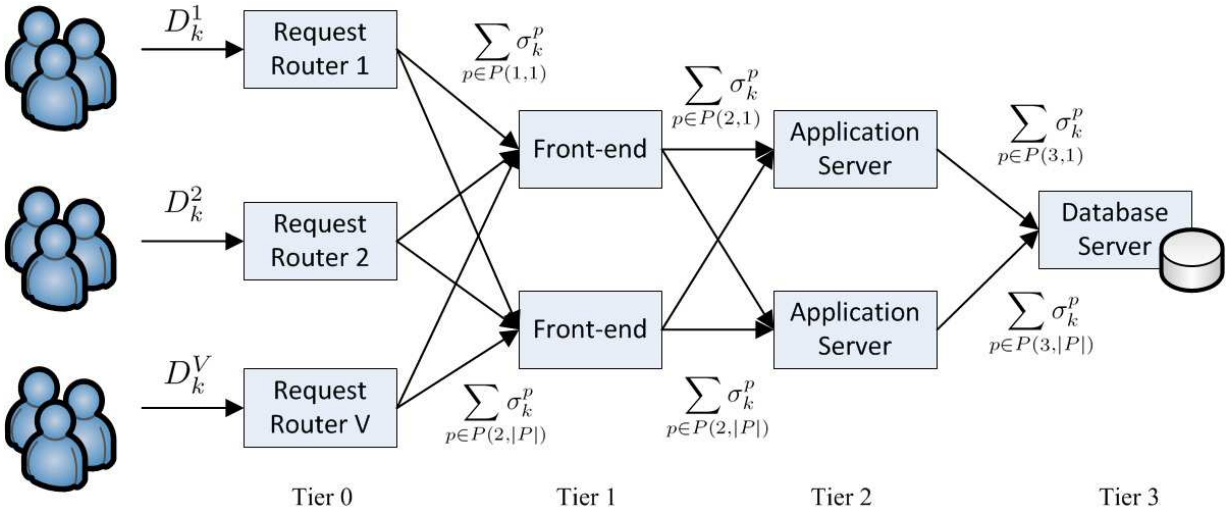


Figure 3.3: Service Placement Model

$$
\begin{aligned}
G_k &= \sum_{n=1}^{N} \sum_{l \in L_n} g^{nl}\Big(\sum_{p \in P(n,l)} u_k^{pnl} \Big) \\
&= \sum_{n=1}^{N} \sum_{l \in L_n} c_{on}^{nl}\Big(\sum_{p \in P(n,l)} u_k^{pnl} \Big)^{+} - c_{off}^{nl}\Big(\sum_{p \in P(n,l)} u_k^{pnl} \Big)^{-}.
\end{aligned} \tag{3.15}
$$

We also know that total tier n demand that arrives at data center l at time k can be computed as $\lambda_k^{nl} = \sum_{p \in P(n,l)} \lambda_k^p$. We assume at each data center, there is a load balancer that spreads the total demand evenly across all servers. Assuming the service time for a tier n server is μ^n, the utilization of a tier n server at data center l can be computed as

$$
\rho_k^{nl} = \frac{\sum_{p \in P(n,l)} \lambda_k^p}{x_k^{nl} \mu^n} \tag{3.16}
$$

Similar to our previous model, we assume the queuing delay q_k^{nl} of tier n servers at DC l is a convex function of ρ_k^{nl}. Similar to existing work [106], we want to ensure that the latency for each tier of service is bounded, namely

$$
q_k^{nl} + d_{l_{n-1} l_n} \leq \bar{d}^n \qquad \forall p \in P \tag{3.17}
$$

By defining the constant

$$a^{pnl} = \begin{cases} \frac{1}{f^{-1}(\bar{d}^n - d_{l_{n-1}l_n})\mu^n}, & \text{if } \bar{d} - d_{l_{n-1}l_n} > 0, \\ \infty, & \text{otherwise,} \end{cases} \tag{3.18}$$

It is easy to see that the we can rewrite the performance objective (i.e. constraint (3.17)) as

$$\sum_{l \in L} \sum_{p \in P(0,v) \cup P(n,l)} \frac{x_k^{pnl}}{a_k^{pnl}} \geq D_k^v \qquad \forall p \in P \tag{3.19}$$

Alternatively, we can model the performance objective as a SLA penalty function. Assume there is a convex penalty function $h(\cdot)$ that measures revenue loss due to exceeding the delay requirement. Thus the total performance penalty can thus be measured as:

$$P_k = \sum_{v \in V} \sum_{n \in N} h(\sum_{l \in L} \sum_{p \in P(0,v) \cup P(n,l)} \frac{x_k^{pnl}}{a_k^{pnl}} - D_k^v) \tag{3.20}$$

We thus have the following objective function:

$$\begin{aligned} \min \quad & \sum_{k=1}^{K} R_k + G_k + P_k + C_k \\ \text{s.t} \quad & x_k^{pnl} = x_k^{pnl} + u_k^{pnl} && \forall l \in L, v \in V, k \in \mathcal{K} \\ & x_k^{pnl} \in \mathbb{R}_+, u_k^{pnl} \in \mathbb{R} && \forall l \in L, v \in V, k \in \mathcal{K} \end{aligned}$$

Define $\mathbf{x}_k = [x_k^{111}, ...x^{pnl}, ..., x_k^{|P|NL}]^\top \in \mathbb{R}_+^{|P|NL}$, $\mathbf{p}_k^{'} = [p_k^1, p_k^2, ..., p_k^L]^\top \in \mathbb{R}_+^L$, $\mathbf{p}_k = [\mathbf{p}_k^{'}, \mathbf{p}_k^{'}, ..., \mathbf{p}_k^{'}] \in \mathbb{R}_+^{|P|NL}$, $\mathbf{u}_k = [u_k^{111}, ...u^{pnl}, ..., u_k^{|P|NL}]^\top \in \mathbb{R}_+^{|P|NL}$. $g_k : \mathbb{R}^L \to \mathbb{R}^L$ that captures the reconfiguration cost, $\mathbf{R} = [1, 1, ..., 1]^\top \in \mathbb{R}_+^L$, $\psi^{pnl} \in \{0, 1\}$ as a boolean variable that indicates whether the nth hop of path p goes through l. $\mathbf{B}^{nl} = [\psi^{1nl}, ..., \psi^{|P|nl}] \in \mathbb{R}_+^{|P|}$, $\mathbf{B} = [\mathbf{B}^{11\top}, ..., \mathbf{B}^{1L\top}, ...\mathbf{B}^{NL\top}] \in \mathbb{R}_+^{|P|NL}$, $\mathbf{D}_k = [D_k^1, ..., D_k^V]^\top$, $\mathbf{C} = [C^1, C^2, ...C^L]^\top \in \mathbb{R}_+^L$, $\mathbf{s} = [\mathbf{I}^{L\times L}, ..\mathbf{I}^{L\times L}]^\top \in \mathbb{R}_+^{LV\times L}$. Furthermore, define $\theta^{pnl} \in \{0, 1\}$ as a boolean variable that indicates whether the nth hop of path p goes through $l \in L \cup V$. Also, define $\mathbf{a}_k^{pn} = [\frac{1}{a_k^{pn1}}, ..., \frac{1}{a_k^{pnL}}] \in \mathbb{R}_+^L$, $\mathbf{a}_k^n = \begin{bmatrix} \mathbf{a}_k^{1n} \cdot \theta^{p01} & \cdots & \mathbf{a}_k^{|P|n} \cdot \theta^{p01} \\ \vdots & \ddots & \vdots \\ \mathbf{a}_k^{1n} \cdot \theta^{p0V} & \cdots & \mathbf{a}_k^{|P|n} \cdot \theta^{p0V} \end{bmatrix} \in \mathbb{R}_+^{|P|L\times V}$, $\mathbf{a}_k =$

Algorithm 2 MPC Algorithm for DSPP

1: Provide initial state $\mathbf{x}_0$, $k \leftarrow 0$
2: **while true do**
3: At beginning of control period k:
4: Predict $\mathbf{D}^l_{k+i|k}$ for horizons $i = 1, \cdots, K$ using a demand prediction model
5: Solve DSPP to obtain $\mathbf{u}_{k+t|k}$ for $t = 0, \cdots, W-1$
6: Change the resource allocation according to $\mathbf{u}_{k|k}$
7: Update demand assignment policy of request routers according to equation (3.13)
8: $k \leftarrow k+1$
9: **end while**

$\text{diag}^{-1}[\mathbf{a}^1_k, ..., \mathbf{a}^n_k] \in \mathbb{R}_+^{|P|NL \times NV}$, $\mathbf{D}'_k = [D^1_k, ..., D^V_k]^\top \in \mathbb{R}^V_+$, $\mathbf{D}_k = [D'^\top_k, ..., D'^\top_k]^\top \in \mathbb{R}^{NV}_+$, $\mathbf{s}^r = \begin{bmatrix} \theta^{p11} & \dots & \theta^{p11} \\ \vdots & \ddots & \vdots \\ \theta^{pNL} & \dots & \theta^{pNL} \end{bmatrix} \cdot s^r$, $\mathbf{s} = [\mathbf{s}^{1\top}, ..., \mathbf{s}^{R\top}]$, we can rewrite DSPP as:

$$\begin{aligned} \min_{\{\mathbf{u}_0,..,\mathbf{u}_{K-1}\}} \quad & J = \sum_{k=0}^{K} \mathbf{p}_k^\top \mathbf{x}_k + \mathbf{R}^\top g_k(\mathbf{B}\mathbf{u}_k) \quad + P_k(\mathbf{a}_k \mathbf{x}_k - \mathbf{D}_k) + \pi(\mathbf{s}^\top \mathbf{x}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{u}_k \qquad \forall k \in \mathcal{K}, \\ & \mathbf{x}_k \in \mathbb{R}^{LV}_+ \qquad \forall k \in \mathcal{K} \end{aligned}$$

which is exactly our DSPP model as described in Section 3.4.5. Thus, a solution technique for DSPP can also be used to solve the mult-tiered service placement problem.

3.5 Controller design for DSPP

DSPP is a convex optimization problem that can be solved optimally using standard methods [37] However, even though DSPP can be solved optimally, the resource controller must solve this problem in an online fashion where future demand is unknown. In this case, we use the Model Predictive Control (MPC) framework that is widely used for solving online control problems. Algorithm 2 is our MPC algorithm used by the resource controller for solving DSPP online. It can be described as follows. At time k, the resource controller predicts the future demand $\mathbf{D}^v_k$ for multiple periods $[k+1, ..., k+W]$. Using the demand predicted by the analysis and prediction module, where W is the prediction horizon. Denote by $\mathbf{D}^v_{k+t|k}$ the demand predicted for time $k+t$ at time k. The controller then solves

the optimization problem for the horizon $[k, ..., k + W]$, starting with the initial state $\mathbf{x}_{k|k} = \mathbf{x}_k$. Even though the solution of the optimization problem will contain a set of values $\mathbf{u}_{k|k}, ..., \mathbf{u}_{k+W-1|k}$, the controller will only execute the first step in sequence $\mathbf{u}_{k|k}$. When the next control period $k + 1$ starts, the same procedure is performed again by the controller. Using this MPC algorithm, the controller can effectively adjust the number of servers in each data center.

3.6 Competition Among Multiple Providers

In this section, we extend our previous model and consider the case where multiple service providers share the cloud platform in terms of resources in data centers. The goal of each service provider is to minimize its operational costs while respecting the SLA performance requirements and the data center capacity constraints. In our model, we assume that the placement configuration of each service provider is kept private from other service providers. In this scenario, strategic interactions may arise as each service provider makes decisions independently. Therefore, we can model the system as a multi-person non-cooperative game. Our objective is to analyze the equilibrium outcome, and design appropriate algorithms if the resulting competition leads to sub-optimal outcome in terms of social welfare. Generally speaking, the social welfare is defined as the sum of the service revenue minus the resource rental cost of all the service providers. In our case, the service revenue of each service provider is fixed assuming that it can provision resources to satisfy its demand. As a result, the social welfare is maximized when the sum of the total resource rental cost of all service providers is minimized.

3.6.1 Player Model

We formally define the resource competition game in this section. Define $\mathcal{N} = \{1, 2, ..., N\}$ as the set of service providers, and let $i \in \mathcal{N}$ represent the index of each service provider. Let $\mathcal{K} = \{0, 1, 2, ...K\}$ denote the set of stages (i.e., time indices) of the game. At time k, $0 \leq k \leq K$, each service provider i has a state $\mathbf{x}_k^i$ that describes the number of servers allocated in data center $l \in L$, as defined in the previous section. Each service provider i also makes a control decision $\mathbf{u}_k^i$ at time $k \in \mathcal{K}$, where u_k^{il} denotes the change in the number of servers at data center $l \in L$ at time k. Given an initial system state $\mathbf{x}_0^i$, the system dynamics are captured by the following state equation:

$$\mathbf{x}_{k+1}^i = \mathbf{x}_k^i + \mathbf{u}_k^i \quad \forall i \in \mathcal{N}, v \in V, k \in \mathcal{K} \tag{3.21}$$

At time $k \in \mathcal{K}$, we assume each service provider i has a demand $\mathbf{D}_k^i = [D_k^{i1}, D_k^{i2}, ..., D_k^{iv}]^\top$. Lastly, we define $\mathbf{u}^i = \{\mathbf{u}_0^i, \mathbf{u}_1^i, ..., \mathbf{u}_{K-1}^i\}$ $\mathbf{x}^i = \{\mathbf{x}_0^i, \mathbf{x}_1^i, ..., \mathbf{x}_{K-1}^i\}$. Furthermore, let $\mathbf{u}^{-i} = \{\mathbf{u}^1, ..., \mathbf{u}^{(i-1)}, \mathbf{u}^{(i+1)}, ..., \mathbf{u}^N\}$ represent the control decisions of the other service providers $N \backslash \{i\}$, the objective of service provider i is to minimize its cost function:

$$J^i(\mathbf{u}^i, \mathbf{u}^{-i}) = \sum_{k=0}^{K} \mathbf{p}_k^\top \mathbf{x}_k^i + \mathbf{R}^{i\top} g_k^i(\mathbf{u}_k^i) \quad + P_k(\mathbf{a}_k^i \mathbf{x}_k^i - \mathbf{D}_k^i) + \pi^i(\textstyle\sum_{i \in \mathcal{N}} \mathbf{s}^i \mathbf{x}_k^i)$$

where $\mathbf{a}_k^i$ is defined as in Section 3.4.5 for each service provider i. subject to equation (3.21) and the following constraint, which specifies the number of servers per location must remain non-negative:

$$\mathbf{x}_k^i \succeq 0 \forall i \in \mathcal{N}, k \in \mathcal{K}. \tag{3.22}$$

3.6.2 Modeling Capacity Constraint

In this section, we discuss how capacity constraint is modeled in an environment where resources are shared among multiple service providers. Given a set of resources (e.g. CPU, memory and disk) R, define $\mathbf{C_r} = [C_r^1, C_r^2, ...C_r^L] \in \mathbb{R}_+^L$ as a vector that captures the capacity of resource type $r \in R$, in each data center $l \in L$. Define $\mathbf{C} = [\mathbf{C}_1, ..., \mathbf{C}_R]^\top$, we capture the capacity requirement using a convex penalty function $\pi^i(\cdot)$. The benefit of using the penalty function $\pi^i(\cdot)$ is that it supports a variety of ways in which capacity constraints are modeled in practice:

- The capacity constraint is *exact* if the resources in each data center can be optimally allocated to servers without wastage. This is a valid assumption if all VMs have the same size. Even though in many practical situations where this assumption may not hold in general, there is strong incentive for InPs to do so. In practice, InPs typically design VM sizes to match physical machine capacities to reduce resource wastage. An example is the GoGrid public cloud service, which offers VMs in 6 different types. Arranged from the smallest to the largest, each type of VMs has exactly twice the size of previous type in terms of CPU, memory and disk capacity. When VM sizes are multiples of each other, bin-packing can be solved optimally using First-Fit-Decrease (FFD) policy, and no resource is wasted during the process. In this case, our model can be applied exactly to the case of GoGrid. We can specify a hard capacity constraint

$$\sum_{i \in \mathcal{N}} \mathbf{s}^i \mathbf{x}_k^i \preceq \mathbf{C}, \quad 0 \leq k \leq K. \tag{3.23}$$

which can be modeled as

$$\pi^i(\sum_{i\in\mathcal{N}} \mathbf{s}^i\mathbf{x}_k^i) = \zeta(\sum_{i\in\mathcal{N}} \mathbf{s}^i\mathbf{x}_k^i - \mathbf{C})^+$$

where ζ is a very large constant that severely penalizes the violation of capacity constraints. In practice, the CP will inform the service provider about the number of launcheable VMs, allowing the service provider to make informed placement decisions.

- In contrast, in many private cloud systems where launching a new server implies scheduling a new VM, the resource capacity constraints are hardly exact, as determining whether a VM scheduled requires solving a vector bin-packing problem. In this context, we can measure the penalty for approaching the data center capacity in terms of VM *scheduling delay,* which is the time it takes the request to be scheduled. Scheduling delay is an important penalty cost as it directly affects the availability of the VM. Even though the detailed analysis of scheduling delay will be presented in Chapter 4, in this chapter, we use the property that scheduling delay is a convex function of the utilization of bottleneck resource, as reported in recent work [127, 128, 131]. In this case, we can model the scheduling delay of a data center l as a convex function of resource utilization $T(\cdot)$

$$q^l = T(\max_{r\in R}(\frac{\sum_{i\in\mathcal{N}}\sum_{i\in\mathcal{N}} s^{ir}x^{il}}{C^{lr}}))$$

where the exact form of T can be obtained either through formal models or experiments [127, 128]. Therefore, we can model $\pi(\sum_{i\in\mathcal{N}} \mathbf{s}^i\mathbf{x}_k^i) = [q^1, q^2, ..., q^L] \cdot \upsilon^i$ where υ^i represents the unit penalty cost for scheduling delay. Notice that $\pi(\cdot)$ is a convex function of $\sum_{i\in\mathcal{N}} \mathbf{s}^i\mathbf{x}_k^i$. In practice, the CP will inform the service provider about the expected scheduling delay, allowing the service provider to make informed placement decisions. Such a service is already available, e.g. in Google Data Centers.

To summarize, we assume each data center has finite resource capacity for hosting services, which can be model as a penalty function of resource demand (i.e. $\sum_{i\in\mathcal{N}} \mathbf{s}^i\mathbf{x}_k^i$) and resource available in each data center (i.e. $\mathbf{C}$). Thus, the multi-service provider service placement problem can be modelled as resource competition game, where service providers compete for resources in their prefered data centers. This raises the question of whether such competition will lead to an efficient outcome (i.e. the outcome where the total cost of all the service providers is minimized), and if not, whether there exist an mechanism that can be implemented to incentivize desired behaviour from service providers to achieve an efficient outcome.

3.6.3 Game Analysis

In Game theory, the outcome of a game is captured by the concept of Nash equilibrium (NE), which is an equilibrium state where no player can selfishly improve its cost without violating constraints. We now characterize the Nash equilibrium (NE) of the resource competition game. The NE refers to the stable outcome of the competition, where no service provider can improve its cost by unilaterally changing its server allocation over time. Formally, the resource competition game can be represented as a N-player dynamic non-cooperative game Ξ. Notice that as our controller relies on the MPC framework for dynamic resource allocation, we need to introduce a new version of NE for control strategies using the MPC framework. We first start with the following general definitions:

Definition 1 (η-Nash Equilibrium [27])**.** *Let $\mathcal{I}_k^i$ be the information set of a service provider i at time k under a given information structure η^i, and Γ^i is the set of all admissible policies of service provider i under η^i. The policy $\{\gamma^{i*}, i \in \mathcal{N}\}$ is an $\eta-$Nash equilibrium of the game Ξ, where $\mathbf{u}^i = \gamma^{i*}(\mathcal{I}_k^i)$ and $\eta = \{\eta^i, i \in \mathcal{N}\}$ if $J^i(\gamma^{i*}, \gamma^{-i*}) \leq J^i(\gamma^i, \gamma^{-i*})$, for all admissible policies $\gamma^i \in \Gamma^i$ and for all $i \in \mathcal{N}$, where $\gamma^{-i*} = \{\gamma^j : j \neq i, j \in \mathcal{N}\}$.*

Definition 1 provides a general description of NE under a given information structure (IS) η^i. The dynamic game Ξ can admit different NEs under different information structures η. Typical information structures are, for example, open-loop IS, where the policy is only dependent on the initial conditions, and the perfect-state feedback IS, where the policy depends on the perfect measurement of the system state. The IS under MPC algorithms in Algorithm 1 can be deemed a special mixture between open-loop IS and feedback IS since at each stage each service provider computes within a window in an open-loop manner but the initial condition of the computation is the current state known to service providers. With this special IS, we can define NE under MPC-type computations for our resource competition game.

Definition 2 ($\mathbf{W}$-MPC Nash Equilibrium)**.** *Let W^i be the prediction window of service provider i and every service provider adopts MPC as outlined in Algorithm 1. The dynamic non-cooperative game Ξ admits $\mathbf{W}-$MPC Nash Equilibrium, $\mathbf{W} = \{W^i, i \in \mathcal{N}\}$, if the sequences $\mathbf{u}^{i*} := \{\mathbf{u}_k^{i*}, 0 \leq k \leq K\}$ obtained under MPC algorithms satisfy $J^i(\mathbf{u}^{i*}, \mathbf{u}^{-i*}) \leq J^i(\mathbf{u}^i, \mathbf{u}^{-i*})$, for all admissible sequences $\mathbf{u}^i \in \mathcal{U}^i$ and for all $i \in \mathcal{N}$, where $\mathcal{U}^i$ is the set of admissible control sequences under MPC algorithms, and $\mathbf{u}^{-i*} = \{\mathbf{u}^j, j \neq i, j \in \mathcal{N}\}$.*

Note that NE solutions may not be unique, and hence we let $\mathcal{U}^*$ to denote the set of NE solutions $\mathbf{u}^* := \{\mathbf{u}^i, \mathbf{u}^{-i}\}$ that satisfy Definition 2. The $\mathbf{W}-$MPC Nash equilibrium

$\{\mathbf{u}^{i*}, i \in \mathcal{N}\}$ can be used to compare with the optimal MPC solution $\{\mathbf{u}^{i\circ}, i \in \mathcal{N}\}$ to the solution of the following Social Welfare Problem (SWP):

$$\min_{\{\mathbf{u}^1,...,\mathbf{u}^N\}} \quad \sum_{i\in\mathcal{N}} J^i(\mathbf{u}^1, ..., \mathbf{u}^N)$$

subject to equation (3.21) and (3.22). The NE is defined as:

$$J^i(\mathbf{u}^*) = \min_{\mathbf{u}^i \in \mathbb{R}^{LV}} J^i(\mathbf{u}^i, \mathbf{u}^{-i*}) \qquad \forall i \in \mathcal{N}$$

The price of anarchy (PoA) ρ_{MPC} and the price of stability (PoS) ξ_{MPC} of the dynamic non-cooperative game Ξ under centralized MPC Algorithm 1 are defined by

$$\begin{aligned} \rho_{\text{MPC}} &= \sup_{\mathbf{u}^*\in\mathcal{U}^*} \frac{\sum_{i\in\mathcal{N}}\sum_{v\in V} J_v^i(\mathbf{u}^{i*})}{\sum_{i\in\mathcal{N}}\sum_{v\in V} J_v^i(\mathbf{u}^{i\circ})} \\ \xi_{\text{MPC}} &= \inf_{\mathbf{u}^*\in\mathcal{U}^*} \frac{\sum_{i\in\mathcal{N}}\sum_{v\in V} J_v^i(\mathbf{u}^{i*})}{\sum_{i\in\mathcal{N}}\sum_{v\in V} J_v^i(\mathbf{u}^{i\circ})}, \end{aligned}$$

where $\{\mathbf{u}^{i\circ}, i \in \mathcal{N}\}$ is the optimal solution to (SWP) obtained by the MPC algorithm 2, and $\{\mathbf{u}^{i*}, i \in \mathcal{N}\}$ is the $\mathbf{W}-$MPC Nash equilibrium of the game Ξ. The metrics ξ_{MPC} and ρ_{MPC} are measures of the best-case and worse-case efficiency loss of the game, respectively. It is easy to observe that both ρ_{MPC} and ξ_{MPC} are always greater or equal to 1.

