Cloud Auto-Scaling with Deadline and Budget Constraints

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Ming Mao

Dec 2012

Abstract

The cloud has become an important computing platform. It has attracted many businesses and individual users by offering on-demand computing power and storage capacity. The economies of scale and pay-as-you-go billing model could save users large up-front capital investments and long term operation costs. A key feature of the cloud is the elasticity, the ability to dynamically acquire and release computing resources in response to demand. We believe the key to successful cloud adoption is to first decide how much and what type of resources is needed in the cloud ("provisioning") and then decide how to place computing activities onto each of the resources ("allocation"). This is a challenging problem because the mapping from user objectives to the resource provisioning and allocation plans is not trivial. It needs to carefully consider the following factors. A performance goal can be achieved through different types of resources with different costs. A fixed budget can be used to rent a wide variety of resource configurations for varying durations. The structure of a cloud application could be complex. Task precedence orders need to be preserved in a job. The workload may experience unexpected peaks. The performance requirements and cost constraints may be changed dynamically.

This dissertation solves this resource provisioning and allocation problem using an auto-scaling approach. It solves the batch-queue application model based on the integer programming technique. By ensuring the computing power is always large enough to handle the workload for all the VM types, in our experiment, our approach finishes more than 95% jobs before the deadline and saves 20.2% - 40.1% cost compared to a fixed machine type choice. Our approach contains several innovative heuristics for the workflow application model. In the unlimited budget case, the presented solution - dynamic scaling-consolidation-scheduling (SCS) - can save 9.4% - 40.4% cost compared to two baseline approaches and can work well in both light and heavy workload environments. In the limited budget case, our scheduling-first and scaling-first algorithms can reduce 9.8% - 45.2% job turnaround time than the standard machine choice and they also show good tolerance (between -10.2% and 16.7%) to inaccurate parameters ($\pm$20% error). Finally, this dissertation presents three job scheduling policies and a data prefetching strategy to manage the intermediate data for data-intensive applications running in the cloud. Particularly, the cost-deadline-first (CDF) algorithm can save 13.5% - 33.7% cost compared to the deadline-first (DF) algorithm and the data prefetching strategy can further improve the cost saving up to 44.6% through data locality aware job placement.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Ming Mao

This dissertation has been read and approved by the Examining Committee:

Marty Humphrey, Adviser

Jack Davidson, Committee Chair

Sudhanva Gurumurthi

Jason Lawrence

Teresa Culver, Minor Representative

Accepted for the School of Engineering and Applied Science:

James H. Aylor , Dean, School of Engineering and Applied Science

Dec 2012

To everyone who has helped me succeed

Acknowledgments

First and foremost, I want to thank my adviser Marty Humphrey. Without his guidance and support, I cannot be where I am today. During the past five and a half years, he has inspired me to find the most important and interesting problems, given me independence to cultivate my research capabilities, encouraged me during the difficult times and supported me with his resources. His patience and understanding can always make things clear and efficient. It is a great fortune in my life to work with him for the graduate study.

I want to thank my committee members Sudhanva Gurumurthi, Jack Davidson, Jason Lawrence and Teresa Culver. With their help and suggestions, the problem statement of this dissertation becomes clear and the differences with related work are carefully clarified, which helps me to think the problem from a different angle and at a higher level. I also want to thank all the other professors at the department, taking their courses and talking to them are always enlightening and enjoyable. I will always remember this great five years at this department.

I am so lucky to have so many friends to support me all the time. They share both the joys and tears with me. They help me out of troubles, bring me happiness and inspire me with great ideas. They are Zach Hill, Sang-min Park, Arkaitz Ruiz-Alvarez, Jie Li, Joel Coffman, Luther Tychonievich, Chih-hao Shen, Karolina Sarnowska-Upton, Hongjian Zhu, Minghong Yu and so many friends that I cannot list them all.

I am indebted to my parents. They support and love me in any conditions. Their love means so much to me, and their encouragement to persevere during the difficult times has helped to keep me on track. Though they are far from Charlottesville in the past five years, their voices and smiles are always the closest to my heart.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The cloud, defined as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" by NITS [1], has attracted many businesses and individual users. It has become an important computing platform.

The key benefit of the cloud is the ability to dynamically rent seemingly unlimited resources in response to demand. We refer to this action as *provisioning*. To fully take advantage of this benefit, cloud users need to determine the "right" size of the resources to provision. A separate issue is how to place computing activities onto these rented computing and storage resources. We refer to this decision as *allocation*. To provision cloud resources and allocate the acquired resources for sophisticated computing activities is not a trivial problem. The best approach is dependent on several important factors, e.g. the goals and constraints in the resource provisioning process, the types of resources that are available, the types of applications and workload that need to be supported, etc. This dissertation argues an automatic resource provisioning and allocation mechanism can help cloud users to achieve the application goals and use the cloud resources cost-efficiently, i.e. an auto-scaling solution.

The remainder of this chapter sets up the context for this dissertation. The background section explains the cloud's momentum. More importantly, it analyzes the two most important benefits and concerns in the cloud adoption - dynamic scalability and cost saving - which are the focuses of this dissertation. It also details the Infrastructure-as-a-Service (IaaS, in which the cloud offers the most basic and fundamental computing resources, e.g. virtual machines) service model which defines the types of the cloud resources that are used in our auto-scaling solutions. The problem statement section presents a few motivating examples. They describe two application architectures supported in this dissertation, i.e. batch-queue (users submit their independent jobs into a single queue for batch processing) based applications and Service Oriented Architecture (SOA) based applications, and show that maximizing application performance and minimizing the cost are important goals for service providers. Based on the motivating examples, the section introduces the generalized three-layer cloud application model and defines the auto-scaling problem. Finally, this chapter summarizes the contributions of this dissertation and concludes with an overview of the remaining chapters.

1.1 Background

1.1.1 Cloud

Based on the IDC [2][3][4] and Gartner [5] reports, the cloud will maintain its momentum, and dramatically change the way people provision, maintain and use computing resources. The cloud market are generally divided as public cloud sector and private cloud sector. A public cloud is cloud applications, storage, and other resources that are delivered to the general public over the Internet, such as Amazon EC2 [6]. A private cloud is the cloud infrastructure operated solely for a single organization, whether managed/hosted within the enterprise or externally. Here are some important predictions from the two research organizations.