Theorem 1. *Assume that the prediction horizon of each service provider $i, i \in \mathcal{N}$, is the same, i.e., $W^i = W$ and W is also the prediction window used for (SWP). Then, the price of stability ξ_{MPC} of the game Ξ is equal to 1, i.e., there exists a NE solution that yields no efficiency loss under the common knowledge of the capacity constraint.*

Proof. Since each service provider uses window size W in the MPC algorithm, at time k each service provider i solves the following problem:

$$\begin{aligned} \min_{\{\mathbf{u}_0,..,\mathbf{u}_{W-1}\}} \quad & J_W^i = \sum_{k=0}^{W} \mathbf{p}_k^\top \mathbf{x}_k^i + \mathbf{R}^\top g_k(\mathbf{B}\mathbf{u}_k^i) \quad + P_k^i(\mathbf{a}_k\mathbf{x}_k^i - \mathbf{D}_k^i) + \pi^i(\sum_{i\in\mathcal{N}} \mathbf{s}^i\mathbf{x}_k^i) \\ \text{s.t} \quad & \mathbf{x}_{k+1}^i = \mathbf{x}_k^i + \mathbf{u}_k^i \qquad\qquad \forall 0 \le k \le W \\ & \mathbf{x}_k \succeq 0 \qquad\qquad \forall 0 \le k \le W \end{aligned}$$

Since $\mathbf{x}_k^i = \mathbf{x}_0^i + \sum_{k'=0}^{k-1} \mathbf{u}_{k'}^i$ for $k > 0$, we can rewrite the optimization problem as:

$$\begin{aligned}
\min_{\{\mathbf{u}_0,..,\mathbf{u}_{W-1}\}} \quad & J_W^i = \sum_{k=1}^{W} \sum_{k'=0}^{k-1} \mathbf{p}_k^\top \mathbf{u}_{k'}^i + \sum_{k=0}^{W-1} \mathbf{R}^\top g_k(\mathbf{B}\mathbf{u}_k^i) + \\
& \sum_{k=1}^{W} P_k^i(\mathbf{a}_k \mathbf{x}_0 + \mathbf{a}_k \sum_{k'=0}^{k-1} \mathbf{u}_{k'} - \mathbf{D}_k) + \pi^i(\sum_{i\in\mathcal{N}} \mathbf{s}^i(\mathbf{x}_0^i + \sum_{k'=0}^{k-1} \mathbf{u}_{k'}^i)) \\
\text{s.t.} \quad & \mathbf{x}_0^i + \sum_{k'=0}^{k-1} \mathbf{u}_{k'}^i \succeq 0, \ \ \forall 0 \leq k \leq W
\end{aligned}$$

Each service provider faces an internal constraint shown above. We can associate each internal constraint with Lagrange multipliers $\mu^i, i \in \mathcal{N}$ and the coupled constraint with ν. The Lagrangian of service provider i is given by

$$\mathcal{L}^i = J^i - \mu^i(\mathbf{x}_0 + \sum_{k'=0}^{k-1} \mathbf{u}_{k'}) \tag{3.24}$$

On the other hand, the SWP problem can be captured by the following problem at every time k:

$$\min_{\{\mathbf{u}^1,\ldots,\mathbf{u}^N\}} \quad \sum_{i\in\mathcal{N}} J_W^i \tag{3.25}$$

$$\mathbf{x}_0 + \sum_{k'=0}^{k-1} \mathbf{u}_{k'} \succeq 0 \tag{3.26}$$

By associating Lagrangian multipliers $\tilde{\mu}^i, i \in \mathcal{N}$ and with constraints (3.26), we have the Lagrangian of the social welfare problem

$$\mathcal{L} = \sum_{i\in\mathcal{N}} J_W^i - \sum_{i\in\mathcal{N}} \tilde{\mu}^i(\mathbf{x}_0 + \sum_{k'=0}^{k-1} \mathbf{u}_{k'}) \tag{3.27}$$

By letting $\mu^i = \tilde{\mu}^i$, and $\nu = \tilde{\nu}$, we can further decompose $\mathcal{L}$ into $\mathcal{L} = \sum_{i\in\mathcal{N}} \mathcal{L}^i$.Since it is strictly convex and separable in i, the $W-$horizon social welfare problem admits a unique solution, which also corresponds to the solution of each convex subproblem associated with $\mathcal{L}^i$. Hence, the social optimal solution is a NE at every k and the result follows. □

Theorem 2. *The price of anarchy ρ_{MPC} of the game Ξ is unbounded.*

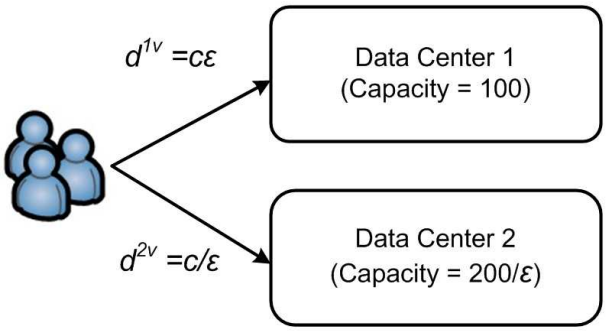


Figure 3.4: Example illustrating the PoA of the game

Proof. We provide an example to illustrate that the price of anarchy is unbounded even when demand of each service provider remains static over time. Consider the scenario illustrated by Figure 3.4: there are two data centers serving demand from a single location v. The data centers have capacities $C^1 = 100$ and $C^2 = \frac{200}{\epsilon}$ respectively. The distance to each data center are $d^{1v} = \epsilon c$, $d^{2v} = \frac{c}{\epsilon}$ respectively, where c and ϵ are constants. Both data centers lease resources at the same unit price p, i.e., $p_k^1 = p_k^2 = p \, \forall k \geq 0$. Furthermore, There are two service providers in the game. Their SLAs are $\bar{d^1} = (1 + \epsilon + \frac{1}{\epsilon})c$ and $\bar{d^2} = (K + 1 + \epsilon + \frac{1}{\epsilon})c$, respectively, where $K \geq 1$ is a constant. For both service providers, a single server can process requests at rate $\mu = \frac{1}{c}$. The demand from location v for both service providers are $D_1 = \frac{100}{c(1+\epsilon)}$, $D_2 = \frac{100}{c(1+\frac{1}{K\epsilon+1})}$. More over, assume all servers have identical size and $\pi^i(\sum_{i\in\mathcal{N}} \mathbf{s}^i \mathbf{x}_k^i) = \zeta(\sum_{i\in\mathcal{N}} \mathbf{s}^i \mathbf{x}_k^i - \mathbf{C})^+$, where $\zeta \in \mathbb{R}_+$ is a large constant. Now, consider the following allocation for both service providers: service provider 2 serves all its demands using capacities in DC 1, and service provider 1 serves all its demands from DC2. It is easy to see this is a NE, as there is no free capacity in DC 1 for service provider 1. The total cost of this NE is $J_{NE1} = \sum_{i\in\mathcal{N}} J^i = (1 + \frac{1}{\epsilon})100p$. Now consider another NE, where service provider 1 uses all the capacities in DC 1, and service provider 2 serves all its demands from DC 2. The total cost of this NE is $J_{NE2} = 100(1 + \frac{1+\frac{1}{K+\epsilon}}{1+\frac{1}{K\epsilon+1}})p$. As $\epsilon \to 0$, we have $\rho_{\text{MPC}} \geq \lim_{\epsilon\to 0} \frac{J_{NE1}}{J_{NE2}} = \lim_{\epsilon\to 0} \frac{1+\frac{1}{\epsilon}}{1+\frac{1+\frac{1}{K}}{2}} = \infty$.

□

3.6.4 Mechanism Design for a Single Infrastructure Provider

The result provided in the previous section is rather discouraging: If each service provider behave selfishly in an uncoordinated manner, then the outcome can be severely unfair: certain service providers will experience much higher cost than others due to insufficient resource capacities in their preferred locations. This also hurts the efficiency of the resulting NE. In this case, a natural question to be asked is whether it is possible to address the inefficiency with proper mechanism design. In this section, we answer this question by analyzing the outcome with the participation of the InP.

Similar to existing work on cloud resource pricing (e.g. [86]), we consider a market where there is a single CP who wishes to maximize the social welfare of all the participants, including both the service providers and the CP itself. For the CP, the selfish objective is to maximize the total revenue from selling resources:

$$U^{CP}(\mathbf{p}, \mathbf{u}^1, ..\mathbf{u}^N) = \sum_{i \in \mathcal{N}} \sum_{k=0}^{K} (\mathbf{p}_k - \mathbf{e}_k) \mathbf{s}^i \mathbf{x}_k^i$$

where $\mathbf{e}_k$ represents the production cost of resources. Typically, $\mathbf{e}_k$ includes the cost of electricity, server and land cost amortized over time. For each service provider, the objective function, as described in the previous section, is to minimize:

$$J^i(\mathbf{p}, \mathbf{u}^1, ..\mathbf{u}^N) = \sum_{i \in \mathcal{N}} \sum_{k=0}^{K} \mathbf{p}_k^\top \mathbf{x}_k^i + \mathbf{R}^{i\top} g_k^i(\mathbf{u}_k^i) + P_k^i(\mathbf{a}_k^i \mathbf{x}_k^i - \mathbf{D}_k^i) - \sum_{k=1}^{K} \pi^i (\sum_{i \in \mathcal{N}} \sum_{v \in V} \mathbf{s}^i \mathbf{x}_k^{iv})$$

The goal of the cloud provider is to maximize the total social welfare, which is to maximize $J^{SW} = U^{CP} - \sum_{i \in N} J^i$. This is a reasonable objective because a CP typically wants to (1) maximize the revenue from selling resources, while (2) improving service satisfaction, which translates into minimizing the performance penalties incurred by individual service providers. The problem of maximizing social welfare can be written as minimizing

$$J^{SW}(\mathbf{p}, \mathbf{u}^1, ..\mathbf{u}^N) = \sum_{i \in \mathcal{N}} \sum_{k=0}^{K} \mathbf{e}_k^\top \mathbf{s}^i \mathbf{x}_k^i + \mathbf{R}^{i\top} g_k^i(\mathbf{u}_k^i) + P_k^i(\mathbf{a}_k^i \mathbf{x}_k^i - \mathbf{D}_k^i) - \sum_{k=1}^{K} \pi^i (\sum_{i \in \mathcal{N}} \sum_{v \in V} \mathbf{s}^i \mathbf{x}_k^{iv})$$

subject to constraints (3.21) and (3.22).

To solve this problem, we design a mechanism using dual decomposition technique. To facilitate the decomposition, we can introduce an ancillary variable $\mathbf{v}_k = \sum_{i\in\mathcal{N}} \sum_{v\in V} \mathbf{s}^i \mathbf{x}_k^i$. The problem can be rewritten as

$$J^{SW}(\mathbf{p}, \mathbf{v}, \mathbf{u}^1, ..\mathbf{u}^N) = \sum_{i\in\mathcal{N}} \sum_{k=0}^{K} \mathbf{e}_k^\top \mathbf{s}^i \mathbf{x}_k^i + \mathbf{R}^{i\top} g_k^i(\mathbf{u}_k^i) + P_k^i(\mathbf{a}_k^i \mathbf{x}_k^i - \mathbf{D}_k^i) - \pi^i(\mathbf{v}_k)$$

$$s.t. \mathbf{v}_k = \sum_{i\in\mathcal{N}} \sum_{v\in V} \mathbf{s}^i \mathbf{x}_k^i$$
$$\mathbf{x}_k \succeq 0$$

subject to the constraint that $\mathbf{v}_k = \sum_{i\in\mathcal{N}} \sum_{v\in V} \mathbf{s}^i \mathbf{x}_k^{iv}$, as well as constraints (3.21) and (3.22). The lagrangian dual problem can be stated as:

$$\max_{\lambda_k}(\quad \inf_{\mathbf{u}_k^i, \mathbf{v}_k} \quad \sum_{i\in\mathcal{N}} \sum_{k=0}^{K} \mathbf{e}_k^\top \mathbf{s}^i \mathbf{x}_k^i + \mathbf{R}^{i\top} g_k^i(\mathbf{u}_k^i) + P_k^i(\mathbf{a}_k^i \mathbf{x}_k^i - \mathbf{D}_k^i)$$
$$-\pi^i(\mathbf{v}_k) + \lambda_k(\sum_{i\in\mathcal{N}} \sum_{v\in V} \mathbf{s}^i \mathbf{x}_k^i - \mathbf{v}_k))$$

which is separable. Therefore, our dual decomposition mechanism is described by Algorithm 2. At time k, the InP announces the future resource prices for a window $1 \leq t \leq W$, and each provider $i \in \mathcal{N}$ submits the demand $\mathbf{x}_{k+t|k}^{iv} \forall v \in V$. At each step, the service provider solves the problem

$$\min_{\mathbf{u}_{k+t|k}^i} \sum_{t=0}^{W} \sum_{v\in V} (\mathbf{e}_{k+t|k}^\top \mathbf{s}^i + \lambda_{k+t|k} \mathbf{s}^i) \mathbf{x}_{k+t|k}^{iv} + \mathbf{R}^{i\top} g_k^i(\mathbf{u}_{k+t|k}^i) + P_k^i(\mathbf{a}_{k+t}^i \mathbf{x}_{k+t|k}^i - \mathbf{D}_{k+t|k}^i)$$

subject to constraints (3.21) and (3.22). The cloud provider will first solve the following problem:

$$\min_{\mathbf{v}_{k+t|k}\in\mathbb{R}^V} \quad \sum_{t=0}^{W} \pi(\mathbf{v}_{k+t|k}) - \lambda_{k+t|k}(\mathbf{v}_{k+t|k}),$$

then updates the $\lambda_{k+t|k}$ according to the following equation, where $\alpha \in \mathbb{R}_+$ is the step size:

$$\lambda_{k+t|k} := (\lambda_{k+t|k} + \alpha(\sum_{i\in\mathcal{N}} \sum_{v\in V} \mathbf{s}^i \mathbf{x}_{k+t|k}^i - \mathbf{v}_{k+t|k}))_+$$

Algorithm 3 Iterative Algorithm for Achieving the best NEMPC Algorithm for DSPP

At beginning of time k, provide initial state $\mathbf{x}_0$, $k \leftarrow 0$, Initialize $\mathbf{C}^i \in \mathbb{R}_+^L$, $\bar{\mathbf{x}}_k^i \leftarrow 0$, $\forall k \in \mathcal{K}$ $\bar{J}(\mathbf{u}^1, ..., \mathbf{u}^N) \leftarrow \infty$,*converged* $\leftarrow$**false**
$\mathbf{p}_{k+t|k} \leftarrow \mathbf{s}^i \mathbf{e}_k \forall 0 \leq t \leq W-1$
while *converged* $\neq$**true do**
  for all $i = \{1, ..., N\}$ **do**
    $\mathbf{u}^i \leftarrow$ solution of DSPP^i with price $\mathbf{p}_{k+t|k}, \forall 0 \leq t \leq W-1$
  end for
  $J(\mathbf{u}^1, ..., \mathbf{u}^N) = \sum_{i \in \mathcal{N}} J^i(\mathbf{u}^1, ..., \mathbf{u}^N)$
  if $|J(\mathbf{u}^1, ..., \mathbf{u}^N) - \bar{J}(\mathbf{u}^1, ..., \mathbf{u}^N)| \leq 0.01 \times \bar{J}(\mathbf{u}^1, ..., \mathbf{u}^N)$ **then**
    $|J(\mathbf{u}^1, ..., \mathbf{u}^N) - \bar{J}(\mathbf{u}^1, ..., \mathbf{u}^N)| \leq \epsilon \times \bar{J}(\mathbf{u}^1, ..., \mathbf{u}^N)$*converged* $\leftarrow$**true**
  end if
  if *converged* $\neq$**true then**
    $\lambda_{k+t|k} := (\lambda_{k+t|k} + \alpha(\sum_{i \in \mathcal{N}} \sum_{v \in V} \mathbf{s}^i \mathbf{x}_{k+t|k}^i - \mathbf{v}_{k+t|k}))_+$
    $\mathbf{p}_{k+t|k} = \mathbf{e}_k^\top \mathbf{s}^i + \lambda_k \mathbf{s}^i$
  end if
end while

Finally, the cloud provider announces a new price $\mathbf{p}_{k+t|k} = \mathbf{e}_k^\top \mathbf{s}^i + \lambda_k \mathbf{s}^i$. This process repeats until the solution converges to a local optimal solution. Finally, even though convergence rate can be a practical concern for gradient-based algorithms, our mechanism can simultaneously adjust prices many steps into the future, which gives more time for prices to converge. The convergence rate of Algorithm 2 is analyzed in Section 3.7.

Remark 3. In this work we assume a monopolized market where the goal of the InP is to maximize social welfare rather than purely optimizing revenue. This is a reasonable model as the social welfare, in some sense, is a measure of service quality. An InP that maximizes social welfare is likely to attract more businesses in the future. Secondly, to prevent the InP from setting unfair prices, governments are likely to impose a fair return price [76]. One limitation of setting fair return price is the possibility of lower income of the InP. However, it is straightforward to consider the case where the fair return price considers the revenue gain of the InP.

In summary, the lesson we learned in the theoretical analysis is that simply rejecting requests when capacity is reached can lead to inefficient outcomes where certain service providers may be treated unfairly. A dynamic congestion-pricing mechanism can be helpful for mitigating this problem.

3.7 Simulations

We have implemented our solutions and conducted several simulation studies. In our simulations, we have used a real Internet topology graph from the *Rocketfuel project* [103], which contains link latency information. However, as the data set only contains topologies for several tier-1 Internet Service Providers (ISPs), we have augmented the topology graph by introducing regional and local ISPs, similar to the procedure for generating transit-stub networks in the GT-ITM network topology generator [13]. We specify the communication latency at intra-transit, stub-transit and intra-stub domain links to be $20ms$, $5ms$ and $2ms$, respectively [96]. Based on our experience with Google's data centers, in our experiment, we have created 3 large data centers located in Mountain View, CA, Houston, and TX, Atlanta, GA. To generate realistic service requests, we have decided to use the Worldcup 98 dataset [19] which contains HTTP requests for a total duration of 92 days. This dataset contains several occurrences of demand spikes, which is useful for demonstrating the performance of our algorithm. However, as the dataset does not contain location information of access networks, we use the request *regions* provided in the dataset to approximate the source of requests. In our simulation, we assume there is one access network responsible for generating requests from a single region. Figure 3.5 illustrates the service demand for the week between June 13 and June 20 for 4 different regions. In our experiment, the price of resources in each data center is set to the electricity price per VM according to the VM size. Figure 3.11 shows the electricity price during different times of the day. For comparison purpose, we assume the price in Atlanta is constant (i.e., not market-driven).

3.7.1 The Case of a Single Service Provider

In this subsection, we report our experiment results for the single service provider case. We first evaluate the quality of demand prediction model (ARIMA in our case), which plays an important role in determining the quality of the solution. In our evaluation, we divided the workload into two data sets. The first set is used to estimate the parameters of the ARIMA model. The second set is used to assess the accuracy of the prediction. The prediction error is measured by the Relative Squared Error (RSE) [128].When the RSE is less than 1, the prediction model is considered better than the Mean model, where the mean of the data is used as the predicted value. Typically, the smaller is the RSE, the better is the prediction. Figure 3.6 shows the prediction error as a function of the number of lags (previous samples) used as inputs for the ARIMA model. It can be seen that the prediction using 6 lags achieves the lowest RSE. Figure 3.7 shows the predicted demand compared to the actual demand for region 2, which has the highest demand fluctuation

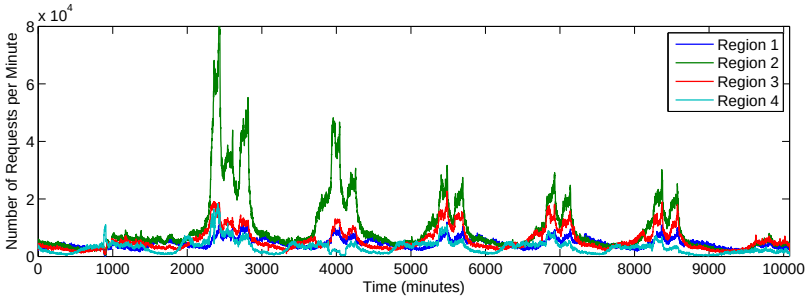


Figure 3.5: HTTP requests in the Worldcup 98 dataset

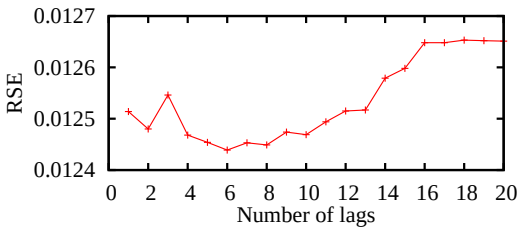


Figure 3.6: Prediction Accuracy vs. number of lags used

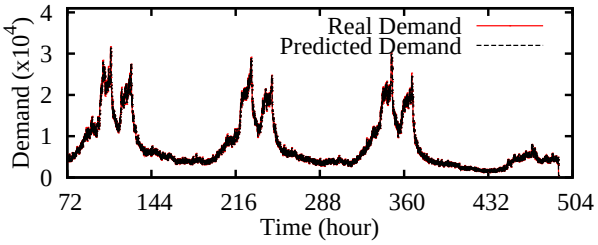


Figure 3.7: Actual vs. Predicted demand for region 2

among all the regions. It is clear that ARIMA model can achieve an accurate prediction of service demand (RSE$\approx$ 0.01245).

To demonstrate how our controller adjusts resource allocation to handle demand fluctuation, we consider the case where there is a single data center responsible for requests from two regions. Figure 3.8 shows that the controller always tries to adjust the resource allocation to match the demand. We also analyzed the effect of the prediction horizon W on the outcome of dynamic resource allocation. Figure 3.9 shows that the change in the number of servers decreases as W increases. The controller with a long window size is less aggressive than a controller that only looks few steps into the future. But at the same time, it can cause higher cost due to poor prediction of future demand (Figure 3.10). To demonstrate controller's reaction to price change, we have simulated a scenario where 2 data centers (Mountain View and Atlanta) are used to serve demand from region 2. Our experiment result is shown in Figure 3.12. Accordingly, Figure 3.12 shows that our controller allocates fewer servers in the Mountain View DC in the afternoon, since price in Atlanta is cheaper. This confirms our algorithm can balance load according to price change. Finally, we demonstrate the importance of reconfiguration cost. We have implemented a greedy algorithm that ignores reconfiguration cost. The output of the algorithm is shown in Figure 3.13. It can be seen that, since the price in Atlanta is cheaper in the afternoon the greedy algorithm performs a massive migration of servers in the afternoon. Similarly, once the price in Mountain View becomes cheaper, the greedy algorithm performs another massive migration to send them back to Mountain View. It is evident that reconfiguration cost plays a crucial role in avoiding massive migrations in this scenario. We also found the prediction window produces a similar effect as reconfiguration cost (Figure 3.9 and Figure 3.14). However, since the prediction window is used to capture trend in system inputs, and its effect is highly dependent on the prediction accuracy, it should not be used to control the aggressiveness of dynamic adaptation. Finally, we have also evaluated the running time using all 4 DC locations and 24 access networks with synthetic workloads, and found Algorithm 1 usually runs in less than 1 second.

3.7.2 The Case of Multiple Service Providers

To analyze the outcome of the resource competition game, we generate the input parameters (μ^i, $\mathbf{D}^i_k$, s^i, c^{il},$\bar{d}^i$) for each service provider $i \in \mathcal{N}$ randomly. We first simulated the standard game by allowing every service provider to move in a sequential order, until no service provider can further reduce its cost without violating the capacity constraint. We also implemented our coordination algorithm described by Algorithm 2. We set the step size $\alpha = 0.5$, and define $\pi(\mathbf{v}) = \mathbf{v}^\top \mathbf{I}\mathbf{v} \cdot P$, where $P = 10$. We plotted the average cost

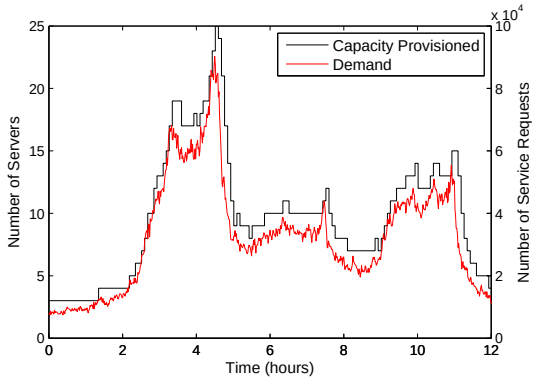


Figure 3.8: Response to Demand Fluctuation

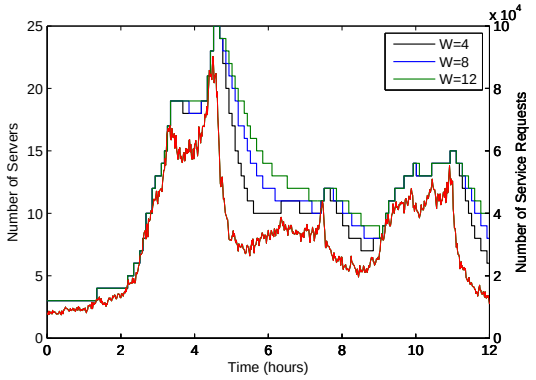


Figure 3.9: Effect of prediction window size on the number of servers

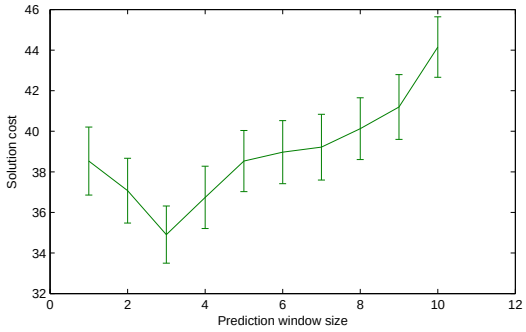


Figure 3.10: Effect of prediction horizon on the solution cost

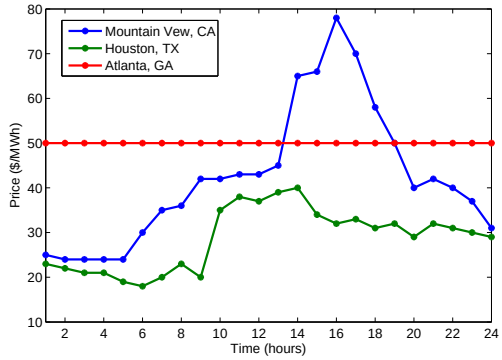


Figure 3.11: Prices of electricity used in experiments

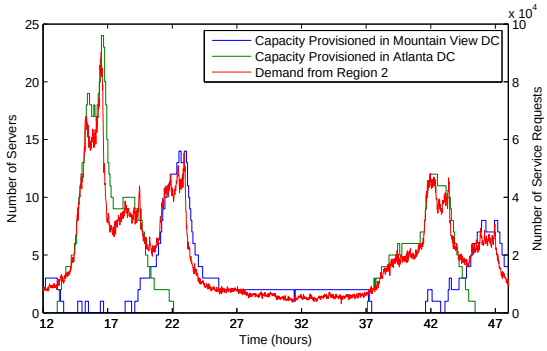


Figure 3.12: Impact of Price on Solution quality

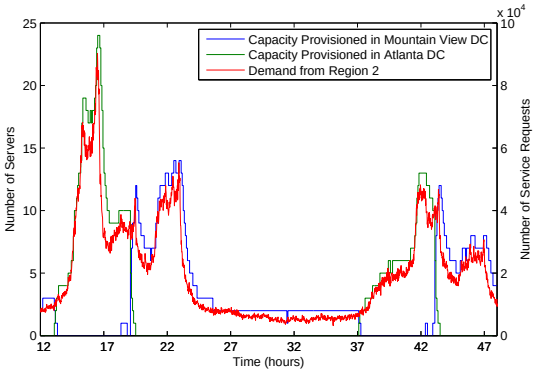


Figure 3.13: Output of the greedy algorithm

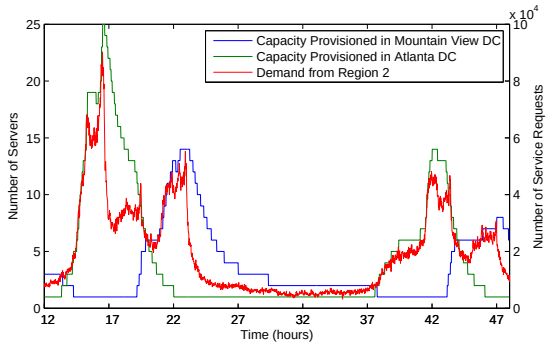


Figure 3.14: Output with no reconfiguration cost and long window size

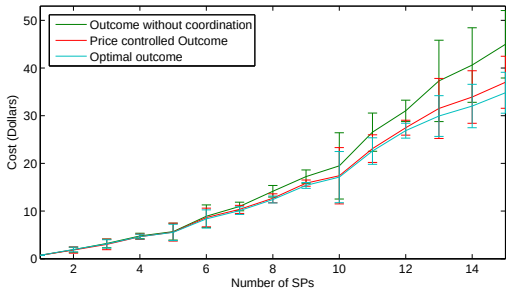


Figure 3.15: Comparing the cost of NEs versus the number of players

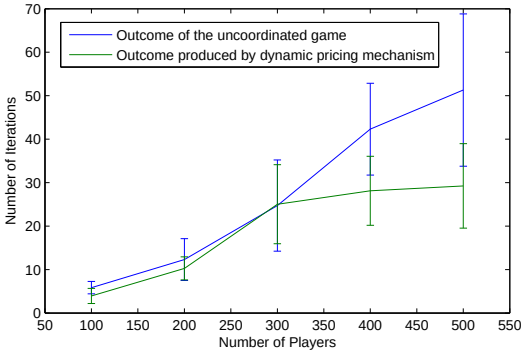


Figure 3.16: Number of players vs. convergence rate

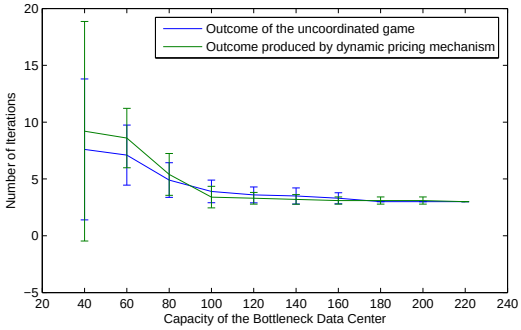


Figure 3.17: Capacity of the Bottleneck DC vs. convergence rate

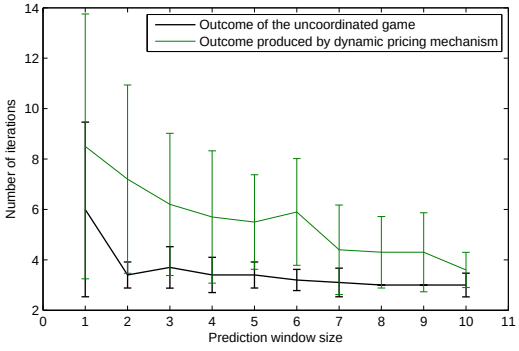


Figure 3.18: Prediction horizon length vs. convergence rate

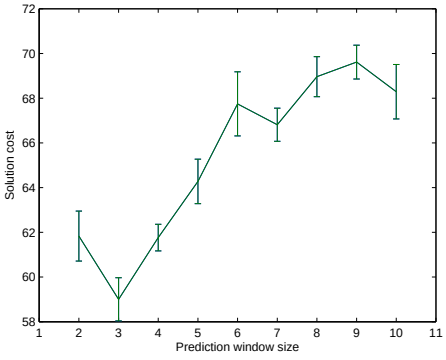


Figure 3.19: Impact of prediction horizon length on the cost

of uncoordinated NEs and the cost produced by Algorithm 2 for the duration of 24 hours in Figure 3.15. For comparison purpose, we also plotted the optimal offline solution in Figure 3.15. Clearly, Algorithm 2 improves the social welfare by $10-20\%$ compared to uncoordinated NEs.

We then analyze the scalability of the algorithm in terms of convergence rate. To produce a competition scenario, we set the number of servers in the data center with the cheapest cost (i.e., Data center in Houston, TX) to 500 respectively, and record the number of iterations required to produce an approximately stable outcome. In our experiment, we call an outcome approximately stable if $|\bar{J}(\mathbf{u}^1,...,\mathbf{u}^N)-J(\mathbf{u}^1,...,\mathbf{u}^N)| \leq 0.01\cdot\bar{J}(\mathbf{u}^1,...,\mathbf{u}^N))$, where $\bar{J}(\mathbf{u}^1,...,\mathbf{u}^N)$ is the cost of the solution in the previous iteration. Figure 3.16 shows the number of iterations to obtain a stable outcome grows with the number of players. However, as mentioned before, the convergence process can start $W-1$ steps ahead, thus the convergence rate is still acceptable. Finally, even though dynamic conditions such as machine failures, service providers joining and leaving the system can hurt the convergence rate, by correctly modeling the penalty caused by dynamic conditions (e.g. using penalty function $\pi(\cdot)$) and using short time intervals, it is possible to bring the impact of dynamic conditions to a minimum. Finally, we also conducted experiments to examine the impact of prediction horizon W on the solution optimality and convergence rate. Figure 3.18 suggests that longer prediction horizon can improve convergence rate. However, the selecting the right window size should also consider the solution quality. Similar to Figure 3.9, we found setting window size to $W=3$ archives the best outcome.

3.8 Conclusion

In this chapter, we present a framework for the dynamic service placement problem based on control- and game-theoretic models. In particular, we provided a solution that optimizes the hosting cost dynamically over time according to both demand and resource price fluctuations. We also considered the case where multiple service providers compete for revenue dynamically. Our analysis showed that in an uncoordinated scenario where service providers behave in a selfish manner, the resulting NE can be significantly worse than the optimal NE in terms of social welfare. Based on this observation, we proposed a mechanism that can be adopted by the InP to maximize the social welfare of the system. Our simulations not only confirm the theoretical findings, but also demonstrate the benefits of the proposed approach.

Chapter 4

Harmony: Dynamic Heterogeneity-Aware Resource Provisioning in Clouds

4.1 Introduction

Data centers have recently gained significant popularity as a cost-effective platform for hosting large-scale service applications. While large data centers enjoy economies of scale by amortizing long-term capital investments over a large number of machines, they also incur tremendous energy costs in terms of power distribution and cooling. For instance, it has been reported that energy-related costs account for approximately 12% percent of overall data center expenditures [18]. For large companies like Google, a 3% reduction in energy cost can translate to over a million dollars in cost savings [93]. On the other hand, governmental agencies continue to implement standards and regulations to promote energy-efficient computing [9]. As a result, reducing energy consumption has become a primary concern for today's data center operators.

In recent years, there has been extensive research on improving data center energy efficiency [99,128]. One promising technique that received significant attention is *Dynamic Capacity Provisioning* (DCP). The goal of this technique is to dynamically adjust the number of active machines in a data center in order to reduce energy consumption while meeting the *Service Level Objectives* (SLOs) of workloads. In the context of workload scheduling in data centers, a metric of particular importance is *scheduling delay* [82, 97, 100,127], which is the time a request waits in the scheduling queue before it is scheduled on a

machine. Task scheduling delay is a primary concern in data center environments for several reasons: (1) A user may need to immediately scale up an application to accommodate a surge in demand and hence requires the resource request to be satisfied as soon as possible. (2) Even for lower-priority requests (e.g., background applications), long scheduling delay can lead to starvation, which can significantly hurt the performance of these applications. In practice, however, there is often a trade-off between energy savings and scheduling delay. Even though turning off a large number of machines can achieve high energy savings, at the same time, it reduces service capacity and hence leads to high scheduling delay.

However, despite the fact that a large number of DCP schemes have been proposed in the literature in recent years, a key challenge that often has been overlooked or considered difficult to address is *heterogeneity*, which is prevalent in production cloud data centers [97]. We summarize the types of heterogeneity found in production environments as follows:

- **Machine Heterogeneity**. Production data centers often comprise several types of machines from multiple generations [100]. They have heterogeneous processing capacities and capabilities, different hardware features, processor architectures, processor speeds, memory and disk sizes. Consequently, they also have different energy consumption rates at run-time.

- **Workload Heterogeneity**. Production data centers receive a vast number of heterogeneous resource requests with diverse resource demand, durations, priorities and performance objectives [82, 100, 127]. In particular, it has been reported that the differences in resource demand and duration can span several orders of magnitude [33, 100, 127].

The heterogeneous nature of both machine and workload in production cloud environments has profound implications on the design of DCP schemes. In particular, given a rise of workload requests, a heterogeneous-oblivious DCP scheme can turn on wrong types of machines which are not capable of handling these requests (e.g., due to insufficient capacity), resulting in both resource wastage and high scheduling delays. However, designing a heterogeneity-aware DCP scheme can be difficult, because it requires an accurate characterization of both workload and machine heterogeneities. At the same time, it also requires a heterogeneity-aware performance model that balances the trade-off between energy savings and scheduling delay at run-time. Finally, the heterogeneity-aware DCP scheme should also take into account the reconfiguration costs associated with switching on and off individual machines. This is because frequently turning on and off a machine can cause the so-called "wear-and-tear" effect that can reduce the machine lifetime. Therefore, the reconfiguration cost due to server switching should be considered as well.

This chapter presents Harmony, a **H**eterogeneity-**A**ware **R**esource **MON**-itoring and management s**Y**stem that addresses the aforementioned challenges. Specifically, we first present a characterization of the heterogeneity found in one of Google's production compute clusters. Using standard K-means clustering, we show that the heterogeneous workload can be divided into multiple task classes with similar characteristics in terms of resource and performance objectives. We then formulate the DCP as an optimization problem that considers machine and workload heterogeneity as well as reconfiguration costs. We then devise online control algorithms based on the *Model Predictive Control* framework that solves the optimization problem at run-time. Through extensive experiments using traces from Google's production compute clusters, we found Harmony is able to achieve lower scheduling delay and energy consumption compared to heterogeneity-oblivious DCP solutions.

The remainder of the chapter is organized as follows. Section 4.2 surveys related work in the literature. Section 4.3 provides an analysis of a publicly available workload traces from Google to motivate our approach. Section 4.4 provides an overview of Harmony. In Section 4.5, we describe the way Harmony captures the run-time workload composition. We present the mathematical formulation of the heterogeneity-aware DCP in Section 4.6, and provide two technical solutions in Section 4.7. Section 4.8 discusses the deployment of Harmony in practice. Finally, we evaluate our proposed system using Google workload traces in Section 4.9, and draw our conclusions in Section 4.10.

4.2 Related work

In this section we provide a survey of existing work on (1) understanding workload and machine characteristics in production clouds, and (2) dynamic capacity provisioning for balancing the trade-off between energy savings and application performance objectives.

4.2.1 Machine and workload characterization

Characterizing workload in production clouds has received much attention in recent years, as both scheduler design and capacity upgrade require a careful understanding of the workload characteristics in terms of arrival rate, requirements, and duration [82]. For example, Mishra et. al. have analyzed the workload of a Google compute cluster, and proposed an approach to task classification using k-means clustering [82]. Following the same line of research, Chen et. al. provided a characterization of Google cluster workload at

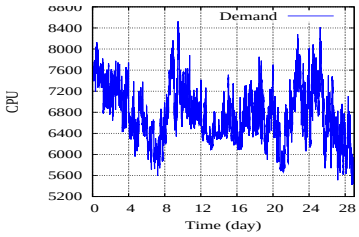


Figure 4.1: Total CPU Demand

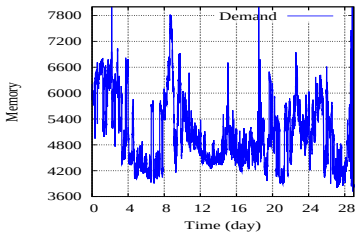


Figure 4.2: Total Memory Demand

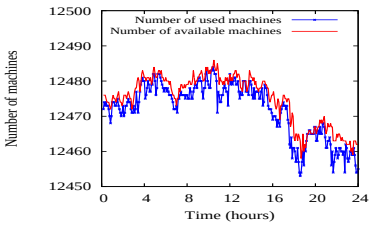


Figure 4.3: Number of Machines

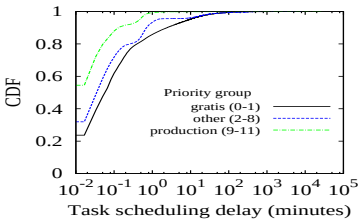


Figure 4.4: CDF of Task Scheduling delay

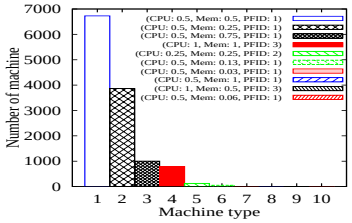


Figure 4.5: Machine Heterogeneity

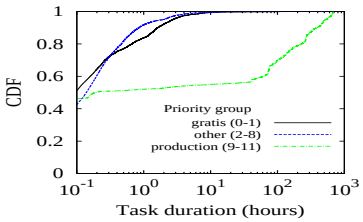


Figure 4.6: CDF of Task duration

job-level applying the k-means algorithm [41]. Sharma et. al. [100] and Zhang et. al. [127] studied the problem of finding accurate workload characterizations through benchmark generation and validation. Recently, Reiss et. al. [97] provided a comprehensive analysis of the heterogeneity and dynamism found in Google cluster traces, and found that both machine configurations and workload composition are highly heterogeneous and dynamic over time. They also pointed out the importance of considering workload heterogeneity for designing adaptive schedulers. However, the goal of these studies was to understand the workload composition in production clouds, rather than using workload characterization for resource allocation and capacity provisioning.

4.2.2 Energy-aware capacity provisioning

There is a large body of literature on energy-aware dynamic capacity provisioning in data centers. For example, pMapper [109] is a migration-aware workload placement framework for optimizing application performance and power consumption in data centers. However, it does not consider the cost of turning on and off machines. Similarly, Mistral [67] is a framework that dynamically adjusts VM placement to find a trade-off between power consumption, application performance, and reconfiguration costs. However, it does not

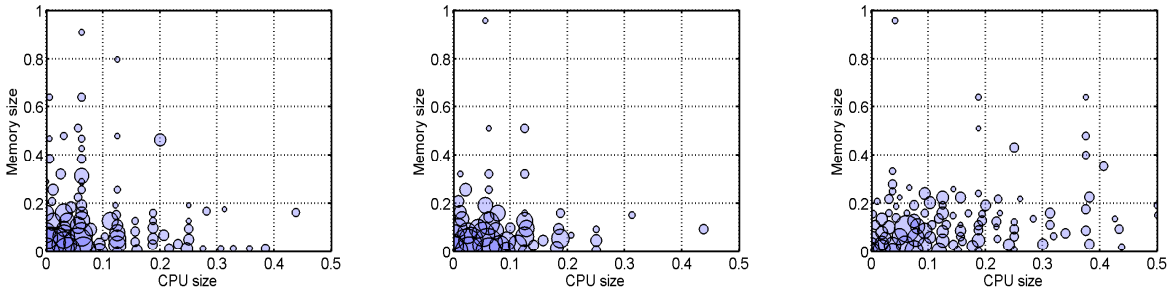
consider the arrival rate of task requests in its formulation. More recently, Ren et. al. [99] studied the problem of scheduling heterogenous batch workload across geographically distributed data centers. Different from our work, they assume workload has already been divided into distinct types. They further assume every task can be scheduled on any machine, which is not always the case as we shall demonstrate in Section 4.3. To the best of our knowledge, no previous work has applied task classification to dynamic capacity provisioning problem in heterogenous data centers. Thus, we design Harmony as a workload-aware DCP framework that can achieve both higher application performance and efficiency in terms of energy savings.

4.3 Workload analysis

To understand the heterogeneity in production cloud data centers, we have conducted an analysis of workload traces from one of Google's production compute clusters [12][1] consisting of approximately $12,000$ machines. The workload traces contain scheduling events, resource demand and usage records for a total of $672,003$ jobs and $25,462,157$ tasks over a time span of 29 days. Specifically, a *job* is an application that consists of one or more tasks. Each *task* is scheduled on a single physical machine. When a job is submitted, the user can specify the maximum allowed resource *demand* for each task in terms of required CPU and memory size. The values of the demand for each resource type were normalized between 0 and 1. Even though the dataset does not provide task size for other resource types such as disk, it is straightforward to extend our approach to consider additional resource types.

In addition to resource demand, the user can also specify a scheduling class, a priority and placement constraints for each task. The *scheduling class* captures the type of the task. Its value ranges from 0 to 3, with 0 corresponding to least latency-sensitive tasks (e.g., batch processing tasks) and 3, the most latency-sensitive tasks (e.g., web servers). The scheduling class is used by every machine to determine the local resource allocation policy that should be applied to each task. The *priority* reflects the importance of each task. There are 12 priorities that are divided into three *priority groups*: $gratis(0-1)$, $other(2-8)$, $production(9-11)$ [97]. Generally speaking, task priorities can be used for specifying the Quality of Service (QoS) in terms of desired task scheduling delay. During busy periods when demand approaches cluster capacity, task priorities can ensure that high priority tasks are scheduled earlier than low priority tasks, resulting in lower scheduling

[1]It should be mentioned that the same dataset has been analyzed by Reiss et. al. [97]. However, our analysis extends, and largely complements the results in [97].



(a) Gratis (0-1) (b) Other (c) Production

Figure 4.7: Task Size Analysis

delay. In this work, we primarily analyze task characteristics at the priority group-level, because priority groups already provide a coarse-grained classification of tasks. In addition, they also have strong correlation with task scheduling classes [12, 97]. Nevertheless, our technical approach can be extended to handle any combination of task priority groups and task scheduling classes.

4.3.1 Machine and Workload Dynamicity

In our analysis, we first plot the total demand for both CPU and memory over time. The results are shown in Figure 4.1 and 4.2, respectively. The total demand at a given time is determined by total resource requirements by all tasks in the system, including the tasks that are waiting to be scheduled. From both figures, it can be observed that the demand for each resource type can fluctuate significantly over time. Figure 4.3 shows the number of machines available and used in the cluster. Specifically, a machine is available if it can be turned on to execute tasks, and is used if there is at least one task running on it. Figure 4.3 also suggests that the capacity of the cluster is not adjusted according to resource demand, as the number of used machines is almost equal to the number of available machines. These observations suggest that a large number of machines can be turned off to save energy.

4.3.2 Analysis of Task Scheduling Delay

While turning off active machines can reduce total energy consumption, turning off too many machines can also hurt task performance in terms of scheduling delay. Figure 4.4 shows the Cumulative Distribution Function (CDF) of the scheduling delay for tasks with

respect to their priority groups. It is apparent that tasks with production priority have better scheduling delay than the gratis ones. Indeed, more than 50% and 30% of the tasks in *production* and *other* priority groups respectively are scheduled immediately. On the other hand, some of the tasks were delayed significantly. During our analysis, we have also noticed that some tasks even with production priority were delayed for up to 21 days. Since the cluster is not constantly overloaded, the only possible explanation is that the task is difficult to schedule due to unrealistic resource requirement or there is a placement constraint that is difficult to satisfy. These results suggests that more efficient provisioning and scheduling methods are needed to reduce the scheduling delay for these difficult-to-schedule tasks.

4.3.3 Understanding Machine Heterogeneity

The traces also provide information about the types of machines used in the cluster. A machine is characterized by its capacity in terms of CPU, memory and disk size as well as a platform ID, which identifies the micro-architecture (e.g., vendor name and chipset version) and memory technology (e.g., DDR or DDR2) of the machine. Similar to tasks, machine capacities are normalized such that the largest machine has a capacity equal to 1. Figure 4.5 shows the different types of machines and their characteristics (capacity and platform ID (PFID)). We found 10 types of machines where more than 50% and 30% of the machines belong to machine types 1 and 2, respectively. On the other hand, machine types 3 and 4 have around 1000 machines each. The remaining machine types (5 to 10) constitute less than 100 machines. Unfortunately, the traces do not provide detailed information about hardware specifications, however, it is clear that such a heterogeneity will translate into different energy consumption models.

4.3.4 Understanding Task Heterogeneity

In order to analyze the workload heterogeneity, we plotted tasks requirements and their durations for the three priority groups. Figure 4.7 shows the CPU and memory size of tasks belonging to each priority group. The coordinates of each point in these figures correspond to a combination of CPU and memory requirements. Radius of each circle is logarithmic in number of tasks within its proximity. It can be seen that most of the tasks have low resource requirements. In particular, we found that 43% of gratis tasks have the same CPU and memory requirements equal to 0.0125 and 0.0159, respectively. Furthermore, most of the large tasks are either CPU-intensive or memory-intensive. There

is usually no correlation between CPU requirement and memory requirements. Another key observation is that the difference in task size can span several orders of magnitude. For example, Figure 4.7a shows that the largest task in the gratis priority group is almost 1000× bigger than the smallest task in the same group for both CPU and memory. Similar characteristics can also be found in Figure 4.7b and 4.7c. Finally, by comparing Figure 4.5 and Figure 4.7, it is easy to see that not every task (e.g., CPU size ≈ 1) can be scheduled on every type of machine (e.g., CPU capacity$= 0.5$).