- Spending on public cloud IT services in 2011 was $28 billion, compared with more than $1.7 trillion in spending on total IT products and services (IDC).
- 80% of new commercial enterprise applications will be deployed on cloud platforms in 2012 (IDC).
- Amazon Web Services [will] exceed $1 billion in cloud services business in 2012 with Google's Enterprise business to follow within 18 months (IDC).
- At year-end 2016, more than 50 percent of Global 1000 companies will have stored customer-sensitive data in the public cloud (Gartner).
- IT cloud services helped organizations of all sizes and all vertical sectors around the world generate more than $400 billion in revenue and 1.5 million new jobs. In the next four years, the number of new jobs will surpass 8.8 million (IDC).
- Analysts report that both the public and private cloud markets continue to grow, with the public cloud market reaching $100 billion by 2016 (IDC).

With higher stability and more complete service offers, the cloud has become a more and more mature computing platform. It gains confidence and popularity from the IT industry and it is becoming a trend to deploy services in the cloud [7]. More organizations will start to move their IT infrastructure from on-premise to off-premise.

1.1.2 Cloud Benefits and Concerns

The cloud has attracted so many users because of its advantages. The key characters and benefits of the cloud include “on-demand self-service”, “broad network access”, “resource pooling”, “rapid elasticity” and “measured service” [1]. Cloud users can provision compute and storage resources in response to demand dynamically. They could perform the resource acquisition and release automatically without any human intervention. The cloud resources can be accessed by different types of clients (e.g. desktop, tablets and mobile devices) through standard mechanisms over the Internet. Cloud services are charged based on the usage. The cost of VM instances, network and storage are precisely defined and measured. Cloud users only pay for what they use. The cost of the compute resources is reduced because of the economies of scale. Compared to the in-house computing infrastructure, capital expenditure is converted to operational expenditure and a large amount of up-front capital investments can be saved. Maintenance process becomes simpler and the maintenance cost is also reduced.

At the same time, cloud users also express their concerns in this cloud movement [4][8]. The top concerns include cost management, security, service availability, unstable application performance, interoperability and standard APIs. Cloud users need to watch their cloud spending closely and keep it within their budget. They are afraid that their sensitive data may be leaked to the cloud providers or other parties sharing the same physical resources. They need cloud providers to offer the services as stated in the service level agreements. They want standard APIs and interoperability to deploy their applications in a multi-cloud environment and avoid data lock-in.

Among all the benefits and concerns, we believe that the two most important factors are dynamic scalability and cost saving. They are the keys to the successful cloud adoption. Figure 1.1 is a survey from 39 major technology companies [8]. Their answers confirmed this statement.

Dynamic Scalability. One of the main advantages of the cloud is the rapid elasticity, the ability to dynamically acquire and release cloud resources in response to demand. It is the key enabler of the cloud adoption. This elasticity saves the cloud users large up-front capital investments and allows the computing resources to grow accordingly with business demand, which becomes one of the main forces that drive more and more applications to be migrated and developed in the cloud. There are three compelling use cases to demonstrate the advantages of dynamic scalability [9]. A first use case is when the workload varies with time. To handle the peak workload, an application needs to maintain the computing capability at maximum

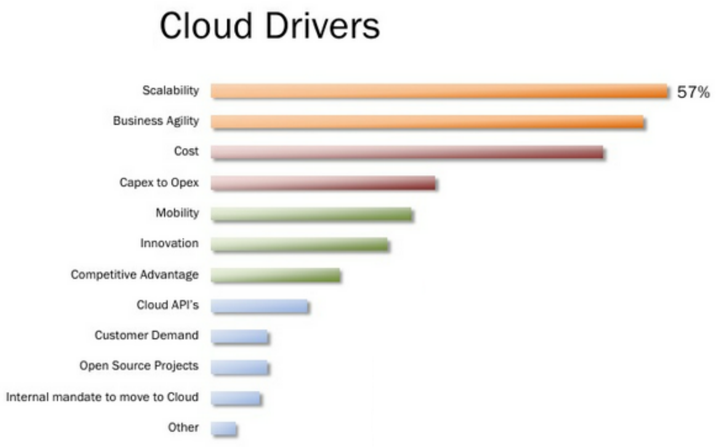

Figure 1.1: Cloud adoption drivers [8]

level all the time, which will lead to significant resource underutilization at other times. The shorter the duration of the peak load and the more significant the difference between the peak and normal load, the more the waste. The second use case is when the demand is not known in advance. For example, a startup company may gain its momentum and needs to support a spike in demand when the service becomes popular. However the demand may reduce when some users turn away. The last use case is called free speedup, which is especially true for processing batch jobs. For example, using 1,000 EC2 machines for one hour costs the same as using one machine for 1,000 hours. Cloud users can greatly speed up the job execution with the same cost.

Though the cloud offers dynamic scalability and it is the major advantage, this does not mean that cloud users can use this feature in a correct and easy way. Though the computing infrastructure is outsourced to the cloud, essentially the cloud users still need to determine the size of the resource pool by themselves. Running their applications in the cloud does not shift away the resource ill-sizing problem as in the in-house infrastructure environment. If they cannot correctly provision the resources needed, application performance will be affected and customers are turned away. The potential cost saving benefits of using the cloud can also disappear.

Cost. Another important factor when considering cloud adoption is the monetary cost. Maximizing the return and minimizing the cost from cloud investment are the two main goals for the cloud users. The main driving force behind cloud movement is cost saving, which results from the economies of scale and dynamic scalability (in fact dynamic scalability is a way to realize cost saving). For the same business scenarios, if it is more expensive to develop and maintain the applications in the cloud, the cloud loses its advantages.

Therefore, cost saving is a first-class concern for the cloud users. In addition to being a goal, sometimes cost could become a constraint for some cloud applications. For example, some service providers may have a budget limit that is allowed to spend on cloud purchasing. The running cost of the acquired resources cannot exceed a certain number. In such cases, cost essentially determines the maximum size of the resource pool. Finally, cloud has its own billing models. Though the concept of utility computing was first introduced in 1960s [10], only in the cloud, the price of all the computing resources can be precisely defined and measured for the first time. Every job processed and every plan executed is essentially associated with a number that will appear on the users' billing statements. Therefore, this dissertation argues that the cost should be carefully modeled and considered when using the cloud.

In summary, the key benefit of the cloud is to provision resources in response to demand dynamically and only pay for the resources used. This benefit can only be realized when the cloud users can determine the right size of the resource pool and allocate the resources in a cost-efficient way. While resource over-provisioning costs users more than necessary, which essentially offsets the cloud advantages, resource under-provisioning hurts the application performance, which could violate the service level agreements and turn away customers.

1.1.3 Service Models

The cloud offers different services. Depending on the type of the services and resources delivered over the Internet, the public cloud can be categorized as follows. Particularly, the Infrastructure-as-a-Service (IaaS) cloud defines the resource types that are used in this dissertation.