Another important parameter that shows the heterogeneity of the tasks is the task duration. Figure 4.6 shows the CDF of task durations for tasks within different priority groups. From Figure 4.6, it can be seen that production tasks (9-11) have long durations that can reach up to 17 days, whereas 90% of the remaining tasks (i.e., gratis and other) have shorter durations that range between 0 and 10 hours. The same observation can be made for production-priority tasks when compared to other priority groups (Figure 4.6). Furthermore, it is worth noting that more than 50% of the tasks are short (less than 100 seconds). This concurs with the previous workload analysis studies [127], which showed that tasks are either short or long.

4.3.5 Summary

The above analysis suggests that while the benefit of dynamic capacity provisioning is apparent for production data center environments, designing an effective and dynamic capacity provisioning scheme is challenging, as it involves finding a satisfactory compromise between energy savings and scheduling delay with consideration to the heterogeneous characteristics of both machines and workload. In particular, we have found the heterogeneity in task size can span several orders of magnitude, and not every type of machine can schedule every task. Similar characteristics have also been recently reported in Microsoft and Facebook data centers [33]. Thus, it is a critical issue to design heterogeneity-aware DCP schemes for production data centers, as failing to consider these heterogeneous characteristics will result in sub-optimal performance for DCP.

4.4 System Overview

As discussed previously, we design Harmony as a DCP framework that considers both task and machine heterogeneity. This requires (1) an accurate characterization of both workload and machines, (2) effectively capture the dynamic workload composition at run time,

and (3) using the captured information to control the number of machines in the compute cluster to achieve a balance between energy savings and scheduling delay. In practice, large cloud infrastructures such as Google compute clusters execute millions of tasks per day. Capturing heterogeneity at fine-grained (i.e., per-task) level is not a viable option due to the high overhead for monitoring and computation. Thus, a medium-grained characterization of the workload is necessary. To this end, we present a workload characterization of Google traces by dividing tasks into *task classes* using the K-means algorithm. However, different from previous work [41,82] whose main objective is to understand workload characteristics, our goal is to find accurate workoad characterization, while supporting task classification (e.g., labeling) at run time. It should be mentioned that machines are naturally characterized (i.e., there are 10 types of machines in the cluster). Thus, our solution will mainly focus on task characterization.

Once the workload characterization has been obtained, we introduce a monitoring mechanism that allows Harmony to capture the run-time workload composition in terms of arrival rate for each task class. To make provisioning decisions, we define a *container* as a logical allocation of resources to a task that belongs to a task class. In our approach, the task containers serve as reservations for helping the controller to make machine allocation decisions (to be described in Section 4.7.2). It is also possible to directly use task containers for scheduling (to be described in Section 4.7.3). Finally, a heterogeneity-aware DCP controller is designed to adjust the number of active machines, based on the current machine availability and workload composition.

The architecture of Harmony is shown in Figure 4.8. It consists of the following components. *The task analysis module* is responsible for monitoring the arrival of every task class in order to identify the type to which it belongs. *The scheduler* is responsible for assigning incoming tasks to active machines in the cluster. *The prediction module* receives statistics of the arrival rate for each task class, and forecasts its future arrival rates. *The container manager* evaluates the number of containers required to schedule the current workload. These parameters are evaluated based on two factors: (1) the predicted arrival rate, and (2) the required average scheduling delay for each type of tasks. The container manager periodically notifies the capacity provisioning module about the number of required containers for each type of task. *The capacity provisioning module* decides which machine in particular should be switched on or off. Obviously, the goal is to select the right combination of machines that can host the containers and, at the same time, minimizes the energy consumption. Finally, *The monitoring module* is responsible for collecting diverse statistics about tasks and machines, including CPU and memory usage, free resources and current task durations. It also reports any failures and anomalies to the management framework. In the following sections, we describe the design of Harmony in details.

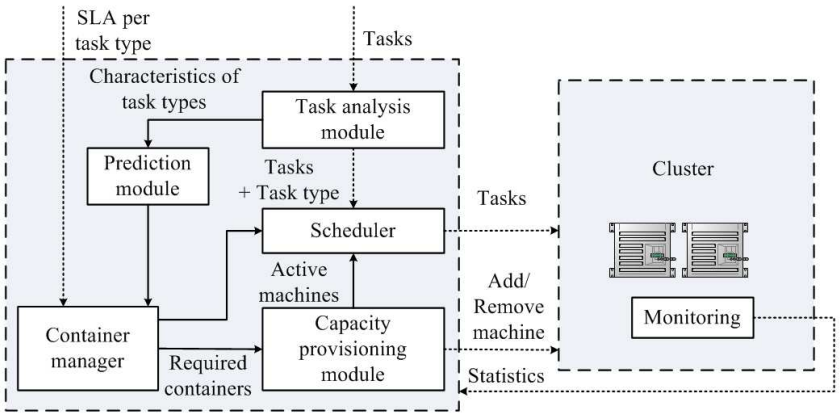


Figure 4.8: System architecture

4.5 Workload Analysis and Modeling

4.5.1 Task Classification

The goals of task classification is to divide tasks into classes with similar resource demand and performance characteristics. For the purpose of resource provisioning, it is necessary to consider task priority group, task size (CPU, memory) as well as task running time as the features for clustering. Specifically, the size of a task i can be modeled as a vector $s^i = (s^{i1}, ..., s^{iF})$, where F denotes the set of features used for clustering. Let N_k denote the tasks that belong to cluster k. Then, the centroid of each cluster can be defined as a vector $\bar{\mu}^k = (\bar{\mu}^{k1}, ..., \bar{\mu}^{kF})$, where $\bar{\mu}^{kr} = \frac{1}{|N_k|} \sum_{s^i \in N_k} s^{ir}$. The K-means clustering algorithm essentially tries to minimize the following similarity score:

$$score = \sum_{i=1}^{K} \sum_{i \in N_k} ||s^i - \bar{\mu}^k||^2 \tag{4.1}$$

where $||a - b||$ denotes the Euclidian distance between two points a and b in the feature space. Even though Harmony does not restrict the type of clustering algorithm used for clustering, in practice we found K-means is simple and sufficient to serve our purpose.

A key issue associated with the use of K-means clustering is to determine the value of K, which is the number of clusters to be produced by the algorithm. A small value of K will lead to low-quality workload characterizations, which reduces the benefit of heterogeneity-aware DCP. On the other hand, a large value of K will lead to high monitoring and

management complexity. In our scheme, we adopt a common approach which is to pick the value K such that adding another cluster does not achieve much better gain in terms of minimizing equation 4.1. We shall report the result of running the K-means clustering algorithm in Section 4.9.1.

Once a characterization of the workload has been made, the next challenge that must be addressed by Harmony is run-time task classification. This means when a task arrives, Harmony needs to determine which task class (i.e. one of the K clusters) it belongs to. An easy solution to achieve this objective is to compute the Euclidean distance between the task and each of the centroids, and assign the task to the class that has shortest Euclidean distance. However, this cannot be done directly at run-time. This is because even though the resource requirements (e.g., CPU, memory, disk size) are known, the task running time is generally unknown to the system until the task finishes. In Harmony, this issue is addressed by realizing the fact that tasks are either short or long, and the majority of the tasks are short tasks. Thus, we can initially label all tasks as short tasks, and gradually update the labels to the correct ones as time passes. Since only a small fraction of tasks are long, the error caused by the incorrect labeling is both small and short-lived.

We now describe our task clustering and run-time task labeling procedure in details. Specifically, we adopt a two-step approach for workload clustering. In the first step, tasks are classified based on static characteristics (e.g., priority, CPU and memory size specified in the job request) using the K-means algorithm. In the second step, each task class is further divided into sub-classes based on task running time. At run-time, each task is initially assumed to be short. The initial task label is determined by computing the Euclidean distance between the task feature vector and the centroids produced by the K-means algorithm. Later on if the task running time exceeds the partitioning threshold between short and long tasks, the task will be relabelled and assigned to the right task class. The advantage of this clustering procedure is that it not only simplifies the relabeling process, but also reduces the error introduced by the relabeling process.

4.5.2 Demand Prediction

Once the incoming tasks can be classified, the prediction module is responsible for monitoring forecasting the arrival rate of each task class. Currently, we have implemented a time series-based predictor using the ARIMA [36] model, which has been shown to be effective for predicting workload arrival rate [128]. However, Harmony can adopt other demand prediction models as well.

Once the predicted task arrival rates have been obtained, the next step is to deter-

mine the combination of machines that need to be provisioned in next control period. In Harmony, the container manager is responsible for computing the number of containers required to support the workload of each task class. Specifically, let c_i denote the number of containers for tasks type i such that the average scheduling delay is equal to $\bar{d}_i$. We can model the queue of tasks of type i and its corresponding N_t^i containers at time t by $M/G/N_t^i$ queue since a single container can process one task at a time. Based on queuing theory, the average waiting time d_i for type i tasks is given by [57]:

$$d_i \approx \frac{\pi_{N_t^i}}{1-\rho_i} \cdot \frac{1+CV_i^2}{2} \cdot \frac{1}{N_t^i \mu_i} \tag{4.2}$$

where μ_i is the execution rate of task type i, $\rho_i = \frac{\lambda_i}{N_t^i \mu_i}$ is the traffic intensity of tasks type i, CV_i^2 is the squared coefficient of variation of the average duration, and $\pi_{N_t^i}$ is the probability that a task has to wait in the queue. It is expressed as:

$$\pi_{N_t^i} = \frac{(N_t^i \rho)^{N_t^i}}{N_t^i!(1-\rho_i)} \left[\sum_{k=0}^{N_t^i - 1} \frac{(N_t^i \rho_i)^k}{k!} + \frac{(N_t^i \rho_i)^{N_t^i}}{N_t^i!(1-\rho_i)} \right]^{-1} \tag{4.3}$$

Given an average scheduling delay and using Eq. (4.2), it is easy to estimate N_t^i to ensure $d_i \leq \bar{d}_i$ and $\rho_i < 1$.

In our experiments, we have found this queuing model generally works well for estimating task resource requirements except for long-running tasks, for which queuing theory makes inaccurate resource predictions. We found a simple solution to deal with this limitation is to estimate the number of long running tasks using the ARIMA model. As each task runs for very long time, the number of required containers is practically the number of long running tasks. As a result, we use the ARIMA model to predict the number of long running tasks, which translates into the number of required containers.

4.6 The Capacity Provisioning Problem

We now provide a formal model for DCP in heterogenous environments. In our model, time is divided into intervals of equal duration, and control decision is made at the beginning of each time interval. The cluster consists of M types of machines. Let N_t^m denote the set of type m machines available (either active or not) at time interval t. Denote by $C^{mr} \in \mathbb{R}^+$

Table 4.1: Table of Notations

Symbol	Meaning
d_i	Average scheduling delay for task class i
R	Resource types
s^{kr}	Size of task i for resource type r
c^{kr}	Size of a container of type i for resource type r
C^{mr}	Capacity of a type m machine resource type r
$E^{idle,m}$	Energy consumption of a type m machine when idle
$\alpha^{\bar{m}r}$	Energy efficiency ratio of a type m machine for type r
u_t^{ir}	Util. of machine i for resource type r at time t
y_t^i	Boolean var. indicating machine i is active at time t
υ_t^i	Change in machine i's state at time t
a_t^{ik}	Num. of type k containers in machine i at time t
γ_t^{ik}	Change in a_t^{ik} at time t
z_t^{ik}	Number of type m machines active at time t
δ_t^m	Change in the num. of type m machines at time t
x_t^{mk}	Num. of type n containers in machine m at time t
σ_t^{mk}	Change in x_t^{mk} at t

the capacity of a single machine of type $m \in M$ for resource type $r \in R$. Similarly, there are K types of containers to be scheduled at time t, the number of containers of type k is N_t^k. Let $c^{kr} \in \mathbb{R}^+$ denote the size of a type k container for resource type $r \in R$.

Let $y_t^i \in \{0, 1\}$ denote whether machine i is active at time t. Furthermore, define $\upsilon_t^i \in \{-1, 0, 1\}$ as an integer variable that indicates whether the machine is turned on ($u_t^i = 1$) or off ($u_t^i = -1$), or unchanged ($u_t^i = 0$). We also define $a_t^{ik} \in \mathbb{N} \cup \{0\}$ as an integer variable that indicates the number of type n containers on machine i at time t, and γ_t^{ik} as the change in z_t^{ik} at time t. We thus have the following state equations:

$$y_{t+1}^i = y_t^i + \upsilon_t^i \tag{4.4}$$

$$a_{t+1}^{ik} = a_t^{ik} + \gamma_t^{ik} \tag{4.5}$$

The utilization of type r resource on machine i at time t can be computed as:

$$u_t^{ir} = \frac{1}{C^{mr}} \sum_{n \in N} z_t^{ik} c^{kr}. \tag{4.6}$$

As total energy usage of a physical machine can be estimated by a linear function of resource utilization [128], the energy consumption of all the active machines at time t can

be computed as:

$$E_t = p_t \sum_{m \in M} \sum_{i \in N_t^m} y_t^i \left(E^{idle,m} + \sum_{r \in R} \alpha^{mr} u_t^{ir} \right) \tag{4.7}$$

where $E^{idle,m} \in \mathbb{R}^+$ is the energy consumption of a type m machine when it is idle, and $\alpha^{mr} \in \mathbb{R}^+$ is the slope of the energy consumption function. We can define $E_t^{idle} = p_t \sum_{m \in M} \sum_{i \in N_t^m} y_t^i E^{idle,m}$, $E_t^{util} = p_t \sum_{m \in M} \sum_{i \in N_t^m} \sum_{r \in R} \alpha^{mr} u_t^{ir}$ and rewrite E_t as $E_t = E_t^{idle} + E_t^{util}$.

To model task scheduling delay, since it is not possible for all containers to be scheduled when demand exceeds data center capacity, we assume there is a utility function $f^k(\cdot)$ that models the monetary gain for scheduling containers. $f^k(\cdot)$ is assumed to be a concave function that can be derived from SLO objectives. For example, $f^k(a^k)$ can model the gain in monetary cost when a^k containers are scheduled for task class k. The total revenue can now be written as:

$$U_t^{perf} = \sum_{k \in K} f^k \Big(\sum_{m \in M} \sum_{i \in N_t^m} a_t^{ik} \Big) \tag{4.8}$$

.

The machine switching cost can be described by:

$$C_t^{sw}(v_t^i) = \sum_{m \in M} \sum_{i \in N_t^m} q^{on,m} (v_t^i)^+ + q^{off,m} (v_t^i)^- \tag{4.9}$$

where $q^{on,m} \in \mathbb{R}^+$ and $q^{off,m} \in \mathbb{R}^+$ denotes the cost for turning on and off of a single type m machine, respectively. Finally, equation (4.10) ensures that containers scheduled on the same machine do not exceed the resource capacity of the machine.

$$\sum_{k \in K} a_t^{ik} c^{kr} \leq y_t^i C^{mr} \quad \forall m \in M, i \in N_t^m, t \in \mathcal{T} \tag{4.10}$$

Thus, the overall objective of DCP is to control the number of active machines and to adjust container placement in a way that maximizes the total performance gain in terms of scheduling delay, while minimizes the energy consumption and machine switching cost over a time horizon $\mathcal{T} = \{1, 2, ..., T\}$:

$$\max_{a_t^{ik}, v_t^i, y_t^i, \gamma_t^{ik}} R_T = \sum_{t=1}^{T} U_t^{pref} - E_t - C_t^{sw} \tag{DCP}$$

subjects to constraints (4.4), (4.5), and (4.10).

DCP is $\mathcal{NP}$-hard to solve as it generalizes the vector bin-packing problem [39]. Furthermore, linear programming based solutions cannot be applied to DCP due to the large number of variables involved. For example, given 10 task classes and over $10K$ machines, DCP contains at least $100K$ variables, making it difficult to solve in online settings. Finally, traditional bin-packing heuristics (e.g., First-Fit) do not apply directly to DCP as they do not consider machine switching costs.

4.7 Solution Techniques

Realizing that directly solving DCP is not a viable option, in this section we present two fast heuristics for solving DCP. Both techniques rely on solving the integer-relaxation of DCP (i.e., relaxing the constraints that variables must take integer values) called $DCP - RELAX$, which is much easier to solve than DCP. Once the solution for $DCP - RELAX$ is obtained, one of our solution techniques called Container-Based Provisioning (CBP) directly rounds the numbers of machines to the nearest integer values and use these values for capacity provisioning. On the other hand, another solution technique called the Container-Based Scheduling (CBS) attempts to find a feasible placement of containers in physical machines, and use containers for run-time scheduling. In this section, we shall first present the formulation $DCP - RELAX$, followed by describing CBP and CBS in details. The benefits and limitations of each approach will be discussed in Section 4.8.

4.7.1 The Relaxation of DCP

In $DCP - RELAX$, we relax the integer constraints so that the number of machines (i.e., y_{it}) and container assignment (i.e., z_t^{ik}) no longer take integer values. This relaxation yields a simpler formulation, as we only need to solve the total number of containers for each type of machines, rather than solving the number of containers per machine. Specifically, we denote by $z_t^m \in \mathbb{R}^+$ the number of type m machines that are active at time t, and $\delta_t^m \in \mathbb{R}$ the change in the number of active machines at time t. Similarly, define $x_t^{mk} \in \mathbb{R}^+$ as the number of type k containers assigned to machines of type m that is capable of hosting containers of type k, and $\sigma_t^{mk} \in \mathbb{R}^+$ the change in x_t^{mk} at time t. We thus have the following state equations:

$$z_{t+1}^m = z_t^m + \delta_t^m \tag{4.11}$$

$$x_{t+1}^{mk} = x_t^{mk} + \sigma_t^{mn} \tag{4.12}$$

Furthermore, as x_t^{mk} can take fractional values, we need to ensure that each type of containers can only be assigned to machines that are capable of hosting them. This is achieved by introducing a predefined boolean variable ψ^{mk} that indicates whether a container of type n can be scheduled on a machine of type m. We thus have the following schedulability constraint:

$$c_{kr} x_t^{mn} \leq z_t^m \psi^{mk} C^{mr} \quad \forall m \in M, k \in K, r \in R, t \in \mathcal{T} \tag{4.13}$$

$DCP - RELAX$ can now be stated as:

$$\max_{\delta_t^m, \sigma_t^{mk}} \sum_{t=0}^{T} \sum_{m \in M} -p_t \left(z_t^m E^{idle,m} + \sum_{r \in R} \sum_{k \in K} \frac{\alpha^{mr} c^{kr}}{C^{mr}} \cdot x_t^{mk} \right)$$

$$+ \sum_{t=0}^{T} \sum_{k \in K} f^k \left(\sum_{m \in M} x_t^{mk} \right) - \sum_{m \in M} C_t^{sw}(\delta_t^m) \tag{DCP-RELAX}$$

$$\begin{aligned} \text{subject to } z_t^m &\leq N_t^m \quad && \forall m \in M, t \in \mathcal{T} \\ x_t^{mk}, z_t^m &\in \mathbb{R}^+ && \forall k \in K, m \in M, t \in \mathcal{T} \end{aligned} \tag{4.14}$$

along with constraints (4.11), (4.12) and (4.13). This problem is a convex optimization problem that can be solved using standard methods [37].

4.7.2 Container-Based Provisioning (CBP) for DCP

Container-Based Provisioning (CBP) is a simple heuristic for solving DCP. After $DCP - RELAX$ is solved, CBP simply rounds up the fractional values of (x_t^{mk}, z_t^m) to obtain an integer solution for DCP, which gives the number of machines to be provisioned (i.e., $\lceil z_t^m \rceil$) and the number of type n tasks that should be scheduled on type m machines (i.e., $\lceil x_t^{mk} \rceil$). However, at run time, the scheduler needs to ensure that the number of type n tasks assigned to type m machines must respect the provisioned capacity $\lceil x_t^{mk} \rceil$. A simple strategy is to ensure the number of type k tasks assigned to m (denoted by $Assign_t^{mk}$) is proportional to the number of containers:

$$Assign_t^{mk} = \frac{x_t^{mk}}{\sum_{j \in M} x_t^{jk}}$$

This can be achieved easily by using a weighted round-robin scheduling policy. Furthermore, it can be easily integrated with existing scheduling algorithms. For example, variants of first-fit and best-fit algorithms (which are used in production clouds such as Microsoft [75], Google [100] and Open source platforms such as Eucalyptus [11]) can adopt this mechanism by changing the scheduling policy to weight round-robin first fit and weight round-robin best fit, respectively.

A key drawback of the above rounding scheme is that it often under-estimates the required capacity. The reason is that the fractional solution of $DCP - RELAX$ assumes that each container can be arbitrarily divided and placed on multiple machines. However, in practice, this is not realizable because each container must be scheduled on a single machine. To account for the fact that $DCP - RELAX$ under-estimates the required machine capacities, we define an over-provisioning factor $\omega^k \in \mathbb{R}^+$ for each container type k. ω^k essentially captures how much extra resource is required to fully pack a given set of type n containers. To account for ω^k, it suffices to replace constraint 4.13 by the following constraint:

$$\sum_{k \in K} \omega^k c^{kr} x_t^{mk} \leq z_t^m C^{mr} \qquad \forall m \in M, r \in R, t \in \mathcal{T} \tag{4.15}$$

The value of ω^n can be obtained through experiments. For example, we have found setting $\omega^k = 1.2$ for all $n \in N$ to be a reasonable value in practice.

The main benefit of CBP is its simplicity and practicality for deployment in existing systems. However, the main drawback of CBP is that it still relies on bin-packing algorithms for scheduling. At run-time, tasks of different classes can still compete for resources in each type of machine. As a result, CBP does not provide high performance guarantee in terms of task scheduling delay.

4.7.3 Container-Based Scheduling (CBS) for DCP

In this section, we present an alternative solution to CBP called *Container-Based Scheduling* (CBS). Unlike CBP that uses bin-packing algorithms for scheduling, CBS allocates containers in each physical machine and use them for run-time task scheduling. Specifically, a type k container represents a resource reservation for tasks of type k. The number of type k containers on a machine i indicates the number of type k tasks that can be scheduled on machine i. At run-time, CBS adopts the following simple scheduling policy: each task is scheduled in the first available container such that scheduling the task on the

machine does not cause machine capacity violation. If none of the machines can schedule the task without violating machine capacity constraint, the task will be kept in the scheduling queue.

The main benefit of CBS is that it provides low scheduling delay due to resource reservations on each physical machine. However, it also introduces several challenges which we shall discuss in the following subsections.

Modeling Container Size

One of the main challenges for container-based scheduling is to select appropriate container size. Unlike in CBP where we can simply use the centroid to determine the container size, in CBS we need to set the container size large enough to ensure that with high probability, each task can be scheduled without exceeding the capacity of the physical machine. Specifically, setting the container size equal to the maximum possible container size can cause resource wastage due to over-estimation of true task resource demand. On the other hand, setting the container size equal to the average task size will lead to under-estimation of task resource consumption, resulting in tasks unschedulable in machines with available containers.

To address this issue, we rely on the statistical multiplexing of task resource demand to ensure the probability of machine capacity violation to be low. Specifically, the result of the K-means clustering algorithm divides the feature space into K partitions, where every point in the space belongs to exactly one partition (i.e. the partition whose centroid has shortest distance to the point). We assume the tasks in each partition $1 \leq k \leq K$ are independently distributed according to a common distribution $\mathcal{D}^k$ (which can be an arbitrary distribution) with mean $\mu^k = (\mu^{k1}, ..., \mu^{kR})$. and standard deviation $\sigma^k = (\sigma^{k1}, ..., \sigma^{kR})$. Our goal is to select the container size $c^k = (c^{k1}, ..., c^{iR})$ for each task class $1 \leq k \leq K$ to ensure that given a task j of type k to be scheduled, the probability a task cannot be scheduled on any of the machines that have available type k containers is less than a small value ϵ . Mathematically, given M^k machines with available type k containers, let N^n denote the tasks scheduled on each machine $n \in M^k$. Given a task i to be scheduled, we want to ensure that

$$\prod_{m \in M^k} \Pr(\exists r : \sum_{j \in N} s^{jr} + s^{ir} > C^{mr} | \sum_{j \in N} c^{jr} + c^{ir} \leq C^{mr}) \leq \epsilon \tag{4.16}$$

Theorem 4. *Assume each task s^{kr} in each class k is independently and identically distributed with mean μ^{kr} and standard deviation σ^{kr} for each resource type r. Also, let M^k*

denote the minimum number of machines on which a type k task can be scheduled. We can set container size of task type k to

$$c^{kr} = \mu^{kr} + \sqrt{\frac{|R| - \epsilon^{\frac{1}{M^k}}}{\epsilon^{\frac{1}{M^k}}}}\sigma^{kr}$$

for each $r \in R$ to ensure equation (4.16) holds.