Infrastructure-as-a-Service (IaaS) The cloud providers offer the consumers computers, storage, network, firewalls, IP addresses and other fundamental computing resources on demand. Computers are normally provisioned as Virtual Machines (VMs) in the cloud providers' multi-tenant data centers. The cloud users can therefore control the operating systems and the whole software stack installed on the machine. The IaaS cloud providers include Amazon EC2[6], Rackspace[11], Google Compute Engine[12] and Windows Azure[13], etc.

Platform-as-a-Service (PaaS) The cloud providers offer a computing platform or a solution stack which normally includes operating system, runtime execution environment, web servers and databases for users to deploy their applications. The cloud users do not need to manage the underlying hardware and software environment. The PaaS cloud providers include Windows Azure[13], Google App Engine[14], etc.

Software-as-a-Service (SaaS) The cloud providers offer consumers the software and associated data over the Internet in a on-demand way. The cloud consumers do not manage and control the underlying the infrastructure, operating systems, network or storage that the software run on. The SaaS cloud providers include Saleforce[15], Google Apps[16], Microsoft Office365[17], etc.

X-as-a-Service (XaaS) As the service providers get more specialized and focused in a particular area, the categorization becomes even more fine-grained. Recently, there are more emerging public markets X-as-a-Service, such as Storage-as-a-Service (STaaS), Security-as-a-Service (SECaaS), Data-as-a-Service (DaaS), Test environment-as-a-Service (TEaaS), Desktop as-a-Service (DaaS), API-as-a-Service (APIaaS) etc. [18].

IaaS cloud defines the types of resources that are used in our auto-scaling solution. This dissertation chooses IaaS because it offers the most general resource type and the OS-level virtualization provides users the most power and flexibility compared to other service types [19]. It allows users to configure the operating systems settings, install customized kernel modules, mount file systems and customize required binaries and services. The OS images can be can snapshotted, copied and saved in the users' cloud account for later use. The saved OS images serve as the templates for the scalable services and components. In addition, cloud users can choose appropriate VM types (CPU, memory, disk space, etc.) based on the application needs. For example, Amazon EC2 [6] currently offers 14 types of VM instances. Each instance provides a predictable amount of dedicated compute capacity. Different VM types may provide different CPU power, memory, disk storage space and I/O performance. Therefore, a task may prefer one instance type over another depending on different performance metrics, such as execution time, cost, cost-efficiency, etc. One notable fact is that the VMs are not necessarily priced linearly to their performance. In other words, for a particular task, a more expensive machine does not imply a faster machine [20]. Moreover, the VM instances are charged based on the per time quantum billing model. The current prevailing time quantum granularity is one hour. Therefore, the partial instance hours are always billed as full hours. A machine costs the same for running 1 minute or 1 hour. Ignoring the partial instance hours could waste a large portion of purchased computing power [21].

This dissertation focuses on the resource consumer side - cloud users. It does not tackle the problem of building the cloud, but the problem of using the cloud. It answers the resource provisioning and allocation questions for the cloud consumers. Particularly, it targets the applications running on the IaaS cloud. It aims to offer cost-efficient resource auto-scaling solutions to the cloud users who build their applications and services using the cloud VMs, network and storage, which are low level infrastructure services (IaaS cloud).

1.2 Problem Description

1.2.1 Motivating Examples

Three motivating examples are presented in this section to illustrate this resource provisioning and allocation problem. While these examples show the resource management challenges when using the cloud resources, they also show different application architectures, workload types, service goals and constraints that are required to be supported in this dissertation.

Case 1). MODISAzure [22] is an eScience cloud application that has been running in Windows Azure cloud [13]. The MODIS data is generated by the Terra and Aqua satellites and is a viewing of the entire Earth's surface in 36 spectral bands, at multiple spatial resolutions, generated every 1-2 days. MODIS provides various biophysical variables (e.g. gross carbon uptake, albedo etc.) with spectral irradiance ranging visible, near infrared, infrared and thermal regions of the electromagnetic spectrum. It has been an important scientific source and can be applied to many environmental studies from local to global scale. For each request (job) submitted by the domain scientists, MODISAzure downloads the raw satellite images from data sources and reprojects the source images to target images. While the total number and size of the data processing requirements for the whole computation request can be substantial (e.g. compute reprojected data on US continent for a whole year), the computation requirement for a single task is relatively small, as the whole request can be separated into a large number of embarrassingly parallel small tasks (bags-of-tasks) [22]. In addition, these requests are associated with deadlines. These deadlines are not hard deadlines but indication of the users' preference on the job completion time. For example, the domain scientists may prefer all the reprojection jobs are finished before 8:00AM in the morning, so they can start to test different algorithms when they come back to work the next morning. However, though Windows Azure [13] allows users to dynamically acquire/release VMs, it is not trivial for the domain scientists to decide what types of resources are needed and for how long to finish their jobs before deadlines. It is certainly not their expertise, and more importantly, determining the resource size also becomes a new burden and a great barrier to the cloud adoption. If they cannot provision the resources in a cost-efficient way, their jobs may not complete in time and their progress are slowed down, or they could pay far more for the computation cost than the in-house computing infrastructure.

How/why does this example motivate this research - MODISAzure is a batch-queue based application. It accepts bags-of-tasks (embarrassingly parallel jobs) as the workload. The jobs are associated with deadlines. The domain scientists need an auto-scaling solution to help them determine the resource size dynamically. The auto-scaling solution should not only acquire right enough resources to finish all the jobs before deadlines but also acquire the resources in a cost-efficient way to minimize the application cost. This example confirms the resource management challenge in the cloud. Though the cloud application has already been successfully running, moving the computing infrastructure into the cloud does not shift away the resource ill-sizing problem and the problem can become quite complex when considering different goals and constraints.

Case 2). A large insurance company that offers many types of insurance products (e.g., automobile, home, business and health) is to move its application into the cloud to lower cost. The primary goal of the company is to increase its profits by minimizing risks and attracting/retaining more customers. In the insurance business, the premiums paid by customers are a function of the risk posed by the insurance policy. Traditionally, insurance companies use a combination of actuarial data with some minimal amount of personal data to establish the risk associated with a policy and thus its premium. If the risk assessment overestimate the risk, premiums increase and the insurance company may lose customers. In an effort to increase its profits, the insurance company is moving towards a highly customized risk assessment model. With this approach, a much larger number of information sources about a customer are queried to obtain a much richer set of inputs to the risk assessment model. The new model provides a risk rating specific to a given customer (e.g., John Doe who besides being a non-smoking male, straight A college student, under 25, with a very good credit record, undergoes a physical exam every year and is in perfect health, and has a clean record with federal, state, and local law enforcement agencies). Clearly, these customized models are much more sophisticated and significantly more compute- and data- intensive, and may take long to finish depending on the data set size. Potential customers may submit quote requests with different priorities - immediate, non-immediate and delayed. Immediate quote jobs must provide fast responses (1-8 seconds) to customers before they leave the website, while non-immediate jobs will use more complex risk assessment models and have less stringent response time requirement (10 seconds to 3 minutes). Delayed quote requests are even more compute- and data- intensive jobs because they use more queried personal data information and run the heaviest risk assessment model. It provides the most accurate results with the longest deadlines (30 minutes to 3 hours). In addition to these quote requests regarding customer premiums, the insurance company also needs to generate daily reports for management review and internal audit. The research department will regularly crunch a large volume of data to perform data mining and online analytical processing. The IT department also needs to back up data weekly. The whole cloud application is based on the service oriented architecture (SOA) and jobs can be described as workflows which go through multiple service components. Jobs may share the same service components but have different execution paths. At the early stages, the insurance company only allocates a fixed budget to the application, mainly to test the feasibility of cloud migration. The goal in such cases is to maximize the application performance within the budget constraints. When the cloud solution becomes mature and stable, all the query requests are routed to the cloud and the budget cap is removed. Therefore the goal is to minimize the cost while meeting the job deadline constraints. Considering the complexity of the workload (task dependencies and individual deadlines) and the limited/unlimited budget caps, resource provisioning and allocation becomes an even more challenging problem in this example.

How/why does this example motivate this research - Though this cloud application is a simplified hypothetical example, it however demonstrates another very important application architecture and explains the unlimited/limited budget scenarios. This application from the insurance company is based on the Service Oriented Architecture (SOA). It accepts jobs that have different execution flows among the service components, which can be treated as workflows [23]. A workflow consists of a sequence of connected steps. It is normally represented as a direct acyclic graph (DAG), e.g. a four step workflow in figure 1.2. In each workflow job, task precedence constraints need to be preserved. Such task dependency handling largely increases the complexity of the problem. The auto-scaling problem has two different goals depending on the available budget. In the unlimited budget case, the goal is to minimize cost while meeting job deadlines. In the limited budget case, the goal is to maximize the application performance within the budget constraints.

Case 3) Montage [24][25] was created by the NASA/IPAC Infrared Science Archive as an open source toolkit that can be used to generate custom mosaics of the sky using input images in the Flexible Image Transport System (FITS) format. During the production of the final mosaic, the geometry of the output is calculated from the geometry of the input images. The production process consists of several steps which can be considered as workflows. The number of inputs processed by the workflow may increase over time as more images of a particular region of the sky are available. As such, the structure of the workflow changes to accommodate the increase in the number of inputs, which also translates to an increase in the number of computational jobs. The inputs are then re-projected (another step) to be of the same spatial scale and rotation. The outputs are the reprojected image and an area image that consists of the fraction of the image that belongs in the final mosaic. Finally, the sky background emissions in the images are corrected to be of the same level in all images. One important feature of the Montage application is that there are large volume of intermediate files generated, transferred and stored during the application execution. It is a data-intensive application and this data-intensive feature largely increases the problem complexity. In addition to the resource provisioning and allocation decisions, the intermediate data should be explicitly considered and managed in a fine granularity because its transfer and storage plans could dominate the application' performance and cost. This intermediate data management requirement makes it even more challenging for the users to use cloud resources cost-efficiently.

How/why does this example motivate this research - Montage can also be considered as a SOA based application. The workload are workflow jobs. One important feature of the Montage application is that there could be a large amount of intermediate data transfer among the workflow steps, such as the half-processed images. The data transfer time could be longer than image processing time. The cost of transferring and storing the intermediate images could be more expensive than the computation cost. This requires the resource management mechanism explicitly and carefully consider the data transfer and storage plans. In this motivating example, the workflow jobs are associated with deadlines which indicate the preference on the job completion time. In addition to the resource provisioning and allocation decisions, it essentially requires an intermediate data management policy to help cloud users to finish their jobs before deadlines and minimize the application cost.

1.2.2 Problem Statement

The motivating examples in section 1.2.1 describe different applications that could benefit from cloud auto-scaling mechanisms to use the cloud resources in a cost-efficient way. However every application has its own requirements and features that need to be carefully considered (e.g. workload, application architectures, etc.). These requirements and features will help to shape the resource provisioning and allocation mechanisms in the cloud. They become the goals and constraints of the auto-scaling solutions. Before analyzing these important factors in detail, this dissertation first describes the three-layer application model which can be generalized from the three examples. This generalized model helps to clarify the assumptions and define the problem.

In the three-layer cloud application model, as shown in figure 1.2, there are three roles. They are Infrastructure-as-a-Service (IasS) cloud providers who offer different types of virtual machines (VMs) and cloud storage, such as Amazon AWS [6], Windows Azure [13] and Rackspace [11], service providers who build their value-added services using the cloud resources (e.g. the insurance company), and service customers (end users) who process their jobs through the services built on the cloud resources. The service providers purchase the cloud resources (VMs) from the cloud providers and serve the client requests submitted by the service customers. In this scenario, the cloud providers charge the service providers for the consumed resources and the service providers may or may not charge the service clients depending on their business goals. For some service providers, they charge the service clients for the services they provide. As long as the revenue of serving each client request is higher than the cloud cost, the service provider is making profits. Therefore, these service providers have an unlimited budget. Their goal of using cloud is to minimize the cost (hence maximize the profits) while meeting the application performance requirements. These service providers include Netflix [26], the insurance company example, etc. For some other service providers, they mainly use the cloud for internal computing services instead of making profits, such as the supporting IT team for a scientific research organization, and they normally have a budget cap when purchasing the cloud resources that is approved by the finance department every fiscal year or constrained by the available project funding. For such service providers, their goal is to get the fastest performance within the budget constraints to maximize the return of cloud investment. Many scientific applications fall into this category.