Proof. The proof is provided in the Appendix. □

Theorem 4 provides a bound on selecting container size for CBS. For instance, if we want to achieve $\epsilon = 0.01$ for $M^k = 100$, $|R| = 2$, then Theorem 4 states that we can set container size of task type k to $\mu^{kr} + 1.1\sigma kr$, which is typically much smaller than the maximum possible size for task type k. In practice, we can use the sample mean and standard deviation to approximate the values of μ^{kr} and σ^{kr} for each $1 \leq k \leq K$. This is reasonable because there is usually a large number of samples per task class. Assume each sample is drawn independently, the sample mean and sample standard deviation will be close to the true mean and the true standard deviation. Finally, if a task still cannot be scheduled immediately, Harmony will keep the task in the front of scheduling queue until it finds a machine with sufficient resources to schedule the task. This simple policy can achieve low scheduling delay as we shall demonstrate in Section 4.9.2.

Solution Algorithm

In order to leverage containers for task scheduling, we present an alternative way to round the fractional solution of $DCP - RELAX$. The idea is to leverage the following property of the first-fit (FF) algorithm:

Lemma 5. *Given a fractional solution of $DCP - RELAX$ with z_t^{m*} type m machines and $x_t^{mn^*}$ type n containers, the first-fit (FF) algorithm can place at least $\left\lfloor \frac{x_t^{mk}}{2|R|} \right\rfloor$ of each type of container n in $z_t^{m*} + 1$ machines.*

Proof. The proof is provided in the Appendix. □

Lemma 5 essentially states that, given a fractional solution of $DCP - RELAX$ that uses z_t^{m*} type m machines and $x_t^{mk^*}$ type n containers, FF can ensure that at least $\left\lfloor \frac{x_t^{mk}}{2|R|} \right\rfloor$ containers can be placed in $z_t^{m*} + 1$ machines. Using this result, we devise our CBS

Algorithm 4 Controller Algorithm for CBS

Provide initial state z_0^m, x_0^{mk}, $t \leftarrow 0$
loop
At beginning of control period t:
  Predict $N_{t+i|t}^k$, $p_{t+i|t}$ for horizons $t = 1, \cdots, W$ using a demand prediction model
  Solve $DCP - RELAX$ to obtain $\delta_{t+i|t}^m$,$\sigma_{t+i|t}^{mk}$ for $i = 0, \cdots, W-1$
  Sort new containers based on their utilities
  for $m \in M$ **do**
    Select $z_{t|t}^m$ machines of type m as active machines
  end for
  Compute a re-packing configuration for all selected active machines
  Turn on selected machines, perform re-packing using FF, turn off other machines
  $t \leftarrow t + 1$
end loop

algorithm (Algorithm 4) as follows: When the control interval t starts, the controller uses the predicted values $N_{t+i|t}^k \forall k \in K, 1 \leq i \leq W$[2] to solve $DCP - RELAX$, which gives $z_{t|t}^{m*}$, the number of active type m machines to be made available at time t. Then the controller computes an integer solution by first reducing the number of type n containers to at most $\left\lfloor \frac{x_t^{mk}}{2|R|} \right\rfloor$ and then adding containers using FF to ensure the number of type k containers is at least $\left\lfloor \frac{x_t^{mk*}}{2|R|} \right\rfloor$ for all $1 \leq k \leq K$. Container reassignment (i.e., migration) is then performed to ensure there are at most $z_{t|t}^{m*} + 1$ machines to be active. In our formulation, container reassignment cost is modeled as part of the machine switching cost, as it is only used to allow machines to be turned off. The average switching cost can be obtained through experiments. Once the container reassignment is completed and there is still room for more containers, the controller is free to schedule additional containers as long as the total number of type k containers is at most x_t^{mk}. Finally, the controller will realize the new configuration by actually turning off unused machines and making container allocations.

Theorem 6. *The integer solution produced by Algorithm 4 ensures* $U_t^{pref} - E_t - C_t^{sw} \geq (\frac{1}{2|R|} - \epsilon)U_t^{pref*} - (1+\epsilon)(E_t^* - C_t^{sw*})$ *when* z_t^m *is sufficiently large for all* $m \in M$, $t \in T$.

Proof. The proof is provided in the Appendix. □

Theorem 6 provides a bound on the worst case performance of CBS. In experiments, we have observed Algorithm 4 typically performs much better than the worst case bound.

[2]We use $(t+i|t)$ to denote future value for time $t+i$ either predicted or computed at time t

Furthermore, realizing the bin-packing solutions often cannot fully utilize the machine capacities, similar to CBP, we can use a provisioning factor $\omega^k \in \mathbb{R}^+$ to account for the bin-packing inefficiencies. To account for ω^k, it suffices to replace constraint (4.13) by constraint (4.15). and run Algorithm 1 to find a suitable container placement. However, using ω^k does not lead to a better performance guarantee. To see this, consider an example where N_t^m of type m machines are selected by $DCP - RELAX$ to be active. All other machines are inactive and have $E^{idle} \approx \infty$. In this case, no matter how we adjust the value of ω^k, the number of containers scheduled by the algorithm will not improve.

4.8 Discussion

In this section we discuss considerations related to the deployment of Harmony in practice.

4.8.1 Task Classification and Prediction

It should be mentioned that many public cloud providers today (e.g., Amazon EC2 [6]) already offer VMs in distinct types. In such a case, our DCP algorithms can be applied directly to these public clouds. However, we argue that predefined VM sizes may not match the actual need of each customer in all cases. This is reflected by the fact that workload heterogeneity is prevalent in private clouds such as Google's compute clusters, where customers are given the flexibility to choose desired VM size. In these cases, our approach is more flexible and can provide highly efficient DCP solution for arbitrary workload compositions.

Another important issue concerns the accuracy of the demand prediction. our previous work [128] suggests that the ARIMA can forecast future demand with high accuracy when the trend of resource demand is stable. However, it is still insufficient when an unexpected demand spike occurs. In this case, we can minimize the risk of under-provisioning using the over-provisioning factor ω^k. Even though exact value of over-provisioning factor can be set based on experience, in Section 4.9.2 we shall evaluate the impact of the over-provisioning factor on the performance of CBS and CBP using the Google Traces.

4.8.2 Comparing CBS and CBP

Although CBS provides a theoretically-sound solution for DCP, it requires the scheduler to adopt a container-based scheduling algorithm, which is not always available in practice.

As many production cloud systems (e.g., Google's compute cluster) have also developed sophisticated schedulering algorithms, implementing CBS requires major change to the design of the scheduler. On the other hand, CBP does not suffer from this limitation. However, due to lack of control of the scheduler, we have found CBP often produce worse task scheduling delay compared to CBS in our experiments. In the next section we will present our evaluation of both methods and quantitatively analyze the benefits and limitations of both designs.

4.9 Simulation Studies

We simulate a heterogeneous cluster composed of a mixture of servers from multiple manufacturers and models. The 4 types of servers correspond to the 4 most popular types of machines (type 1-4) found in the Google cluster traces. Table 4.10 provides the characteristics of the simulated servers. We normalized the CPU core count and memory capacity to the largest machine size. Hence, HP DLG585 G7 has a capacity 1 CPU unit and 1 memory unit, which corresponds to 48 cores and 64GB, respectively. To demonstrate the effectiveness of Harmony, we also adjusted he number of machines according to Table 4.10.

In our experiments, the energy consumption of the different machines is modeled according to equation 4.7. The parameters $E^{idle,m}$ and α^{mr} for each type of servers were estimated using energy measurements available in [9]. Figure 4.9 shows the energy consumption as function of CPU usage. Indeed, this figure demonstrates the importance of considering the machine heterogeneity when scheduling tasks in order to reduce energy consumption. For instance, a container requiring 0.2 CPU unit should be placed in a HP DL385 G7 since the PowerEdge R210 does not have enough CPU capacity, whereas the other types of servers are able to host it but will consume much more energy. Selecting the "right" machines to switch on becomes even more challenging when millions of heterogeneous tasks have to be scheduled in the cluster.

4.9.1 Results for Task Classification

We performed task classification as described in Section 4.5.1. For each priority group, we varied the value of k and evaluated the quality of the resulting clusters produced by the K-means algorithm. The best value of k for each priority group is selected as the one for which no significant benefit can be achieved by increasing the value of k. The results

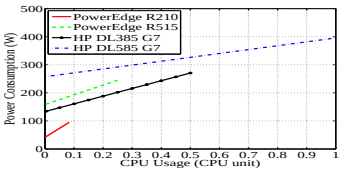


Figure 4.9: Energy Efficency

Model	Num. of Cores	Memory Memory	Num. of Machines
Dell PowerEdge R210	4	4 GB	7000
Dell PowerEdge R515	6	32 GB	1500
HP DL385 G7	12	16 GB	1000
HP DL585 G7	12	64 GB	500

Figure 4.10: Machine Configurations

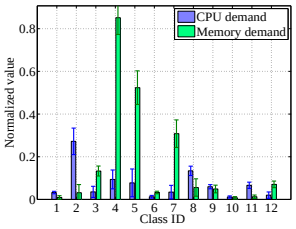


Figure 4.11: Class size (Gratis)

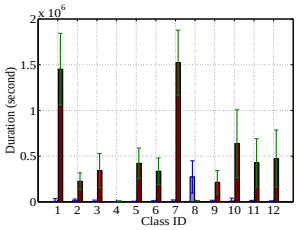


Figure 4.12: Task duration (Gratis)

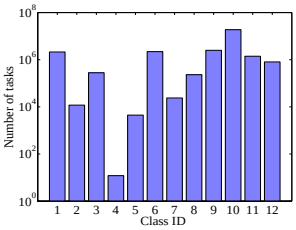


Figure 4.13: Tasks Count (Gratis)

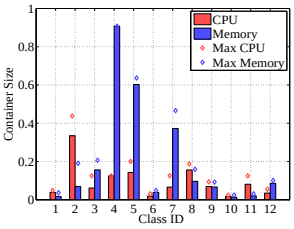


Figure 4.14: Container size (Gratis)

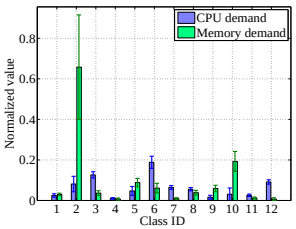


Figure 4.15: Class size (Other)

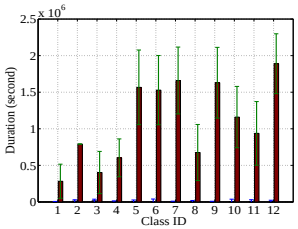


Figure 4.16: Task duration (Other)

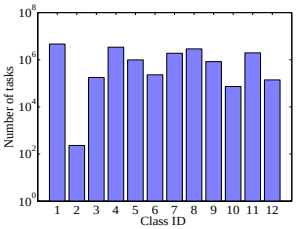


Figure 4.17: Tasks Count (Other)

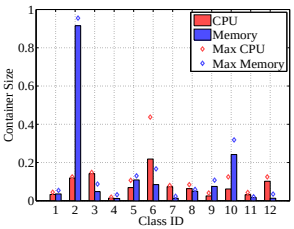


Figure 4.18: Container size (Other)

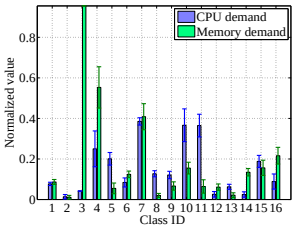


Figure 4.19: Class size (Production)

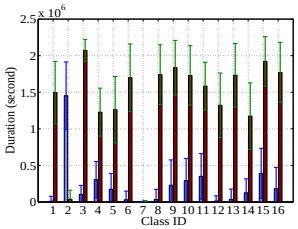


Figure 4.20: Task duration (Production)

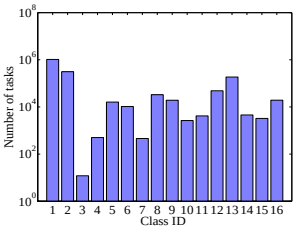


Figure 4.21: Tasks Count (Production)

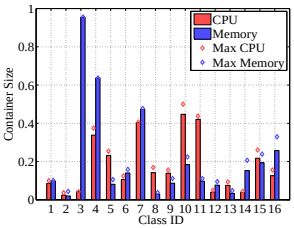


Figure 4.22: Container size (Prod.)

after the first step of our characterization for each priority group are shown in Figure 4.11, 4.15, and 4.19, respectively. These diagrams show the clustering algorithm captures the differences in task sizes and identifies cpu-intensive tasks and memory-intensive tasks. Furthermore, the standard deviation is much less than the mean value for both CPU and memory, which confirms the accuracy of the characterization. The number of tasks in each task class is shown in Figure 4.13, 4.17 and 4.21, respectively. It is clear that the number of tasks within each cluster can vary significantly. Most of the classes have between 10^4 and 10^6 tasks except cluster 4 for Gratis priority group, which has only 100 tasks. Lastly, we run the k-means algorithm with $k = 2$ to categorize tasks of each task class as either short or long. The results are shown in Figure 4.12, 4.16 and 4.20, respectively. These diagrams confirm that long tasks typically run several orders of magnitude longer than short tasks. Finally, we also computed the container size as described in Section 4.7.3, with $\epsilon = 0.001$, $|R| = 2$ and $M^k = 100$. The results are shown in Figure 4.14, 4.18 and 4.22, respectively. Clearly, the the container size is typically smaller than the maximum task size within the cluster for a majority of the clusters.

4.9.2 Controller Performance

We have evaluated the performance of CBS and CBP algorithms using Google workload traces. In our experiments, the sum of arrival rate of tasks belonging to each priority group is shown in Figure 4.23. Figure 4.24 shows the sum of the total number of containers belonging to each priority group computed by Harmony.

For comparison purpose, we have also implemented a baseline (heterogeneity-oblivious) algorithm that tries to find a balance between energy savings and scheduling by maintaining an 80% utilization of the bottleneck resource. Essentially, given the total resource demand, the baseline algorithm provisions machines in a "greedy" fashion by turning them on in decreasing order of energy efficiency (e.g., always turning on HP-DL585-G7 machines first). We picked the value of 80% because we have observed that a utilization higher than 80% can cause a significant increase in task scheduling delay. As the Google workload contains many long running tasks that were scheduled before the start of the traces, in our current simulation, we mainly focus on simulating the arrival of new tasks.

In our first experiment, we use an over-provisioning factor of 1.2 to demonstrate the behavior of our algorithms. The number of active servers provisioned by the baseline algorithm, CBS and CBP are shown in Figure 4.25, Figure 4.26 respectively. Note that both CBS and CBP provision the same number of machines as indicated by the MPC algorithm. It can be seen that the number of machines provisioned by CBS and CBP is

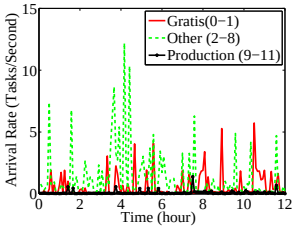


Figure 4.23: Aggregated Task Arrival Rates

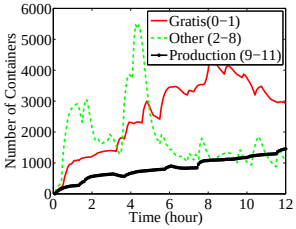


Figure 4.24: Number of required containers

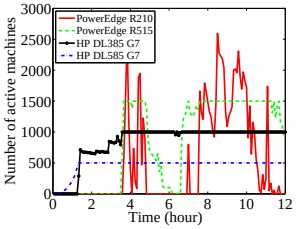


Figure 4.25: Num. of machines used by the baseline

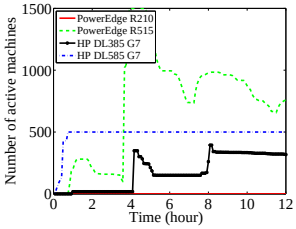


Figure 4.26: Number of machines used by CBS/CBP

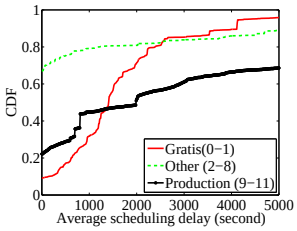


Figure 4.27: CDF of scheduling delay for baseline

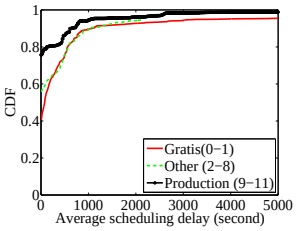


Figure 4.28: CDF of scheduling delay for CBP

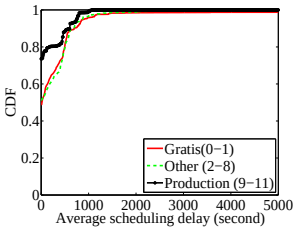


Figure 4.29: CDF of scheduling delay for CBS

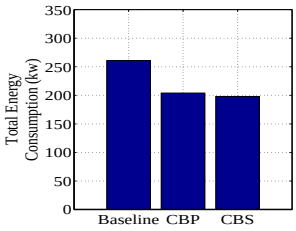


Figure 4.30: Comparison of Energy Consumption

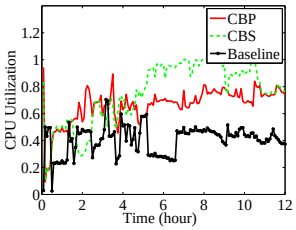


Figure 4.31: CPU utilization in the data center

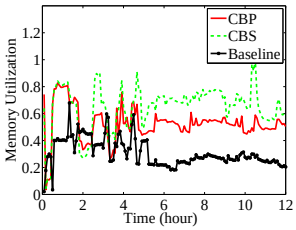


Figure 4.32: Memory utilization in the data center

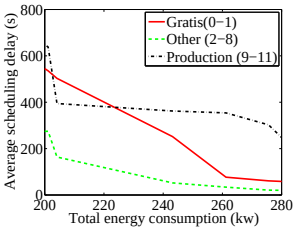


Figure 4.33: Energy vs. Scheduling Delay for CBP

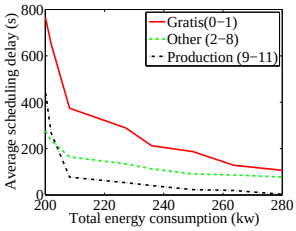


Figure 4.34: Energy vs. Scheduling Delay for CBS

much less than the number of machines selected by the baseline algorithm. Furthermore, It can be seen that they are able to make intelligent decisions regarding what type of machines to turn on and off.

The CDF of task scheduling delays are shown in Figure 4.27, 4.28 and 4.29, respectively. It can be seen that CBS and CBP can substantially reduce the scheduling delay compared to the baseline algorithm. The CPU and memory utilizations of the baseline, CBS and CBP are compared in Figure 4.31 and Figure 4.32, respectively. It can be seen from the diagrams that the baseline achieves low utilization for both CPU and memory. This is because the baseline only ensures the total provisioned capacity is $\frac{1}{80\%}$ times the required capacity, and does not consider the type of machines provisioned. As soon as the most energy efficient (i.e., HP DL585 G7) machines were all turned on, it began to make wrong decisions regarding the type of machines to be turned on. As a result, many tasks can not be scheduled in the newly provisioned machines, resulting in low utilization and high scheduling delay. In contrast, both CBS and CBP can significantly outperform the baseline algorithm in terms of both resource utilization and scheduling delay. Furthermore, CBS generally outperforms CBP in our experiments. This is because CBS uses dedicated containers for scheduling, thus ensures large tasks can be scheduled quickly. In contrast, CBP does not provide guaranteed resources for scheduling large tasks, making them more difficult to schedule.

To better understand the difference between CBS and CBP, we also varied the value of the over-provisioning factor to produce different trade-offs between energy and the average scheduling delay. The results are shown in Figure 4.33 and Figure 4.34, respectively. We found CBS generally produces a better trade-off between scheduling delay and energy savings. This observation can be explained as follows: As CBS pre-allocates containers for scheduling, the variability in task scheduling delay is much smaller. This variability has strong impact on the average scheduling delay, because most of the scheduling delays in our experiments are causes by a small fraction of "difficult-to-schedule" tasks, as suggested in Figure 4.27, 4.28 and 4.29. Furthermore, Figure 4.33 shows that CBP incurs large scheduling delay for production tasks. This is because these production tasks are typically very large. As CBP still relies on the first-fit algorithm to schedule tasks, it does not ensure large production tasks can be scheduled immediately, unless the over-provisioning factor is set to a large value. Based on these observations, we conclude that CBS can slightly outperform CBP especially when energy consumption must be minimized. However, CBS is more restrictive in terms of scheduling policy, making it less adaptive to different scheduling requirements found in practice. Thus, the cloud provider must carefully analyze these trade-offs in order to decide whether CBS or CBP should be used in a given scenario.

4.10 Conclusion

Dynamic capacity provisioning has become a promising solution for reducing energy consumption in data centers in recent years. However, existing work on this topic has not addressed a key challenge, which is the heterogeneity of both workloads and physical machines. In this chapter, we first provide a characterization of both workload and machine heterogeneity found in one of Google's production compute clusters. Then we present Harmony, a heterogeneity-aware framework that dynamically adjusts the number of machines to strike a balance between energy savings and scheduling delay, while considering the reconfiguration cost. Through experiments using Google workload traces, we found Harmony can yield large energy savings while significantly improving task scheduling delay.

Chapter 5

PRISM: Fine-Grained Resource-Aware Scheduling for MapReduce

5.1 Introduction

Businesses today are increasingly reliant on large-scale data analytics to make critical day-to-day business decisions. This shift towards data-driven decision making has fueled the development of MapReduce [46], a parallel programming model that has become synonymous with large-scale, data-intensive computation. In MapReduce, a job is a collection of *Map* and *Reduce* tasks that can be scheduled concurrently on multiple machines, resulting in significant reduction in job running time. Many large companies, such as Google, Facebook, Amazon, and Yahoo!, routinely use MapReduce to process large volumes of data on a daily basis. Consequently, the performance and efficiency of MapReduce frameworks have become critical to the success of today's Internet companies.

A central component to a MapReduce system is its job scheduler. Its role is to create a schedule of Map and Reduce tasks spanning one or more jobs, that minimizes job completion time and maximizes resource utilization. A schedule with too many concurrently running tasks will result in heavy resource contention and long job completion time. Conversely, a schedule with too few concurrently running tasks will have poor resource utilization.

The job scheduling problem becomes significantly easier to solve if we can assume all map tasks (and similarly, all reduce tasks) have homogenous resource requirements in terms

of CPU, memory, disk and network bandwidth. Indeed, current MapReduce systems, such as Hadoop MapReduce Version 1, make this assumption to simplify the scheduling problem. These systems use a simple slot-based resource allocation scheme, where physical resources on each machine are captured by the number of identical slots that can be assigned to tasks. Unfortunately, in practice, run-time resource consumption varies from task to task and from job to job. Several recent studies [63, 123] have reported that production workloads often have diverse utilization profiles and performance requirements [34]. Failing to consider these resource properties and job usage characteristics can potentially lead to inefficient run-time job schedules with low resource utilization and long job execution time.

Motivated by this observation, several recent proposals, such as Resource-Aware Adaptive Scheduling (RAS) [90] and Hadoop MapReduce Version 2 (also known as Hadoop NextGen and Hadoop Yarn) [16], have introduced resource-aware job schedulers to the MapReduce framework. These schedulers specify a fixed size for each task in terms of required resources (e. g. CPU and memory), thus assuming the run-time resource consumption of the task is stable over its life time. However, this is not true for many MapReduce jobs. In particular, it has been reported that the execution of each MapReduce task can be divided into multiple phases of data transfer, processing and storage [61]. A *phase* is a subprocedure in the task that has a distinct purpose and can be characterized by the uniform resource consumption over its duration. As we shall demonstrate in Section 5.2, the phases involved in the same task can have different resource demand in terms of CPU, memory, disk and network usage. Therefore, scheduling tasks based on fixed resource requirements over their durations will often cause either excessive resource contention by scheduling too many simultaneous tasks on a single machine, or significant resource under-utilization by scheduling too few.