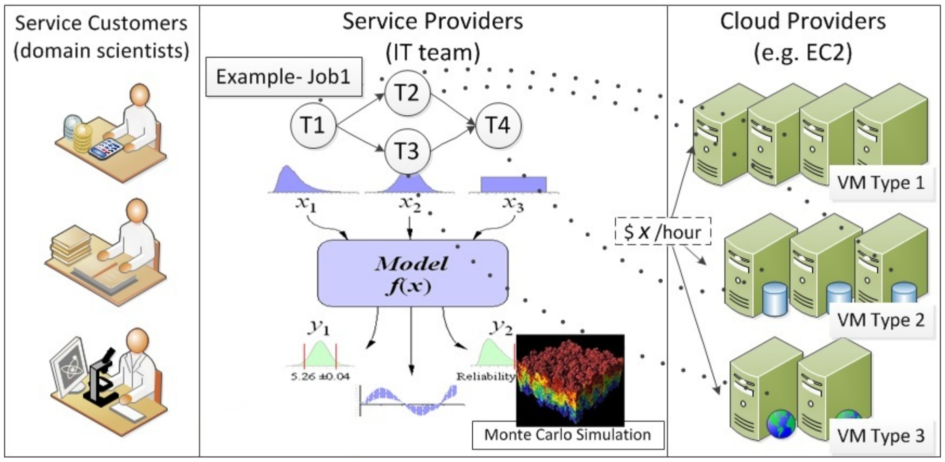

Figure 1.2: The three-layer cloud application

The three-layer model illustrates an important problem when using the cloud - what resources should be acquired/released in the cloud, and how should the computing activities be mapped to the cloud resources, so that the application goals can be met at the least financial cost? It is an important question for the service providers to fully take advantage of the dynamic scalability benefit. Currently, there are two options available. First, by monitoring the application workload and cloud resource utilization information (e.g. via a web browser based application status "dashboard"), the service providers manually acquire resources when the workload cannot be handled in time and release resources when the resources are underutilized. This is the easiest approach. However this approach is fragile and subject to human mistakes. It is not a systematic solution and cannot work in a large scale. Second, some cloud providers such as Amazon EC2 [6] offer trigger mechanisms that allow users to specify resource utilization indicators and thresholds to acquire and release the cloud resources automatically (e.g. acquire 1 small instance when the average CPU utilization rate is above 70% for more than 5 minutes). While such automation can help to simplify the resource provisioning process and avoid human mistakes, it is not straightforward to determine the "right" resource indicators and thresholds, especially when considering that the limited resource indicators available are low level and the application structures could be very complex. More importantly, these trigger mechanisms only help to determine the resource size but do not help to allocate the acquired resources. The cloud users still need to allocate the resources to the computing activities by themselves (more discussions about the trigger mechanisms can be found in the related work chapter). In this dissertation, we argue an auto-scaling mechanism in the cloud needs to answer the following three questions. (1) What types of resources need to be provisioned, how much and for how long. This is a resource provisioning or capacity determination problem (also referred as scaling). (2) When the acquired resources are available, how to allocate the resources to the workload. This is a resource allocation or job scheduling problem (also referred as scheduling). (3) In the data-intensive applications, how to store and transfer the data. This is an intermediate data management problem. When answering these three questions, an auto-scaling solution also needs to carefully consider the following factors - target, resource type, cost model, workload, application objectives and constraints. They are shared by all the subproblems as common assumptions.

Target The target of this dissertation is the cloud applications running on the IaaS cloud, i.e., the service providers described in the three-layer application model. The target is not some specific workload, e.g. a single job or a set of jobs. The application is continuously running and processing jobs submitted by the service customers.

Workload Service customers submit their jobs to the cloud application. The workload is dynamic. It may experience seasonal behaviors and unexpected peaks. For example, weekdays have higher workload than weekends. Business open hours have higher workload than business off hours.

Job Depending on the application architecture, jobs can be bags-of-tasks which have no dependencies on each other or workflow jobs in which task precedence constraints need to be preserved. Jobs are associated with deadlines and priorities. Deadline is not the hard deadline as in the real-time systems. Deadline misses are allowed. Deadline reflects the service customers' preference on the job completion time.

IaaS Cloud The applications are built on the IaaS cloud resources. Cloud VMs can be dynamically acquired by the application. Cloud VMs are with different configurations, e.g. CPU, memory, disk, etc. Therefore, the same task may have different execution times on different VMs. One interesting fact is that the VM price may not be linear to the performance. In other words, a more expensive machine does not imply a faster machine. The VMs are charged based on the cost per time quantum scheme. The current prevailing billing model is dollars per instance hour. Partial instance hours will be rounded up as full hours. Data transferred in and out are charged based on the data size.

Goals and Constraints Deadline and cost serve as either goals or constraints in the resource provisioning and allocation problem. The goal of an auto-scaling solution is to help the service providers to make resource provisioning decisions and allocate the acquired VMs to the workload within the deadline and budget constraints, such as minimizing the application cost or maximizing the application performance.

Output The output are the answers to the three questions, i.e. scaling plan, scheduling plan and data management plan. The scaling plan is to determine the number of instances for each VM type. The scheduling plan is to schedule each job on the acquired VMs. The data management plan is to determine the intermediate data placement and transfer among the cloud VMs and cloud storage.

Based on the motivating examples and the factors analyzed above, this dissertation tackles the cloud auto-scaling problem through four subproblems. They target different application structures with different types of workload, they treat deadlines and cost as either goals or constraints, and they have different assumptions on the performance and cost of the intermediate data. The subproblems are formally defined in chapters 3, 4, 5 and 6. The above assumptions will be revisited whenever necessary.

Problem (1): Auto-scaling for batch-queue based cloud applications. It targets batch-queue based applications that accept bags-of-tasks as the workload. All the jobs have the same job turnaround time deadline. The goal is to determine the VM type and VM number to minimize application cost while meeting job deadlines.

Problem (2): Auto-scaling to minimize cost and meet application deadlines for cloud workflows. It targets budget-unlimited workflow applications. Jobs have individual deadlines. The goal is to determine the VM number and schedule the jobs on the acquired VMs to minimize application cost and meet job deadlines.

Problem (3): Auto-scaling to maximize application performance within budget constraints for cloud workflows. It targets budget-limited workflow applications. Jobs have priorities. The goal is to determine the VM number and schedule the jobs on the acquired VMs to maximize the application performance within the budget constraints.

Problem (4): Auto-scaling and intermediate data management for data-intensive workflows. It targets budget-unlimited data-intensive workflow applications. The goal is to not only determine the resource provisioning and allocation plans, but also develop an intermediate data management policy to minimize application cost and meet job deadlines.

1.2.3 Challenges

The key benefit and driving force of the cloud is to provision resources in response to demand dynamically and only pay for the resources consumed. The real challenge lies in how to determine the computing capacity and allocate the resources cost-efficiently. The benefit of cloud dynamic scalability can only be realized when this question is answered sufficiently. This is a dynamic mapping problem from the application objectives to the underlying resources. It is a challenging problem because of the following reasons.

- The application structure could be simple or complex. The batch-queue based applications accept bags-of-tasks in which there is no dependencies, while the SOA based applications accept workflow jobs in which the task precedence constraints need to be preserved.

- The workload is dynamic. It may experience seasonal behaviors. This requires that the auto-scaling mechanism closely monitor the workload information and make resource acquisition/release decisions in response to the demand in time.
- The mapping problem includes two subproblems - the size of resource pool (including the VM types and numbers) and the job placement (scheduling the jobs onto the provisioned resources). These two subproblems are connected with each other. On the one hand, the scheduling decisions determine the resources that need to be provisioned. On the other hand, the provisioned resources are the resources available on which jobs can be scheduled. This is a circular reference problem.
- It should not only determine the numbers of the VM instances but also the types of the VM instances, in other words, both vertical scaling and horizontal scaling. A fixed budget can purchase different types of machines for different durations. The heterogeneous environment makes the problem even more challenging.
- It should help service providers to make trade-offs among conflicting goals, such as maximizing the application performance within the budget constraints, or minimizing the cost while meeting job deadlines.
- In practice, cloud VMs take time to start up and are billed by instance hours. This requires the auto-scaling mechanism carefully handle these practical concerns, such as reacting in advance to accommodate the possible VM startup delay and avoid wasting partial instance hours.
- Scheduling the processing nodes of a data-flow graph onto a set of available machines is a well-known NP-hard problem, even in its simplest form [27][28], Therefore, optimized and fast solutions are needed to speed up the auto-scaling decision making process.

1.3 Contributions

This dissertation makes several contributions towards the automatic resource provisioning and allocation (auto-scaling) in the cloud. Driven by the motivating examples and problem description in the previous section, the major contributions of this dissertation are:

- This dissertation targets applications built on the cloud resources instead of some specific workload known in advance, e.g. a single job or job ensembles. Therefore, the cost constraint is applied to the whole application. The solution is particularly beneficial to the service providers instead of service customers. This dissertation covers two important application models, batch-queue models (bags-of-tasks) and SOA (workflows) models.

- In terms of application goals, this dissertation solves both dimensions of the optimization problem - minimizing the application cost within the deadline constraints and minimizing the job turnaround time within the cost constraints.

- It uses both formal methods and heuristics to solve the auto-scaling problems. In the batch-queue based applications, an integer programming based technique is developed to determine the resource size. By ensuring the computing power is always large enough to handle the workload for all the VM types, our solution can finish more than 95% jobs before the deadline and save 20.2% - 40.1% cost compared to a fixed machine type choice. In the workflow applications, several innovate heuristics are developed to break task dependencies, calculate VM numbers, distribute budget, schedule jobs and manage the intermediate data. In the unlimited budget case, dynamic scaling-consolidation-scheduling (SCS) can save 9.4% - 40.4% cost compared to two baseline approaches and can work well in both light and heavy workload environments. In the limited budget case, the scheduling-first and scaling-first algorithms can reduce 9.8% - 45.2% job turnaround time than the standard machine choice.

- In the cost-constrained subproblem (chapter 5), this dissertation solves the circular reference problem through two approaches - scaling-first and scheduling-first. These two algorithms make resource provisioning and allocation decisions in different orders. The scheduling-first algorithm allocates budget first and schedules tasks to their fastest machines. The scaling-first algorithm makes resource provisioning decisions first by looking at the overall workload and schedules tasks based on priority. Particularly, we find the scaling-first algorithm works better under low budget ranges while the scheduling-first algorithm works better under high budget ranges. This is a trade-off between the job waiting time and job execution time.

- In the intermediate data management subproblem (chapter 6), this dissertation explicitly models the performance and cost of the intermediate data. To the best of our knowledge, this is the first work to consider intermediate data management in the resource provisioning process. We present three job scheduling policies and a data prefetching strategy to manage intermediate data for data-intensive applications. Particularly, the cost-deadline-first (CDF) algorithm can save 13.5% - 33.7% cost compared to the deadline-first (DF) algorithm because of the higher resource provisioning-allocation consistency. The data prefetching strategy can further improve the cost saving up to 44.6% through data locality aware job placement.

- It explicitly considers two important practical issues in the cloud - VM startup delay and partial instance hours. VM startup delays are modeled in the instance acquisition process. Partial instance hour handling is considered in the instance consolidation process. In the cost-constrained subproblem, we show that the instance consolidation process can help to increase the resource utilization rate by 2.2% - 19.9% and reduce job turnaround time by 9.0% - 35.1%.

- The auto-scaling mechanisms show good tolerance to inaccurate parameters, such as the task execution time, VM startup time, communication ratio, etc. For example, in the cost-constrained subproblem, when the VM startup time and task execution time allow ±20% error, the performance difference ranges from -10.2% to 16.7%. It does not show significant performance degradation until the estimation error reaches ±60%.

1.4 Dissertation Overview

The remainder of this dissertation is organized as follows.

Chapter 2 discusses the related work. The comparison to the related work in the cloud is performed by specific sub categories in detail.

Chapter 3 solves the cloud auto-scaling problem for batch-queue based applications. The workload is considered as bags-of-tasks (i.e. independent jobs) and the solution is based on the integer programming technique. This work has been published in the 2010 11th IEEE/ACM international conference on grid computing (Grid 2010, acceptance rate 21%) [20].

Chapter 4 solves the auto-scaling problem of minimizing cost and meeting application deadlines for cloud workflows. It assumes the application is based on the Service Oriented Architecture and the workload are workflows in which the precedence constraints need to be preserved among the sub-tasks. A deadline assignment and load vector based heuristic is developed to determine the computing capacity. Earliest-deadline-first algorithm is used to schedule the workload. This work has been published in the 2011 international conference for high performance computing, network, storage and analysis (SC 2011, acceptance rate 20%) [29].

Chapter 5 solves the auto-scaling problem of minimizing job turnaround time within the budget constraints for cloud workflows. This chapter solves the other dimension of the optimization problem (switching the goals and constraints) compared to chapter 4. It develops two solutions - scaling first and scheduling first - to tackle the circular reference problem. These two algorithms make scaling and scheduling decisions in different orders and show different performance under different budget ranges. This part of the work has been submitted to a conference for review [30].

Chapter 6 solves the auto-scaling and intermediate data management problem for data-intensive applications, in which the data transfer performance and cost are explicitly modeled in the solution. A data prefetching strategy is developed to reduce the intermediate data transfer overhead and cooperate with the resource provisioning process. This work is in submission ready state [31].

Chapter 7 concludes this dissertation and discusses future work.