In this chapter, we present PRISM, a **P**hase and **R**esource **I**nformation-aware **S**cheduler for **M**apReduce clusters that performs resource-aware scheduling at the level of task phases. Specifically, we show that for most MapReduce applications, the run-time task resource consumption can vary significantly from phase to phase. Therefore, by considering the resource demand at the phase level, it is possible for the scheduler to achieve higher degrees of parallelism while avoiding resource contention. To this end, we have developed a phase-level scheduling algorithm with the aim of achieving high job performance and resource utilization. Through experiments using a real MapReduce cluster running a wide-range of workloads, we show PRISM delivers up to 18% improvement in resource utilization while allowing jobs to complete up to 1.3× faster than current Hadoop schedulers.

The rest of this chapter is organized as follows. Section 5.2 provides an overview of phases involved in MapReduce job execution. We describe the phase-level task usage characteristics and our motivation in Section 5.3. Section 5.4 introduces PRISM and

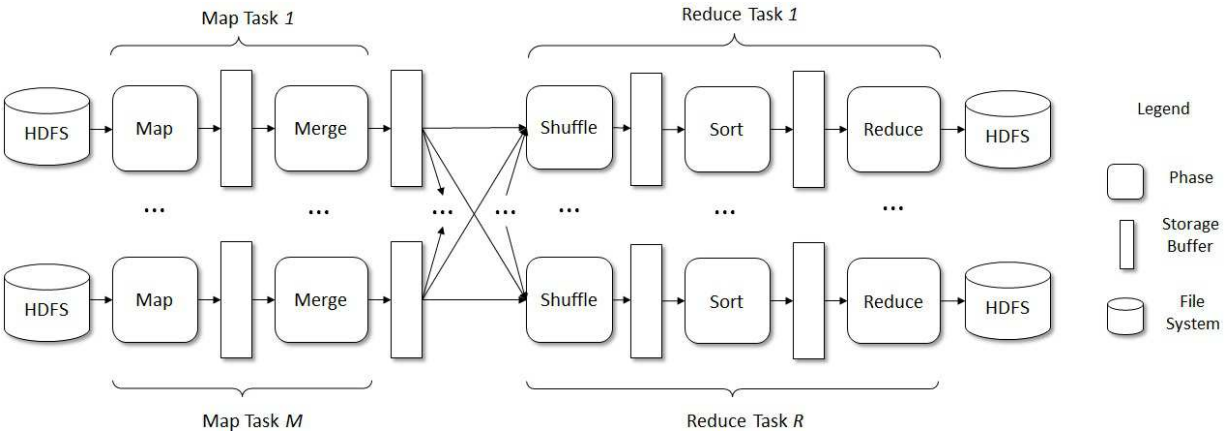


Figure 5.1: Phases involved in the Execution of a Typical MapReduce Job

describes its architecture. The phase-level scheduling algorithm is presented in details in Section 5.5. Our experimental evaluation of PRISM is provided in Section 5.6. Finally, we conclude this chapter in Section 5.7.

5.2 Background

The current MapReduce frameworks such as Apache Hadoop MapReduce [1] adopt a centralized scheduling scheme. A MapReduce cluster consists of a large number of commodity machines with one node serving as the master and the others acting as slaves. The master node runs a resource manager (also known as a job tracker) that is responsible for scheduling tasks on slave nodes. Each slave node runs a local node manager (also known as a task tracker) that is responsible for launching and allocating resources for each task. To do so, the task tracker launches a Java Virtual Machine (JVM) that executes the corresponding map or reduce task.

Current Hadoop job schedulers perform task-level scheduling, where tasks are considered as the finest granularity for scheduling. However, if we examine the execution of each task, we can find that a task consists of multiple phases, as illustrated in Figure 5.1. In particular, a map task can be divided into 2 main phases: map and merge. In the map phase, the mapper fetches the input data block from the Hadoop Distributed File System (HDFS) [2] and applies the user-defined map function on each record. The map function generates records that are serialized and collected into a buffer. When the buffer becomes

full (i.e., content size exceeds a pre-specified threshold), the content of the buffer will be written to the local disk in the background. Finally, the mapper executes a merge phase to group the output records based on the intermediary keys, so that the records can be easily fetched by the reducers.

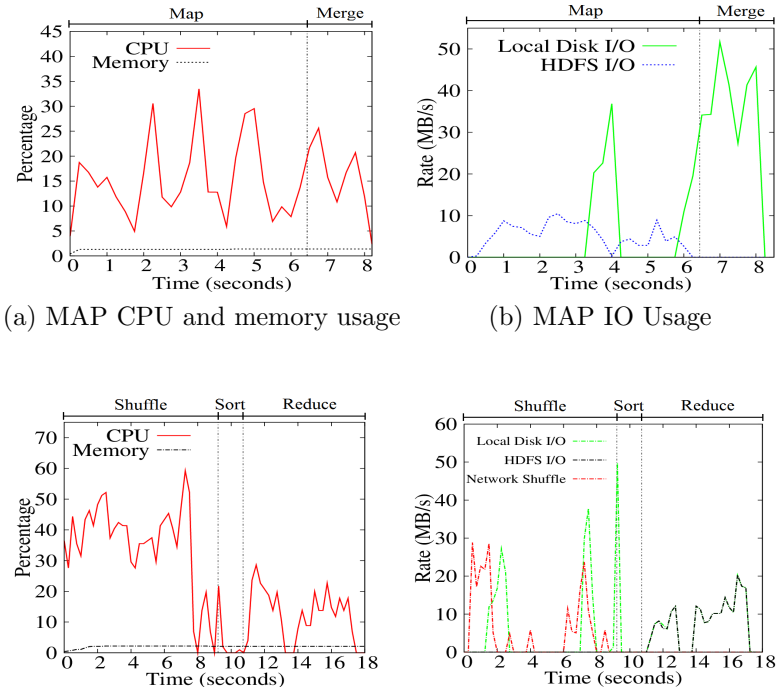
Similarly, the execution of a reduce task can be divided into 3 phases: shuffle, sort, and reduce. In the shuffle phase, the reducer fetches the output records from the local storage of each map task and then places it in a storage buffer that can be either in memory or on disk depending on the size of the content. At the same time, the reducer also launches one or more threads to perform local merge sort in order to reduce the running time of the subsequent sort phase. Once all the map output records have been collected, the sort phase will perform one final sorting procedure to ensure all collected records are in order. Finally, in the reduce phase, records are processed according the user-defined reduce function in the sorted order, and the output is written to the HDFS.

Different phases can have different resource consumption characteristics. For instance, the shuffle phase often consumes significant network I/O resources as it requires collecting outputs from all completed map tasks. In contrast, the map and reduce phases mainly process the records on local machines, thus they typically demand greater CPU resources than network bandwidth. In the next section, we provide empirical evidence to show that the run-time task resource consumption can change significantly across phase boundaries.

5.3 Phase-Level Resource Requirements

In this section we experimentally analyze the run-time task resource requirements in each phase for various Hadoop jobs. We deployed Apache Hadoop 0.20.2 on a 16 node cluster, with one node acting as the master managing the other 15 slave nodes. Each machine has a Quad-core Xeon CPU with 12GB of memory and 1TB local disk storage. We modified the default task tracker in Hadoop 0.20.2 to monitor the execution of phases inside each task.

In our experiments, we evaluate the phase-level resource requirements across various jobs, including the standard examples provided by the Hadoop MapReduce distribution Gridmix2 [3] and the PUMA Benchmarks [4]. We found that task resource usage can change significantly from phase to phase for a large variety of jobs in both benchmarks. For example, Figure 5.2 shows the resource consumption over time of a single map and reduce task for the `sort` job. Figure 5.2a and Figure 5.2b show that even though the



(a) MAP CPU and memory usage

(b) MAP IO Usage

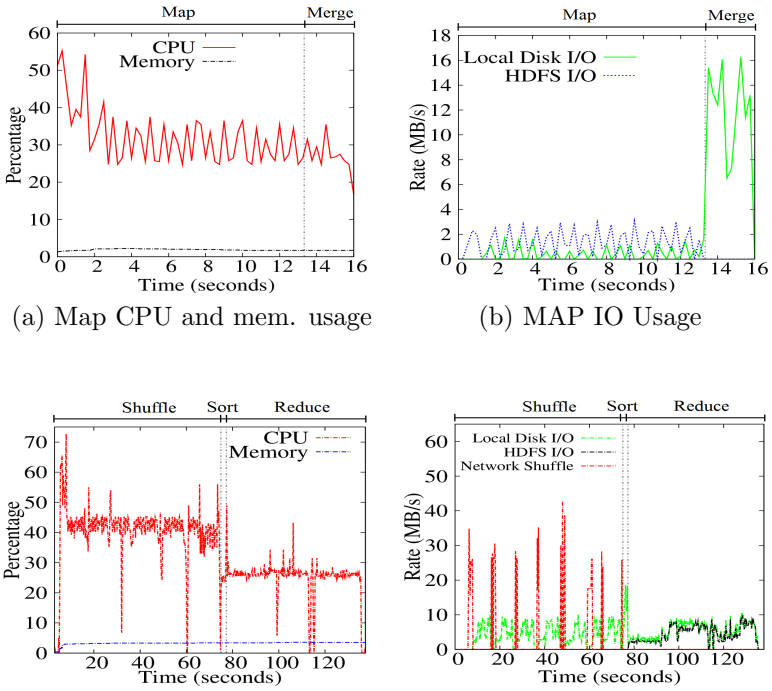
(c) REDUCE CPU and memory usage

(d) REDUCE IO Usage

Figure 5.2: Job Profile for `sort`

CPU usage of the map task remains reasonably stable over time,(around 15% on average) the I/O usage increases significantly as the task progresses from the map phase to the merge phase. The low I/O usage is the result of the map phase incrementally reading the input key-value pairs from the HDFS system. In contrast, the merge phase has high I/O usage because it is responsible for grouping all intermediary key-values pairs within a short period of time.

Similarly, Figure 5.2c and Figure 5.2d show that the run-time resource consumption of the reduce task changes from the shuffle phase to the reduce phase. The reason is that the shuffle phase fetches the intermediary key-values pairs from the map tasks, and performs partial merge on the fetched key-value pairs. As a result, it consumes both CPU and network I/O resources. However, once the reduce phase begins, the reducer only needs to focus on applying the reduce function to each key-value pair to produce the final output. Because the reduce function of the `sort` job is just a simple pass-through function, the



(a) Map CPU and mem. usage

(b) MAP IO Usage

(c) REDUCE CPU and memory usage

(d) REDUCE IO Usage

Figure 5.3: Job Profile for `InvertedIndex`

CPU usage of the reduce phase is lower than that of the shuffle phase.

We also analyze the `InvertedIndex` job in the PUMA benchmark. Figure 5.3 shows that the map tasks and reduce tasks of the `InvertedIndex` job have different running times compared to the `sort` job. Furthermore, unlike in the `sort` job, the map tasks of the `InvertedIndex` job consume almost 8× less I/O resources during map phase than the merge phase.These observations suggest that the run-time task resource consumption is dependent on the phase in which the task is currently executing. Therefore, ignoring the phase-level resource characteristics will lead to inaccurate job usage profiles, which in turn will cause the job scheduler to make inefficient job scheduling decisions.

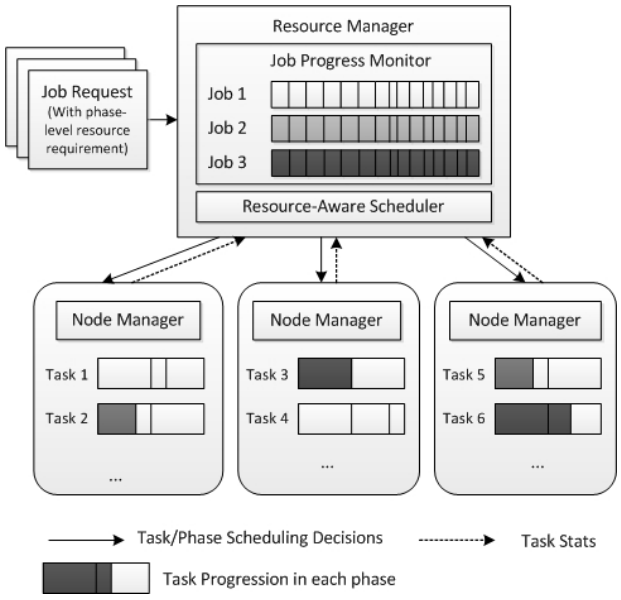


Figure 5.4: System Architecture

5.4 System Overview

Motivated by the observation that task usage is phase-dependent, we present PRISM, a new fine-grained resource-aware scheduler that performs scheduling at phase-level. Unlike existing MapReduce schedulers that only allow job owners to specify resource requirements at task-level, PRISM allows the job owners to specify phase-level resource requirements.

An overview of the PRISM architecture is shown in Figure 5.4. PRISM consists of three main components: a resource-aware scheduler at the master node, local node managers that coordinate phase transitions with the scheduler, and a job progress monitor to capture phase-level progress information. In PRISM, once a task has finished executing a particular phase, it must request the local node manager for permission to start the next phase. The local node manager forwards the permission request to the scheduler through the regular heartbeat message. Given a job's phase-level resource requirements and its current progress information, the scheduler decides whether to start a new task, or allow a paused task to begin its next phase.

In order to perform phase-level scheduling, PRISM requires phase-level resource information for each job. Existing state-of-the-art resource profilers, such as Starfish [61], can already provide accurate phase-level resource information for PRISM. In the absence of

phase-level resource information, PRISM can fall back to use task-level resource information specified for Hadoop Yarn to schedule phases for that job.

Finally, even though the flexibility of phase-based scheduling should allow the scheduler to improve both resource utilization and job performance over existing MapReduce schedulers, realizing such a potential is still a challenging problem. This is because pausing the task execution at run-time may delay the completion of the current and subsequent tasks, which may increase the job completion time (these delayed tasks are commonly referred to as stragglers [46]). Thus, the scheduler must avoid introducing stragglers when switching between phases. In the following sections, we will describe how PRISM overcomes this challenge.

5.5 Scheduler Design

In this section, we describe in detail the design of PRISM's phase-based scheduling algorithm. We first describe the design rationale of the scheduling algorithm in Section 5.5.1, and then provide the details of our algorithm in Section 5.5.2.

5.5.1 Design Rationale

The responsibility of a MapReduce job scheduler is to assign tasks to machines with consideration for both efficiency and fairness [34, 117]. To achieve efficiency, job schedulers must maintain high resource utilization in the cluster. Job running time is another possible measure for efficiency [117], as a lower job running time implies that resources are more efficiently utilized for job execution. In contrast, fairness ensures that resources are fairly divided among jobs such that no job will experience starvation due to unfair resources allocation. However, simultaneously achieving both fairness and efficiency in the context of multi-resource scheduling has been shown to be challenging, as there is usually a trade-off between these properties [66, 117].

Fair scheduling algorithms generally run an iterative procedure by identifying users that experience the highest degree of unfairness (i.e. deficit) in each iteration, and schedule tasks that belong to those users to improve the overall fairness of the system. As described in Section 2.2.2, there are many possible fairness criteria. For instance, Isard et. al. [63] aims at ensuring that when multiple jobs share the same cluster, each job should experience similar performance gain or performance loss in terms of running time. In this context, considering a cluster that currently has J jobs running, and each job is executing n_j tasks

concurrently. Suppose each job j can execute N_j tasks simultaneously when it is given exclusive access to the cluster, then the fair share of each job j is defined as

$$FS_j = \frac{n_j}{N_j} \tag{5.1}$$

we refer to this type of fairness as *running-time fairness*, as it tries to equalize the performance gain (or loss) of individual jobs. This fairness criterion is also supported by original Hadoop fair scheduler, as the scheduler tries to equalize the number of slots that each job receives. In this case, each job will experience similar speed up (or slow down) because it gets similar share of resources in terms of the number of slots.

Similarly, Ghodsi et. al. defined the dominant resource fairness (DRF) which aims at equalizing share of each individual's most highly demanded (i.e., dominant) resource. Specifically, considering a cluster of R types of resources whose capacity for each type of resources $r \in R$ is C_r. Assuming for each individual j, the resource consumption of type r resource is $c_j r$, the dominant share of individual j can be computed as

$$FS_j = \max_{r \in R} \left\{ \frac{c_{jr}}{C_r} \right\} \tag{5.2}$$

Regardless of the fairness criterion used, at run-time the objectives of a fair scheduler is to equalize FS_j by scheduling tasks belonging to the job with the minimum share.

However, directly applying a fair scheduling algorithm for phase-level scheduling requires additional considerations. In particular, given a set of phases that can be scheduled on a machine, the scheduling algorithm must consider the resource requirements of the different phases and the dependency between phases to determine a valid schedule that minimizes job running time. For example, the scheduler needs to consider possible cascading delays, due to the sequential ordering of phases in a task, when making scheduling decisions. In many cases, such delays can also propagate to subsequent phases in the same job, causing them to be delayed as well. For example, even though the execution of a shuffle phase of a reduce task can overlap with the execution of a merge phase of a map task, the shuffle phase cannot finish unless all merge phases of the map tasks have finished. Thus, when choosing between scheduling merge phases and shuffle phases, it is preferable to give sufficient resources to merge phases to allow all of them to finish faster, instead of allocating most of the resources to the shuffle phase and delaying the completion of merge phases.

To address the issue of limited concurrency as a result of phase dependencies, we ensure there is a high degree of task-level concurrency by deploying a sufficient number of running

map and reduce tasks. While a task is running, we also want to ensure each phase within the task is not severely delayed in order to avoid creating stragglers. To achieve both objectives, given a set of phases that can be scheduled on a machine, the scheduler assigns a utility value to each phase which indicates the benefit of scheduling the phase. The scheduler will then schedule the phases in decreasing value of their utility. The utility value is phase-dependent, because phases have different dependencies. If a phase is map or shuffle, scheduling the phase implies scheduling a new map or reduce task. In this case, the utility of the phase is determined by the increase in parallelism from running an additional task. For other phases, the utility is determined by the urgency to complete the phase. A simple metric for measuring urgency is the number of seconds that a task has been paused due to phase-level scheduling. If the task has been paused for a long time, it becomes more urgent to schedule its remaining phases in order to avoid creating a straggler.

5.5.2 Algorithm Description

We formally introduce our scheduling algorithm in this section. Specifically, each job j in the system consists of two types of tasks: map tasks M and reduce task R. Let $\tau(t) \in \{M, R\}$ denote the type of a task t. Given a phase i belonging to a task t that can be scheduled on a machine n, we can define the utility function of assigning a phase i to machine n as:

$$U(i,n) = U_{fairness}(i,n) + \alpha \cdot U_{perf}(i,n) \tag{5.3}$$

where $U_{fairness}$ and U_{perf} represent the utilities for improving fairness and job performance, respectively, and α is an adjustable weight factor. If we set α to a value close to zero, then the algorithm will greedily schedule phases according to the improvement in fairness. Notice that considering job performance objectives will not severely hurt fairness. When a user is severely below its fair share, scheduling any phase with non-zero resource requirement will only improve her fairness. The exact value of α can be determined based on experience.

Now we describe each of the terms in equation (5.3) in detail. We define

$$U_{fairness}(i,n) = U^{before}_{fairness}(i,n) - U^{after}_{fairness}(i,n) \tag{5.4}$$

where $U^{before}_{fairness}(i,n)$ denotes the fairness measure of the user before scheduling i on n and $U^{after}_{fairness}(i,n)$ is the new fairness measure of the user after scheduling i on n. For example,

we can define

$$U^{before}_{fairness}(i,n) = \sum_{j=1}^{J} \left| FS^{before}_{j} - FS^{before*}_{j} \right| \tag{5.5}$$

$$U^{after}_{fairness}(i,n) = \sum_{j=1}^{J} \left| FS^{after}_{j} - FS^{after*}_{j} \right| \tag{5.6}$$

where J denotes the job that are currently running, FS^{before}_{j} and FS^{after}_{j} denote the fair share of job j before and after i is scheduled, and FS^{*}_{j} is the average fair share, (i.e. $FS^{before*}_{j} = \frac{1}{J}\sum_{j=1}^{J} FS^{before}_{j}$ and $FS^{after*}_{j} = \frac{1}{J}\sum_{j=1}^{J} FS^{after}_{j}$).

On the other hand, $U_{perf}(i,n)$ is more difficult to compute. As mentioned previously, if i is the first phase of a map (or reduce) task t, then $U_{perf}(i,n)$ measures the gain in parallelism in terms of the number of running map tasks (or reduce tasks). Otherwise, if i is a subsequent phase of task t, then $U_{perf}(i,n)$ measures the gain in shortening the running time of task t. Formally, we define

$$U_{perf}(i,n) = \begin{cases} U_{task}(i,n) & i\text{is the first phase of a task} \\ U_{phase}(i,n) & \text{Otherwise} \end{cases} \tag{5.7}$$

Even though PRISM does not specify the function for computing the utility of a phase, in our current implementation, we have chosen $U_{task}(i,n)$ to be

$$U_{task}(i,n) = \frac{N_{remaining}}{\max\{N_{current}, \epsilon\}} - \frac{N_{remaining}}{N_{current}+1} \tag{5.8}$$

where $N_{remaining}$ denotes the number of remaining tasks of type $\tau(t)$ (i.e. the number of remaining tasks of the same type as t), and $N_{current}$ denotes the number of tasks of type $\tau(t)$ that are running. The variable ϵ is used to prevent dividing by 0. Intuitively, $U_{task}(i,n)$ measures the gain in parallelism if the number of running tasks is increased from $N_{current}$ to $N_{current}+1$.

On the other hand, let T^{t}_{wait} denote the number of seconds that task t has been paused due to phase-based scheduling. The utility for scheduling a non-leading phase i of task t can be expressed as a function $p(\cdot)$ of T^{t}_{wait}:

$$U_{phase}(i,n) = p(T^{t}_{wait}) \tag{5.9}$$

There are many possible choices for $p(\cdot)$. For example, we can define $p(\cdot)$ as a linear function (i.e. $p(T^{t}_{wait}) = a \cdot T^{t}_{wait} + b$ for constants a and b), which would increase the

Algorithm 5 Phase-Level Scheduling Algorithm

```
Upon receiving a status message from a task tracker on machine n
Compute the resource utilization of machine n
PhaseSelected ← {∅}
CandidatePhases ← {∅}
for each job j in the system do
    for each schedulable phase i ∈ j do
        CandidatePhases ← CandidatePhases ∪ {i}
    end for
end for
while CandidatePhases ≠ ∅ do
    for i ∈ CandidatePhases do
        if i is not schedulable on n given current utilization then
            CandidatePhases ← CandidatePhases\{i}
            continue;
        end if
        Compute the utility U(i, n) as in equation (5.3)
        if U(i, n) ≤ 0 then
            CandidatePhases ← CandidatePhases\{i}
        end if
    end for
    if CandidatePhases ≠ ∅ then
        i ← task with highest U(i, n) in the CandidatePhases
        PhaseSelected ← PhaseSelected ∪ {i}
        CandidatePhases ← CandidatePhases\{i}
        Update the resource utilization of machine n
    end if
end while
return PhaseSelected
```

utility of scheduling i to increase linearly with the number of seconds that the task has been paused. However, in our implementation, we have chosen $p(\cdot)$ to be a quadratic function $p(T^t_{wait}) = a \cdot p(T^t_{wait})^2 + b$. The intuition to using a quadratic function is to increase the urgency for scheduling i more rapidly if i has been paused for a long time. However, PRISM can adopt any type of utility function $p(\cdot)$ as long as it is a monotonically increasing function.

Finally, the scheduling algorithm used by our phase-based scheduler is illustrated by Algorithm 1. Specifically, upon receiving the status message from a node manager running

on machine n, we first compute the utilization u of the machine using job's phase-level profile (Line 2). We then compute a set of schedulable candidate phases (Line $4-9$), and select phases in an iterative manner. In each iteration for each schedulable phase $i \in P(j)$ of each job j, we compute the utility function $U(i,n)$ according to equation (5.3) (Line 16). Then we select the phase with the highest utility for scheduling (Line $22-23$), and update the resource utilization of the machine (Line 25). Then the algorithm repeats by recomputing the utility of all the phases in the candidate set, and select the next best phase to schedule. The algorithm ends when the candidate set is empty, which means there is no suitable phase to be scheduled. As for the running time, assuming there are N tasks in the system[1] and each machine can schedule at most k tasks, the running time of the algorithm is $O(Nk)$.