Chapter 2

Related Work

This chapter discusses the related work. It compares this dissertation with the related research in both the grid and cloud environments. In terms of underlying implementation technology and the way users consume resources, grid and cloud share several common features, such as cluster technology, distributed computing technology, virtulization, resource pooling, etc. However, this does not mean that this dissertation is resolving the resource provisioning and allocation problem. The next two sections detail and summarize the major differences between the grid and cloud. This can be considered as a vertical comparison, comparing the current with the past. As the cloud has become an important and mature platform, this dissertation also sees many other cloud resource provisioning and allocation research works in the past a couple of years. It is a hot research topic. This dissertation will compare with these research works by sub-categories and this comparison can be considered as a horizontal comparison. Finally, this dissertation discusses the advantages and disadvantages of the current industrial auto-scaling solutions.

2.1 Differences between Grid and Cloud

Resource management and job scheduling have been studied extensively in the grid environment. Though the cloud shares many common features with the grid, it does not mean that the auto-scaling problems presented in this dissertation have been solved already. This section will compare the differences between the grid and cloud, discuss the changes that the cloud has brought to their users, describe the challenges that cloud users will encounter when using the cloud resources, explain why the auto-scaling problems have not been solved and this dissertation does not try to reinvent the wheels.

The key problem in the grid is federation - how to federate loosely coupled, heterogeneous, geographically dispersed computing resources from multiple administration domains to reach a common goal [32][33][34][35].

The key idea in the cloud is hosting. As one of the grid pioneers Ian Foster defined, the cloud is a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet [36].

However, from the resource consumer (user) perspective, the border between the cloud and the grid start to blur. Users do not care who provides the resources (whether the resources come from a single giant compute power provider who try to earn money by leasing their resources, or multiple heterogeneous virtual organizations who aim to help resource-limited users to achieve common goals) and the how the resources are connected and provisioned. For the users, no matter it is the grid and cloud, they just see a resource pool from which they can acquire resources on demand. However, the important fact is that these two resource pools have different characteristics, which essentially determine that the way of using the resources are different. This dissertation focuses on the“user-side differences” between the grid and cloud. It tries to answer the questions, like how to use the resources, not how to build a grid or a cloud.

- Generally speaking, a grid is shared by multiple users. Therefore, depending on the design of the grid schedulers, user jobs could largely interfere with each other. Though a cloud may serve users in a much larger scale at the same time, users do not interfere with each other because of the performance virtualization and isolation.

- The resource pool is unlimited to cloud users (in practice, at least large enough to handle all the resource requests from the users), but limited and dynamically changing to grid users (it depends on the number of participants in the grid at runtime).

- In the cloud (IaaS), users have the full control of their resources, e.g. software stack installation and security configurations. However, in the grid, users normally do not have such low level controls and can only submit a job to a service.

- The availability and quality-of-service are guaranteed by the cloud providers while the service level agreements in the grid are not always available. It could be possible that a users' job is interrupted because some higher priority jobs are submitted or the remote resource suddenly goes off-line.

- In the cloud, the resource is billed by resource type and instance hours ($/hour/machine type). Every byte transferred and every CPU cycle consumed will be charged. The billing model is unambiguous and clear. In the grid, though the utility concept has been introduced for many decades, the cost of a resource is not clearly defined or consistently agreed. Among the limited research works, the most common billing model assumption is $/job/service. However, it is not trivial to map between these two billing models. Moreover, the IaaS Cloud also charges for the storage space (\$/GB-Month). Though the grid may also offer central storage services, the cost model has not been discussed [37][38].

All these features enabled by the cloud open up a series of interesting possibilities for the users to build performance- and cost-aware applications for job processing, while grid users can only process their jobs one by one because of the limitation of the grid. Therefore, in the cloud, the goals and constraints, such as deadlines, budgets, resource utilization, service level agreements (SLA), etc. are applied to the application as a whole. While in the grid, the goals and constraints are normally applied to every individual job that may compete for the limited resources.

Though the "resource selection" problem in the grid has been extensively studied, most of the research works are mainly based on resource availability, capability, job performance and users' quality-of-service requirements. They focus on the service discovery, negotiation and inter-operation protocols, and system implementation. The first-class goal is to complete the job successfully and quickly. This is essentially determined by the characters of the grid and its users. Resource-limited geographically distributed organizations contribute their resources and collaborate to solve the problem that cannot be solved using own their computing infrastructure. In the cloud, however, the limitation of the resource size is not a problem any more. Cloud users can always acquire sufficient resources from the cloud. It is cheaper and they only need to pay for what they use. They also have more control. Therefore, the key of the problem shifts from successful completion of the job to finishing the job in a timely manner with minimum cost.

Another important difference that needs to be emphasized is the resource cost model. Among many optimization goals and constraints, resource selection based on performance and cost has not been sufficiently discussed in the grid. One of the main reasons is that the cost of the compute resources and services is not clearly modeled or defined. The notion of cost is vague in practice though the concept of "utility computing" concept was first introduced in the 1960s. For example, the computation cost is priced by job instead of by instance hour. Data transfer cost is either included as part of the computation cost, or ignored in the cluster environment. Data storage cost is simply not considered.

Moreover, there are some unique cloud features that need to be carefully addressed. For example, though VMs can be acquired at any time with any amount, the VMs may take some time to be ready to use, depending on the image size, data center location, VM type, the number of VMs acquired at same time, etc. [39]. VMs are always billed by hours. Partial instance hours are always counted as full hours. Ignoring this fact and simply assuming the task always runs for multiples of full hours can incur a large amount of waste [21].

2.2 Job Scheduling in the Grid

This section discusses the most closely related work in the grid in detail.

For job scheduling in the grid, there are two types of jobs - independent jobs (bags-of-tasks) and workflows. Independent jobs can be executed on any computing service in any grid site, while workflow scheduler involves more complex mechanisms and needs to consider more factors to handle task dependencies. This dissertation covers two application architectures. The batch-queue based applications accept bags-of-tasks as the workload, while the SOA based applications accept workflows as the workload. The difference is that the grid work targets a job (or a set of jobs) while this dissertation targets an application as a whole. Depending on the number of the workflows (workflow multiplicity) scheduled at the same time, there are two classes of workflow scheduling processes - single workflow and multiple workflows[40][41]. In this dissertation, the auto-scaling mechanisms target the whole application and there could be multiple jobs submitted by users. Therefore, this dissertation deals with multiple workflows. However one important difference worth mentioning is that, the workload is changing dynamically. It is not workflow ensembles, in which the multiple workflows are known in advance. In terms of the dynamism of the scheduling process, there are three types [42]. They are just-in-time scheduling (fully dynamic approach) [43], full-ahead planning (fully static approach) [44][45] and hybrid scheduling [46][47]. This dissertation falls into the hybrid category. The capacity determination process (scaling) is static, in which the algorithms need to plan the job placement before execution. The resource allocation process (scheduling) is dynamic, in which a job can be scheduled on a different machine. Both decisions are updated periodically based on the runtime information.