5.6 Experiments

We have implemented PRISM in Hadoop 0.20.2. Implementing this architecture requires minimal change to the existing Hadoop architecture (around 700 lines of code). Even though there are several fairness criteria we can implement in PRISM, currently we implemented the running-time fairness proposed by Isard et. al. [63], where the fair share of each job is computed according to equation (5.1).

We deployed PRISM in a compute cluster which consists of 16 compute nodes. Each compute node has 4-core 2.13GHz Intel Xeon E5606 processors, 12G RAM, 1TB of local high speed hard drive, and runs 64-bit Ubuntu 11.10 OS. The network interface card (NIC) installed on each node is capable of handling up to 1Gb/s of network traffic. Each node is connected to a top-of-rack switch and can communicate with others via a $1Gb/s$ link.

We have chosen two benchmarks to evaluate the performance of PRISM: Gridmix 2 and PUMA. Gridmix 2 [3] a standard benchmark included in the Hadoop distribution. For Gridmix 2 we have chosen 3 jobs for performance evaluation: `MonsterQuery (MQ)`, `WebDataScan (WDS)` and `Combiner (CM)`. Similarly, PUMA [4] is a MapReduce benchmark developed at Purdue University. We have selected 4 jobs for performance evaluation: `sort (SRT)`, `self-join (SJ)`, `inverted-index (II)` and `classification (CL)`. We chose these jobs because they contain a variety of resource usage characteristics. For example, `sort` and `MonsterQuery` are I/O intensive jobs, whereas `Combiner` and `self-join` are more CPU intensive. A mixture of jobs with different resource requirements allows us to better evaluate the performance of PRISM.

[1] As each task can provide at most one candidate phase, N tasks in the system imply there can be at

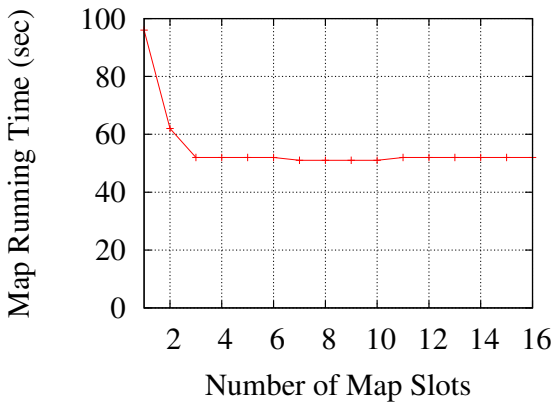


Figure 5.5: Num of Slots vs. Map running time

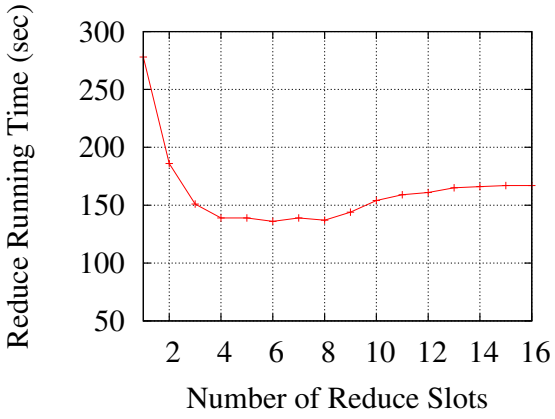


Figure 5.6: Num of Slots vs. Job running time

Job: sorter
Input size: 5GB, Map Count: 40, Reduce Count: 56
Map stage completion: 63s
Reduce stage completion: 147s

Phase	Map	Merge	Shuffle	Sort	Reduce
t_i (s)	7.43	1.25	9.07	0.64	9.69
$CPU(\%)$	17.42	14.35	21.58	7.5	8.21
Mem (%)	1.35	1.40	2.11	2.37	2.33
LFS(MB/s)	3.98	34.31	5.71	11.17	5.29
HDFS(MB/s)	7.17	0	0	0	5.30
Shuffle(MB/s)	0	0	7.16	0	0

Figure 5.7: A Job Profile for Sort Job

In order to evaluate the benefit brought by phase-level scheduling, it is necessary to compare PRISM to existing task-level resource-aware schedulers. In our experiments, we have chosen Hadoop Yarn 2.0.4 as a competitive task-level resource-aware scheduler. Hadoop Yarn 2.0.4 is a recent version of Hadoop NextGen that allows the users to specify both CPU requirement (i.e. number of virtual cores) and memory requirement (i.e. GB of RAM) of each task. Like the previous version of Hadoop MapReduce, The current Hadoop Yarn

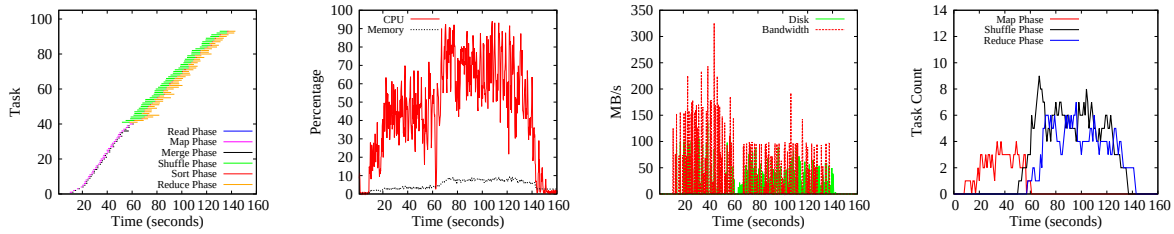
most N candidate phases.

supports both capacity scheduler and fair scheduler. However, only the capacity scheduler supports resource-aware scheduling using both CPU and memory resource requirements. As a state-of-the-art resource-aware scheduler, in our experiments we compare PRISM with Hadoop Yarn using capacity scheduler. As the capacity scheduler is not fair scheduler. Finally, we use Hadoop 0.20.2 with fair scheduler as a (slot-based) baseline for comparing scheduler performance. This also allows us to evaluate the fairness of PRISM because both PRISM and the fair scheduler aim at achieving running-time fairness. In our experiments, we set $\alpha = 1$ to given equal importance to both performance and fairness in PRISM.

5.6.1 Capturing Job Performance Requirements

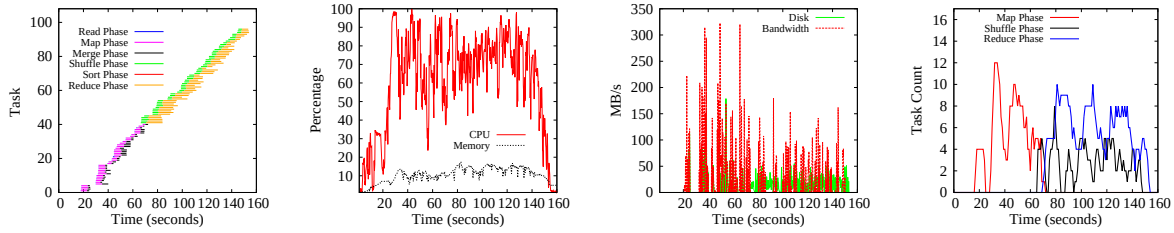
For analysis purposes, we have implemented a simple job profiler that captures the CPU, memory and I/O usage of both tasks and compute nodes. Writing our own profiler allows us to better analyze the fine-grained resource characteristics of individual phases. In our implementation, we monitor the execution of each task and record the start and end time of every phase. As for monitoring run-time resource usage, we rely on linux `top` command to record CPU and memory usage once per second. Network I/O is more difficult to profile. In our current implementation, we modified the Hadoop source code to print the values of I/O counters. The actual disk and network I/O usage over-time can be obtained from Linux utilities such as `iotop` and `nethogs`.

Similar to existing work in [90], we adopt a simple strategy for profiling jobs as follows: given the input parameters (e.g. input data size, number of reduce tasks) of each job, we vary the resource allocation of both map and reduce tasks in the job profile by adjusting the number of slots allocated to map and reduce tasks. Specifically, we first vary the number of map slots to find an optimal number of map slots that minimizes the map completion time. Using this number, we then vary the number of reduce slots to find an optimal number of reduce slots that minimizes the overall job completion time. Figure 5.5 and Figure 5.6 shows the result for adjusting the number of maps slots and reduces for the `sort` job, respectively. Both figures show that the job running time has a non-linear relationship with the number of slots used. When the number of slots is small (e.g. 2 slots) the job running time becomes long due to the low degree of task-level parallelism imposed by the slot allocation. On the other hand, when the number of slots is large (e.g. 12 slots) the running time again becomes high due to multiple tasks competing for bottleneck resources. For the `sort` job, we found setting the number of map slots and reduce slots to 8 and 6 respectively achieves the optimal running time. The profile for the `sort` job is shown in Figure 5.7. The same process is repeated to create the profiles for all other jobs.



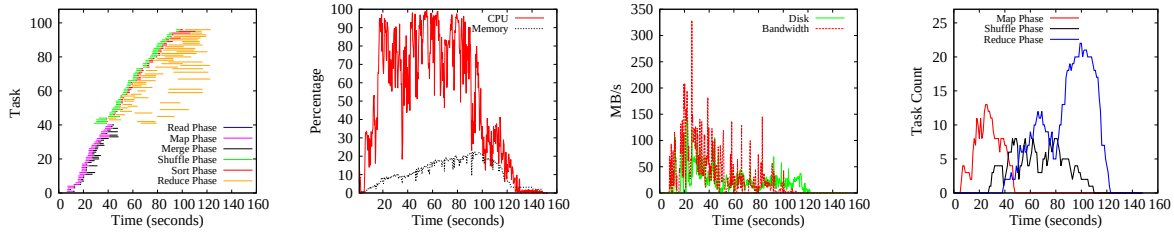
(a) Task Execution Timeline (b) CPU and Mem. Util. (c) I/O Utilization (d) Num. of Phases Running

Figure 5.8: Sorting 5GB data with Fair-Scheduler



(a) Task Execution Timeline (b) CPU and Mem. Util. (c) I/O Utilization (d) Num. of Phases Running

Figure 5.9: Sorting 5GB data with Yarn



(a) Task Timeline (b) CPU and Mem. Util. (c) I/O Utilization (d) Num. of Phases Running

Figure 5.10: Sorting 5GB data with PRISM

To provide fair comparison among all 3 schedulers (i.e., PRISM, Fair Scheduler and Yarn) In our experiments, we repeated the above procedure to find the optimal number of map and reduce slots used by the fair scheduler for each of the workloads (job or benchmarks). For Hadoop Yarn, we set the task container size according to the job profiles. Specifically, the task size is set to the average resource usage across phases weighted by the

duration of each phase. The default configuration of Yarn specifies 16 virtual cores (vCores) per machine. Yarn also requires that the number of vCores per task must take integer values in the job request. In our experiments, we have found the default configuration of Yarn produces lower performance compared to both the Fair Scheduler and PRISM. This is due to the large rounding errors for converting the number of vCores to integer values. Therefore, we modified the default configuration so that each machine provides 128 vCores. This significantly reduces round errors, allowing Yarn to produce comparable performance against both the fair scheduler and PRISM.

5.6.2 Performance Evaluation using Individual Jobs

In our first set of experiments, our goal is to demonstrate the benefit of phase-level scheduling. For this purpose, we run a single `sort` job in a small cluster consisting of only 3 nodes using Fair Scheduler, Yarn and PRISM [2]. The input size is set to $5GB$ in all 3 runs.The number of map and reduce slots used by the Fair Scheduler is set to 8 and 6 as discussed in previous section.

The experiment results for Hadoop fair scheduler, Yarn and PRISM are shown in Figure 5.8, 5.9 and 5.10, respectively. In particular, the fair scheduler is able to complete the job execution in 149 seconds, whereas Yarn finishes the job in 152 seconds. In contrast, PRISM can achieve the same in just 125 seconds (as shown in Figure 5.10a), resulting in a 19% reduction in job running time. To understand the reason behind the performance gain, we first plotted the CPU/Memory usage as well as disk/network I/O usage in Figure 5.8b and 5.8c for Fair Scheduler, in Figure 5.9b and 5.9c for Yarn, and in Figure 5.10b and 5.10c for PRISM. We found Yarn achieves highest utilization while performing slightly worse than the fair scheduler. The main reason is that Yarn has an additional scheduling overhead. Specifically, in order to run a new MapReduce job, Yarn first needs to schedule a job controller called Application Master [16], which will be responsible for monitoring and managing the job execution. This Application Master also consumes cluster resources at run-time, which reduce the resource capacity available for task scheduling. In contrast, PRISM is able to deliver higher utilization for all resource types. The CPU utilization of PRISM is always better than that of Fair scheduler except near the end of the execution.

We also plotted Figure 5.8d, 5.9d and Figure 5.10d to show the number of phases scheduled over time by each scheduler. For clarity of presentation, we only show the plot for the 3 major phases: map, shuffle and reduce. It can be seen that PRISM and

[2]We choose 3 nodes in this experiment mainly to allow us to visualize the execution of the job, as well as to demonstrate the scenarios where PRISM outperforms the fair scheduler

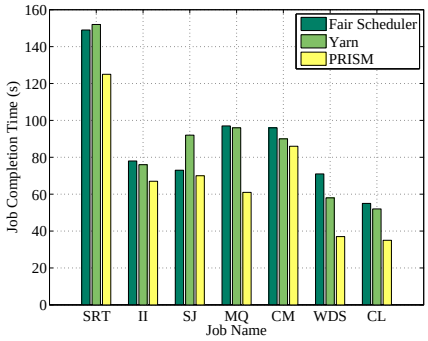
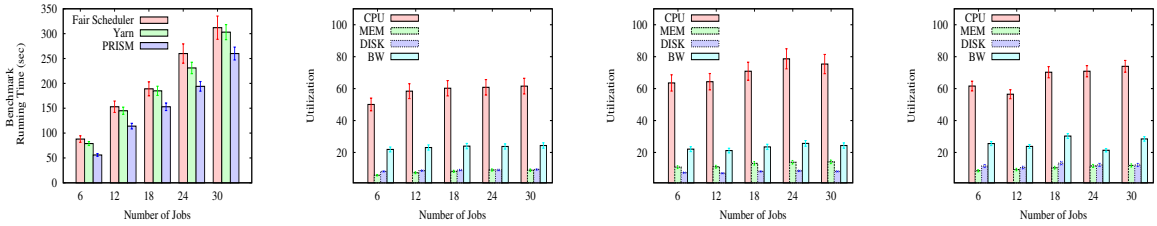


Figure 5.11: Running time of individual jobs

Yarn are able to achieve higher degree of parallelism during the map stage (7 and 6 map tasks running concurrently on average) than the Fair Scheduler. During the reduce stage, as shuffle phases consumes more resources than reduce phases, PRISM recognizes the potential resource bottleneck, and thus delays the start of the reduce phases, allowing more shuffle phases to be scheduled. This makes the shuffle phases to run faster than the Fair scheduler and Yarn (81 seconds for PRISM, 86 seconds for Yarn and 89 seconds for the Fair Scheduler). As reduce phases consume less resources, they can be scheduled in large quantity without causing resource contention. Given the flexibility to separate shuffle phases from reduce phases, PRISM is able to find better schedule for both shuffle and reduce phases without cause resource contention, resulting in significant improvement in job running time.

We have also performed the same experiment for the remaining jobs in the Gridmix 2 and the PUMA benchmark. The results are shown in Figure 5.11. It can be seen that PRISM outperforms both the fair scheduler and Yarn for all the jobs. The reduction in job running time ranges between 5%-38%. Furthermore, we have found that PRISM generally achieves higher reduction in job running time for reduce intensive jobs (e.g. `sort` and `self-join`, where reducers consume more resources than mappers). The main reason behind this observation is that for reduce-intensive jobs, both shuffle and reduce phases take longer time to run and often have drastically different resource consumption characteristics. As a result, PRISM is able to find better schedules compared to both Yarn and fair scheduler, and therefore the gain becomes higher.



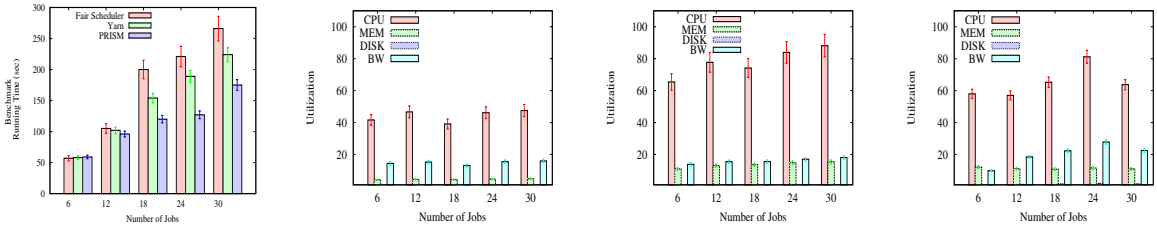
(a) Running Time (b) Using Fair Scheduler (c) Using Yarn (d) Using PRISM

Figure 5.12: Experiment results with the PUMA Benchmark

5.6.3 Performance Evaluation using Benchmarks

We now present our evaluation result using both PUMA and Gridmix 2 benchmarks. In the PUMA benchmark, we vary the number of jobs of each type by $2 \times -10\times$ to create batch workload of different size, and run each of the batch workload using Fair scheduler, Yarn and PRISM. The results for job completion time is shown in Figure 5.12a. It can be seen that PRISM outperforms both Fair scheduler and Yarn in all scenarios. Furthermore, Yarn generally outperforms the Fair scheduler for large workloads, because it is more resource-aware. Figure 5.12b, 5.12c and 5.12d shows the resource utilization of the cluster during the execution of each batch for each scheduler respectively. It can be seen from the diagrams that Yarn achieves the highest utilization, while PRISM generally provider higher resource utilization than the Fair Scheduler. On average, PRISM is able to achieve to up to 24% reduction in job running time The benefit of PRISM for PUMA benchmark mainly comes from the fact that PRISM is able to achieve higher degree of parallelism through better scheduling of phases, resulting in shorter job running time.

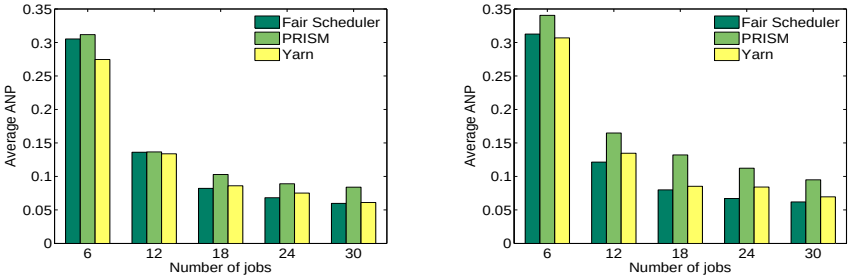
Similarly, for the Gridmix 2 benchmark, we vary the number of jobs of each type by $2 \times -10\times$ to create multiple batches of Gridmix 2 workload. Each batch is then executed on the 16 node cluster using the Fair scheduler Yarn and PRISM. The results for job running time and resource utilizations for both schedulers are shown in Figure 5.13a, 5.13b, 5.13c and 5.13d, respectively. The results are similar to that of the PUMA workload. These results suggest that PRISM is able to achieve shorter job running time while maintaining high resource utilization for large workloads containing a mixture of jobs, which is commonly seen in production MapReduce clusters.



(a) Running Time (b) Using Fair Scheduler (c) Using Yarn (d) Using PRISM

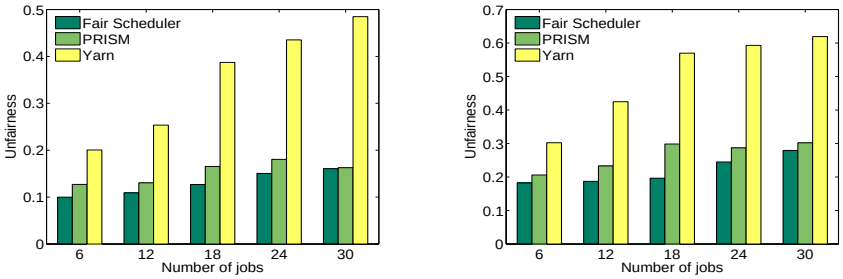
Figure 5.13: Experimental results with the GridMix 2 Benchmark

So far we have only analyzed the aggregate workload running time and resource utilization. However, these objectives should be achieved at the cost of introducing poor job fairness. Therefore, we have also measured the *Application Normalized Performance (ANP)* and the *unfairness* as introduced by Isard et. al. in Quincy [63]. The ANP of a job is the ratio between the ideal job running time (when the job is given sufficient capacity to run at full speed) to actual job running time. Thus, the higher the ANP value is, the better the scheduler performs in terms of improving job running time. The unfairness, on the other hand, is the coefficient of variation (CV) of the ANP values across all jobs in the batch. The intuition is that a fair scheduler should ensure all jobs experience similar amount of delay regardless of the current utilization of the cluster. Therefore, a small CV of ANP values indicates a high level of fairness achieved by the scheduler. The results of ANP and unfairness for both PUMA and Gridmix workload are shown in Figure 5.14a, 5.14b, 5.15a and 5.15b respectively. Specifically, Figure 5.14a and 5.14b show that PRISM is able to achieve high ANP values compared to both Fair Scheduler and Yarn. However, it delivers slightly higher unfairness values than the Fair Scheduler, as shown in Figure 5.15a and 5.15b. We believe this is due to the fact that PRISM tries to find a balance between performance and resource-awareness, thus due to resource constraints it is not possible to achieve ideal fairness values. However, as the difference is relatively small between these two schedulers, we believe sacrificing small fairness for the sake of improving resource utilization and job running time is beneficial to the overall performance of the cluster. Finally, we found that Yarn achieves the worst unfairness values. This is because it uses the capacity scheduler, which does not take fairness into consideration when making scheduling decisions.



(a) ANP using PUMA (b) ANP using Gridmix

Figure 5.14: ANP Result for PUMA and Gridmix Benchmarks



(a) Unfairness using PUMA (b) Unfairness using Gridmix

Figure 5.15: Fairness Result for PUMA and Gridmix Benchmarks

5.7 Conclusion

MapReduce is a popular programming model for data intensive computing. However, despite recent efforts toward designing resource-efficient MapReduce schedulers, existing work mainly focuses on designing task-level schedulers, and is oblivious to the fact that the execution of each task can be divided into phases with drastically different resource consumption characteristics.

To address this limitation, in this chapter we present PRISM, a fine-grained resource-aware MapReduce that coordinates task execution at the level of phases. We first demonstrate the run-time resource usage can vary significantly over time for a variety of MapReduce jobs. We then present a phase-level job scheduling algorithm that improves job

execution without introducing stragglers. In a 16-node Hadoop cluster running standard benchmarks, we demonstrated that PRISM offers high resource utilization and provides 1.3× improvement in job running time compared to the current Hadoop schedulers.

Chapter 6

Conclusion

With the rapid development of hardware and software virtualization technologies, the past few years have witnessed the rise of cloud computing, a paradigm that harnesses massive resource capacity of data centers to support Internet services and applications in a scalable, flexible, reliable and cost-efficient manner. However, despite its recent success, devising efficient resource allocation schemes for cloud data centers still remains a major challenge, as it requires carefully addressing the operational concerns of both the service providers and the cloud providers, while taking into consideration the heterogeneous characteristics of data centers in terms of their locations, physical data center architectures (e.g. data center network topologies and machine configurations) as well as the heterogenous workload characteristics in terms of resource requirement, resource usage, performance objectives and importance level (e.g. priority).

This thesis tackles three key challenges in resource management for cloud computing problems. The first contribution of this dissertation is a solution to the service placement problem in geographically distributed clouds. Given a variety of data center locations and time-varying service demand from Internet users, we have devised a scheme based on Model Predictive Control (MPC) that dynamically adjust the placement of service applications in order to minimize total resource consumption while satisfying service provider's SLA requirement. We also analyzed the problem in a multi-service provider scenario, where service providers compete for resources in preferred data centers. We analyzed the outcome of the resulting competition using game theory, and devised a mechanism to help achieve near-optimal social welfare among the service providers. Experiments show our control algorithms can significantly outperform the baseline solutions in both single service provider (20% reduction in cost) and multi-service provider scenarios (15% improvement in social welfare).

The second contribution of dissertation deals with the energy management problem in data centers. Specifically, dynamic capacity provisioning is a promising approach for reducing energy consumption by dynamically adjusting the number of active machines to match resource demands. However, despite extensive studies of the problem, existing solutions often assume that servers and resource demands are homogeneous. Through analysis using traces from a production cloud cluster at Google, we found that both resource demands and machine configurations are highly heterogenous. This arises the problem of determining not only the number of machines, but also types of machines, to be turned on and off in order to save energy in the presence of heterogeneous resource demands. To this end, we designed Harmony, a heterogeneity-aware resource management system for dynamic capacity provisioning in cloud computing environments. We use clustering algorithms to divide the workload into distinct task classes with similar characteristics in terms of resource demand, running time and performance requirements. Then we present a control-theoretic solution for dynamically adjusting the number of machines of each type in order to minimize total energy consumption in the data center while achieving the desired Service Level Objectives (SLO) in terms of scheduling delay. Experiments show the proposed approach can reduce energy consumption by up to 28% while achieving low average scheduling delay for individual task classes.

Finally, this dissertation presents PRISM, a novel MapReduce scheduler that is capable of scheduling task executions at the level of phases. The design of PRISM is motivated by the fact that the execution of MapReduce tasks can be divided into multiple phases with different resource characteristics. However, none of the existing schedulers has leveraged phase-level resource demand information for task scheduling. In PRISM, we have designed a phase-level scheduling algorithm that finds a good tradeoff between job performance and cluster resource utilization, taking into consideration the change in resource usage across phases. Through experiments we found PRISM can achieve up to 30% improvement for I/O intensive MapReduce workloads compared to existing schedulers (e.g. Hadoop Fair Scheduler and Hadoop NextGen).

In this dissertation, we have proposed solutions to the above three problems. Through experiments and simulations using real data traces, we have demonstrated their superior performance. However, there are limitations pertaining to each of the solutions that require further investigation:

Dynamic Service Placement

The work presented in this dissertation has only considered the placement of services in a cloud environment where a single cloud provider is present. In reality, today's cloud market

typically consists of multiple cloud providers that may compete or collaborate [38] in order to gain higher revenue from service providers. Thus, it remains a challenge to generalize the service placement framework to consider complex relationships between multiple cloud providers. Furthermore, It would be interesting to the extend the framework to consider more general service topologies (e.g. general graphs) and analyze optimal control policies for such topologies in realistic scenarios. Furthermore, it would also be interesting to investigate alternative pricing models, such as auction based pricing schemes, for providing efficient resource allocation in geographically distributed clouds.

Heterogeneity-Aware Dynamic Capacity Provisioning

So far in Harmony we have only considered dynamic capacity provisioning of compute resources (e.g. VMs) and ignored other types of resources, such as the file storage system. While this is reasonable for traditional VM-based cloud environments, many modern data centers today employ distributed file system (e.g. Hadoop Distributed File System [2]) that distribute file blocks across a large number of machines. In this case, if a machine is turned off, the files stored on the machine can become unavailable. Thus, finding a solution to the dynamic capacity provisioning problem that considers both compute resources and storage resources is still an open challenge. Furthermore, in Harmony we assumed that each task has identical running time on two different types of machines, if same amount of resources is allocated on each machine. In reality this may not be true since certain types of machines may execute certain types of tasks faster than others [74]. Leveraging such fine-grained information for dynamic capacity provisioning is still an unsolved problem.

Phase-Level Scheduling for MapReduce

Even though PRISM clearly demonstrates the benefit of phase-level resource scheduling for MapReduce systems, there are many issues to be addressed in order to make phase-level scheduling effective in practical settings. First, the resource usage of the shuffle phase is dependent on the shuffle scheduler used. PRISM can gain benefit from a resource-aware shuffle scheduler that can optimize the CPU, memory and disk usage in the shuffle phase. However, Despite existing work on designing efficient shuffle schedulers (e.g. Orchestra [44]), none of the existing schedulers has considered the CPU and memory usage while making scheduling decisions. Secondly, the task run-time usage is sensitive to the job configuration parameters. In PRISM we assumed that job configuration is given by the user. Thus, designing a framework that dynamically controls job configuration parameters at run-time still poses interesting questions for future research.

Bibliography

[1] Hadoop MapReduce distribution. http://hadoop.apache.org.

[2] Hadoop Distributed File Systems, http://hadoop.apache.org/docs/hdfs/current/.

[3] GridMix benchmark for Hadoop clusters. http://hadoop.apache.org/docs/mapreduce /current/gridmix.html.

[4] PUMA Benchmarks, http://web.ics.purdue.edu/ fahmad/benchmarks/datasets.htm.

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

[6] Amazon elastic computing cloud (amazon ec2). aws.amazon.com/ec2/.

[7] Amazon elastic mapreduce (amazon emr). http://aws.amazon.com/elasticmapreduce/.

[8] Apache hadoop. http://hadoop.apache.org/.

[9] Energy star computer server qualified product list - june 1, 2012. http://www.energystar.gov/ia/products/prod_lists-/enterprise_servers_prod_list.xls.

[10] Energy star computers specification - feb. 14, 2012. http://www.energystar.gov/ia/partners/prod_development/revisions/downloads/computer/ES_Computers-_Draft_1_Version_6.0_Specification.pdf.

[11] Eucalyptus community. http://open.eucalyptus.com/.

[12] Googleclusterdata - traces of google workloads. http://code.google.com/p/googleclusterdata/.

[13] Gtitm homepage. http://www.cc.gatech.edu/projects/gtitm/.

[14] Hadoop fair scheduler. hadoop.apache.org/docs/r1.1.1/fair_scheduler.html.

[15] James hamiltons blog. http://perspectives.mvdirona.com/.

[16] The next generation of apache hadoop mapreduce.

[17] Open cloud computing, managed hosting, dedicated server hosting by racksapce. www.rackspace.com/.

[18] Technology research - Gartner Inc. www.gartner.com.

[19] World cup 1998 web site requests. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[20] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy proportional datacenter networks. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 338–347. ACM, 2010.

[21] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–16. USENIX Association, 2010.

[22] M. Armbrust, A. Fox, R. Griffith, A. 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. *Technical Report*, 2009.

[23] D. Arora, M. Bienkowski, A. Feldmann, G. Schaffrath, and S. Schmid. Online strategies for intra and inter provider service migration in virtual networks. *Arxiv preprint arXiv:1103.0966*, 2011.

[24] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM*, 2011.

[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[26] L. A. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[27] T. Başar and G. Olsder. *Dynamic noncooperative game theory*, volume 23. Society for Industrial Mathematics, 1999.

[28] C. Bash, C. Patel, and R. Sharma. Dynamic thermal management of air cooled data centers. In *IEEE Intersociety Conference on The Thermal and Thermomechanical Phenomena in Electronics Systems (ITHERM)*, 2006.

[29] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 8. ACM, 2011.

[30] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE, 2007.

[31] P. Bodik and et. al. Statistical machine learning makes automatic control practical for internet datacenters. *Proc. of USENIX HotCloud*, 2009.

[32] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 241–252. ACM, 2010.

[33] R. Boutaba, L. Cheng, and Q. Zhang. On cloud computational models and the heterogeneity challenge. *J. Internet Services and App*, 2012.

[34] R. Boutaba, L. Cheng, and Q. Zhang. On cloud computational models and the heterogeneity challenge. *Journal of Internet Services and Applications*, pages 1–10, 2012.

[35] G. Box, G. Jenkins, and G. Reinsel. *Time series analysis*. Holden-day San Francisco, 1970.

[36] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis, Forecasting, and Control*. Prentice-Hall, third edition, 1994.

[37] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge Univ Pr, 2004.

[38] R. Buyya, R. Ranjan, and R. N. Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *Algorithms and architectures for parallel processing*, pages 13–31. Springer, 2010.

[39] C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 185–194. Society for Industrial and Applied Mathematics, 1999.

[40] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. Energy-aware server provisioning and load dispatching for connection-intensive Internet services. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[41] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz. Analysis and lessons from a publicly available Google cluster trace. *Tech. Rep. UCB/EECS-2010-95*, June 2010.

[42] Y. Chen, R. Katz, and J. Kubiatowicz. Dynamic replica placement for scalable content delivery. *Proc. of IPTPS*, 2002.

[43] M. Chowdhury, M. R. Rahman, and R. Boutaba. Vineyard: Virtual network embedding algorithms with coordinated node and link mapping. *IEEE/ACM Transactions on Networking (TON)*, 20(1):206–219, 2012.

[44] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM*, 2011.

[45] K. Church, A. Greenberg, and J. Hamilton. On delivering embarrassingly distributed cloud services. *ACM HotNets*, 2008.

[46] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[47] J. Diaz et al. A guide to concentration bounds. In *Handbook on Randomized Computing*, 2001.

[48] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder. Temperature management in data centers: Why some (might) like it hot. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 163–174. ACM, 2012.

[49] C. Frank and K. Römer. Distributed facility location algorithms for flexible configuration of wireless sensor networks. *Distributed Computing in Sensor Systems*, pages 124–141, 2007.

[50] Y. Fu, C. Lu, and H. Wang. Robust control-theoretic thermal balancing for server clusters. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010.

[51] P. X. Gao, A. R. Curtis, B. Wong, and S. Keshav. It's not easy being green. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 211–222. ACM, 2012.

[52] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[53] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 24–24. USENIX Association, 2011.

[54] Í. Goiri, K. Le, J. Guitart, J. Torres, and R. Bianchini. Intelligent placement of datacenters for internet services. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 131–142. IEEE, 2011.

[55] P. Goyal, H. M. Vin, and H. Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *ACM SIGCOMM Computer Communication Review*, volume 26, pages 157–168. ACM, 1996.

[56] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM Computer Communication Review*, 39(1):68–73, 2008.

[57] D. Gross and C. Harris. Fundamentals of queueing theory. *ISBN: 0-471-17083-6*, pages 244–247, 1998.

[58] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 649–657. Society for Industrial and Applied Mathematics, 1998.

[59] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the Conext*, page 15. ACM, 2010.

[60] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of*

the 7th USENIX conference on Networked systems design and implementation, pages 17–17. USENIX Association, 2010.

[61] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proc. of the 5th Biennial Conf. on Innovative Data Systems Research (CIDR11), Asilomar, California, USA*, 2011.

[62] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22. USENIX Association, 2011.

[63] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, number November. Citeseer, 2009.

[64] S. Islam and J. Grégoire. Network edge intelligence for the emerging next-generation internet. *Future Internet*, 2(4):603–623, 2010.

[65] K. Jain and V. V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *Journal of the ACM (JACM)*, 48(2):274–296, 2001.

[66] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM, 2012 Proceedings IEEE*, pages 1206–1214. IEEE, 2012.

[67] G. Jung, M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 62–73. IEEE, 2010.

[68] M. Karlsson and C. Karamanolis. Choosing replica placement heuristics for wide-area systems. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 350–359. IEEE, 2004.

[69] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 94–103. IEEE, 2010.

[70] J. Kingman. The single server queue in heavy traffic. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 57, pages 902–904. Cambridge Univ Press, 1961.

[71] J. Krarup and P. M. Pruzan. The simple plant location problem: Survey and synthesis. *European Journal of Operational Research*, 12(1):36–81, 1983.

[72] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. *Cluster Computing*, 2009.

[73] N. Laoutaris and et. al. Distributed placement of service facilities in large-scale networks. *Proc. of IEEE INFOCOM*, 2007.

[74] G. Lee, B.-G. Chun, and R. H. Katz. Heterogeneity-aware resource allocation and scheduling in the cloud. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud*, volume 11, 2011.

[75] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubrahmanian, K. Talwar, L. Uyeda, and U. Wieder. Validating heuristics for virtual machines consolidation. *Microsoft Research, MSR-TR-2011-9*, 2011.

[76] H. E. Leland. Regulation of natural monopolies and the fair rate of return. *The Bell Journal of Economics and Management Science*, pages 3–15, 1974.

[77] M. Lin, A. Wierman, L. L. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. In *INFOCOM, 2011 Proceedings IEEE*, pages 1098–1106. IEEE, 2011.

[78] M. Lin, L. Zhang, A. Wierman, and J. Tan. Joint optimization of overlapping phases in mapreduce. *Technical Report*.

[79] Z. Liu, M. Lin, A. Wierman, S. Low, and L. Andrew. Greening geographical load balancing. In *SIGMETRICS*, 2011.

[80] R. F. Love, J. J. Morris, and G. O. Wesolowsky. *Facilities location*, volume 90. North-Holland Amsterdam, 1988.

[81] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012.

[82] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: insights from Google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37, March 2010.

[83] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making scheduling" cool": temperature-aware workload placement in data centers. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5, 2005.

[84] T. Moscibroda and R. Wattenhofer. Facility location: distributed approximation. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 108–117. ACM, 2005.

[85] M. Nelson, B.-H. Lim, G. Hutchins, et al. Fast transparent migration for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, 2005.

[86] D. Niu, C. Feng, and B. Li. Pricing cloud bandwidth reservations under demand uncertainty. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 151–162. ACM, 2012.

[87] D. Oppenheimer, B. Chun, D. Patterson, and A. Snoeren. Service placement in a shared wide-area platform. *in USENIX*, 2006.

[88] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *Cloud Computing*, pages 115–131. Springer, 2010.

[89] A.-M. Pathan and R. Buyya. A taxonomy and survey of content delivery networks. *Technical Report, University of Melbourne, Australia*, 2006.

[90] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé. Resource-aware adaptive scheduling for mapreduce clusters. *Middleware 2011*, pages 187–207, 2011.

[91] F. Presti, N. Bartolini, and C. Petrioli. Dynamic replica placement and user request redirection in content delivery networks. *IEEE International Conference on Communications (ICC)*, 2005.

[92] L. Qiu, V. Padmandabhan, and V. Geoffrey. On the placement of web server replicas. *IEEE INFOCOM*, 2001.

[93] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the electric bill for Internet-scale systems. In *ACM SIGCOMM Computer Communication Review*, volume 39, 2009.

[94] P. Radoslavov, R. Govindan, and D. Estrin. Topology-informed internet replica placement. *Computer Communications*, 25(4):384–392, 2002.

[95] L. Rao, X. Liu, and W. Liu. Minimizing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[96] S. Ratnasamy, M. Handley, R. Karp, and S. Scott. Topologically-aware overlay construction and server selection. *IEEE INFOCOM*, 2002.

[97] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Comp.*, 2012.

[98] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. Technical Report ISTC-CC-TR-12-101, April 2012.

[99] S. Ren et al. Provably-efficient job scheduling for energy and fairness in geographically distributed data centers. In *ICDCS 2012*.

[100] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the* 2^{nd} *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[101] B. Sharma, R. Prabhakar, S. Lim, M. Kandemir, and C. Das. Mrorchestrator: A fine-grained resource orchestration framework for hadoop mapreduce. Technical report, 2012.

[102] J. M. Smith. A survey of process migration mechanisms. *ACM SIGOPS Operating Systems Review*, 22(3):28–40, 1988.

[103] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Transactions on Networking (TON)*, 2009.

[104] X. Tang and J. Xu. Qos-aware replica placement for content distribution. *Parallel and Distributed Systems, IEEE Transactions on*, 16(10):921–932, 2005.

[105] F. Tian and K. Chen. Towards optimal resource provisioning for running mapreduce programs in public clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 155–162. IEEE, 2011.

[106] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1):1, 2008.

[107] V. V. Vazirani. *Approximation algorithms*. springer, 2004.

[108] A. Verma, P. Ahuja, and A. Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 243–264. Springer-Verlag New York, Inc., 2008.

[109] A. Verma, P. Ahuja, and A. Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *ACM/IFIP/USENIX Middleware*, 2008.

[110] A. Verma, L. Cherkasova, and R. Campbell. Resource provisioning framework for mapreduce jobs with performance goals. *Middleware 2011*, pages 165–186, 2011.

[111] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244. ACM, 2011.

[112] A. Verma, G. Dasgupta, T. Nayak, P. De, and R. Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the conference on USENIX Annual technical conference*. USENIX Association, 2009.

[113] C. Vicari, C. Petrioli, and F. Presti. Dynamic replica placement and traffic redirection in content delivery networks. *Proc. of MASCOTS*, 2007.

[114] G. von Laszewski, L. Wang, A. Younge, and X. He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *IEEE International Conference on Cluster Computing and Workshops (CLUSTER)*, 2009.

[115] C. Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[116] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.

[117] W. Wang, B. Liang, and B. Li. On fairness-efficiency tradeoffs for multi-resource packet processing.

[118] P. Wendell, J. Jiang, M. Freedman, and J. Rexford. Donar: decentralized server selection for cloud services. In *in ACM SIGCOMM*, 2010.

[119] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 3–14. ACM, 2010.

[120] D. Xie, N. Ding, Y. Hu, and R. Kompella. The only constant is change: incorporating time-varying network reservations in data centers. *Proceedings of ACM SIGCOMM*, 2012.

[121] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, pages 1–14, 2008.

[122] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55, Apr*, pages 2009–55, 2009.

[123] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[124] M. Zaharia, B. Hindman, A. Konwinski, A. Ghodsi, A. D. Joesph, R. Katz, S. Shenker, and I. Stoica. The datacenter needs an operating system. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, pages 17–17. USENIX Association, 2011.

[125] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX*

conference on Operating systems design and implementation, pages 29–42. USENIX Association, 2008.

[126] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

[127] Q. Zhang, J. Hellerstein, and R. Boutaba. Characterizing task usage shapes in googles compute clusters. 2011.

[128] Q. Zhang, M. F. Zhani, Q. Zhu, S. Zhang, R. Boutaba, and J. L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *Proceedings of the IEEE/ACM International Conference on Autonomic Computing (ICAC)*, 2012.

[129] Q. Zhang, Q. Zhu, and R. Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In *4th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pages 178–185. IEEE, 2011.

[130] Q. Zhang, Q. Zhu, M. F. Zhani, and R. Boutaba. Dynamic service placement in geographically distributed clouds. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, 2012.

[131] M. F. Zhani, Q. Zhang, G. Simon, and R. Boutaba. Vdc planner: Dynamic migration-aware virtual data center embedding for clouds.

[132] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.

[133] J. Zhou, X. Zhang, L. Bhuyan, and B. Liu. Clustered k-center: Effective replica placement in peer-to-peer systems. *IEEE Globecom*, 2007.

Appendix

This section contains the proofs of Lemma 5 and Theorem 4 and 6 in Chapter 4.

Proof of Theorem 4. : Define $\bar{N} = N \cup \{s^{jr}\}$. Since

$$\begin{aligned} & \Pr(\exists r : \sum_{j \in N} s^{jr} + s^{ir} > C^{mr} | \sum_{j \in N} c^{jr} + c^{ir} \leq C^{mr}) \\ = \; & \Pr(\exists r : \sum_{j \in \bar{N}} s^{jr} > C^{mr} | \sum_{j \in \bar{N}} c^{jr} \leq C^{mr}) \\ \leq \; & \Pr(\exists r : \sum_{j \in \bar{N}} s^{jr} > \sum_{j \in \bar{N}} c^{jr}), \end{aligned}$$

given M^k machines that have containers available, if we can ensure that the probability of violating machine capacity constraint is less than $\epsilon^{\frac{1}{M^k}}$ (i.e., $\Pr(\exists r : \sum_{j \in \bar{N}} s^{jr} > \sum_{j \in \bar{N}} c^{jr}) \leq \epsilon^{\frac{1}{M^k}}$, then the inequality will hold. Furthermore, since

$$\Pr(\exists r : \sum_{j \in \bar{N}} s^{jr} > \sum_{j \in \bar{N}} c^{jr}) \;\; \leq \;\; \sum_{r \in R} \Pr(\sum_{j \in \bar{N}} s^{jr} > \sum_{j \in \bar{N}} c^{jr})$$

holds regardless of the resource correlations, if we can ensure that

$$\Pr(\sum_{i \in \bar{N}} s^{ir} \geq \sum_{i \in \bar{N}} c^{ir}) \;\; \leq \;\; \frac{1}{|R|} \epsilon^{\frac{1}{M^k}}$$

for all $r \in R$, then the bound will hold. Define $\epsilon^r = \frac{1}{|R|} \epsilon^{\frac{1}{M^k}}$. To achieve this objective, we use concentration inequalities [47]. Define $c^{ir} = \mu^{ir} + \beta^{ir}$, where β^{ir} is a variable to be determined for each $i \in \bar{N}$. We can rewrite our objective as to ensure

$$\Pr(\sum_{i \in \bar{N}} (s^{ir} - \mu^{ir}) \geq \sum_{i \in \bar{N}} \beta^{ir}) \;\; \leq \;\; \epsilon^r$$

The one-sided Chebyshev's inequality [47] states that

$$\Pr(\sum_{i\in\bar{N}}(s^{ir}-\mu^{ir})\geq\sum_{i\in\bar{N}}\beta^{ir})\leq\frac{\sum_{i\in\bar{N}}(\sigma^{ir})^2}{\sum_{i\in\bar{N}}(\sigma^{ir})^2+(\sum_{i\in\bar{N}}\beta^{ir})^2}$$

Thus it suffices to ensure the following inequality holds:

$$\frac{\sum_{i\in\bar{N}}(\sigma^{ir})^2}{\sum_{i\in\bar{N}}(\sigma^{ir})^2+(\sum_{i\in\bar{N}}\beta^{ir})^2} \quad \leq \quad \epsilon^r$$

Rearranging the equation and using the fact that $\sum_{i\in\bar{N}}(\sigma^{ir})^2\leq(\sum_{i\in\bar{N}}\sigma^{ir})^2$, we obtain

$$\sum_{i\in\bar{N}}\beta^{ir} \quad \geq \quad \sqrt{\frac{1-\epsilon^r}{\epsilon^r}}\sum_{i\in\bar{N}}\sigma^{ir}$$

Thus equation (4.16) holds by setting $\beta^{ir}=\sqrt{\frac{1-\epsilon^r}{\epsilon^r}}\sigma^{ir}$ for each task $i\in\bar{N}$. The result follows. □

Proof of Lemma 5. . We rely on the property that the First-Fit (FF) algorithm produces a solution in which at most one machine i is less than "half-full" (i.e., utilization $u_t^{ir}\leq\frac{1}{2}\forall r\in R$). To see this, suppose this statement is false, i.e., there are two non-empty $i,j\in N_t^m$ that are less than "half-full" and i is filled before j. In this case, when FF tries to pack a container that belongs to j in the solution, it would pack it in i instead. As a result, machine j should hold no containers, which contradicts our assumption. Therefore, given a machine i with utilization u_t^{ir} for resource type $r\in R$, define the *effective utilization* of i as $\frac{1}{|R|}\sum_{r\in R}u_t^{ir}$. Based on this "half-full" property, FF ensures every machine has effective utilization at least $\frac{1}{2|R|}$ except the last non-empty machine.

Given $x_t^{mk^*}$ type n containers for each $n\in N$ that can be scheduled on z_t^{m*} type m machines, the sum of the total effective utilization must be less than z_t^{m*} as it is the maximum possible utilization for z_t^{m*} machines. Now, suppose we scale down the number of type n containers to $\left\lfloor\frac{x_t^{mk*}}{2|R|}\right\rfloor$ for each $n\in N$, the total utilization of machines is thus at most $\frac{z_t^{m*}}{2|R|}$. Suppose there are still containers waiting to be scheduled after using $z_t^{m*}+1$ machines. As FF ensures every machine has effective utilization at least $\frac{1}{2|R|}$ except the last one, the total utilization of the $z_t^{m*}+1$ machines is at least $\frac{z_t^{m^*}}{2|R|}$, which contradicts that the total utilization is at most $\frac{z_t^{m^*}}{2|R|}$. □

Proof of Theorem 6. . Since the number of machines used is determined by $DCP - RELAX$, it is clear that the $C_t^{sw} = C_t^{sw*}$ and $E_t^{idle} = E_t^{idle*}$. As the number of type n containers scheduled on type m machines is upper-bounded by x_t^{mk*}, we have $E_t^{util} \leq E_t^{util*}$. Finally, by Lemma 1, it is easy to show that $\left\lfloor \frac{x_t^{mk*}}{2|R|} \cdot \frac{z_t^{m*}-1}{z_t^{m*}} \right\rfloor$ containers of each type $n \in N$ can be packed in z_t^{m*} machines. As $f(\cdot)$ is a convex function, it must hold that $U_t^{pref} \geq (\max_m\{\frac{z_t^{m*}-1}{z_t^{m*}}\} - \epsilon') \cdot \frac{1}{2|R|} U_t^{pref*}\cdot$, where $\epsilon' = \max_m\{\frac{x_t^{mk*}}{2|R|} \cdot \frac{z_t^m-1}{z_t^m} - \left\lfloor \frac{x_t^{mk*}}{2|R|} \cdot \frac{z_t^m-1}{z_t^m} \right\rfloor\}$ is the rounding error. The theorem is proven by defining $\epsilon = \max_m\{\frac{1}{z_t^m}\} + \epsilon'$ and summing the above equations. □