The following four research works in the grid are considered as the most related with this dissertation. They are four different ideas about scheduling workflows with deadline and budget constraints in the grid. Particularly, the deadline assignment and VM upgradtion ideas in this dissertation are inspired by them.

Sakellariou et al. [48] present two incremental approaches LOSS and GAIN to find an execution plan for a workflow within the deadline constraint. The idea is to first assume all the tasks run on their fastest(cheapest) machines and calculate the job makespan. The next step is to replace the execution machine for the task which has minimum(maximum) performance loss(improvement) but maximum(minimum) cost gain(increasement), until the makespan is beyond(within) the deadline. Essentially, the logic behind the idea is to start with the worst plan and upgrade task scheduling based on the cost-efficiency rank in each step. The final result is therefore an optimized execution plan. In this dissertation, it uses the GAIN approach to find a feasible execution plan for each job class. That is one preprocessing step before the deadline assignment.

Yu et al. [49] present a solution to finish a workflow before the deadline with minimum cost. They assign the workflow deadline to each sub task and select the cheapest resource that can finish the task within the execution interval determined in the deadline assignment phase. In this way, if every task can be completed within the sub-deadline, then the whole job will be finished in time. The deadline assignment idea inspires the solution to the deadline-constrained problem in chapter 4. The solution also assigns deadlines to sub-tasks. However, instead of choosing the cheapest resource, it picks the cost-efficient VMs. An important step developed in this dissertation is to calculate the number of the VMs using load vectors, in which tasks with non-urgent deadlines can wait for the resources. The original solution cannot determine the number of VMs needed. It will perform badly when the task parallelism is high. Moreover, in chapter 6, this dissertation introduces a level based deadline assignment technique to improve data locality and reduce data transfer overhead.

Menasce et al. [50] present a backtracking algorithm to schedule a workflow within the deadline constraint and minimize the cost. It starts with the cheapest plan similar to the GAIN approach in [48]. However instead of defining the GAIN metric to upgrade task execution incrementally, this work always starts from the last task, upgrading the task to the next more expensive machine until the deadline is met. If the workflow still cannot be finished in time, the algorithm will backtrack to the previous tasks. One assumption in this work and [48] is that more expensive machines are faster machines, which is not true in the cloud. Moreover, the cost models are all based on dollars per job which is different from current cloud billing models.

Yu et al. [45] present a workflow scheduling technique using genetic algorithms. The idea is to start with any random scheduling plan and the genetic algorithm will automatically evolve to an optimized solution. Though the genetic algorithm makes it a systematic solution framework and enables users to concentrate on the scheduling criteria by defining the fitness and crossover functions, the fitness and crossover functions are not easy to define, especially when considering the data transfer and storage plans. More importantly, the genetic algorithm does not answer the capacity determination question and it is not trivial to extend it to the cloud environment. The overhead could also be very high when the problem size becomes large.

2.3 Resource Provisioning in the Cloud

The ability to acquire resources dynamically and trivially in the cloud - while being incredibly powerful and useful - essentially exacerbates this particular resource provisioning and allocation problem. It has drawn lots of attention from the research community. The current research work may make different assumptions on the cloud resources and application execution environments. They may target different types of workload, such as bags-of-tasks and workflows, or they may have different application objectives, such as maximizing the application performance, minimizing the cost and meeting service level agreements. In terms of solutions, they may use either formal methods or heuristics to solve the optimization problem. This section compares this dissertation with the related work in the cloud in detail. Figure 2.1 summarizes the sub categories are used.

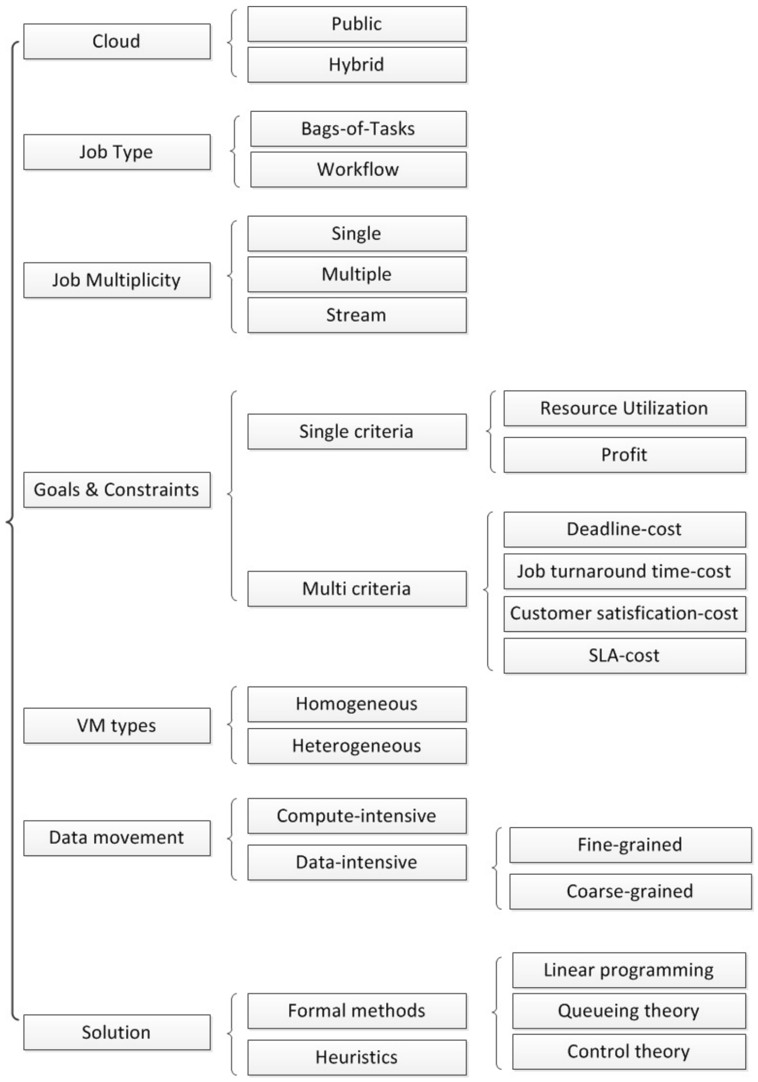

Figure 2.1: Sub categories of related work

Bags-of-tasks vs workflows. There are two types of workload. One is bags-of-tasks, in which jobs are independent and embarrassingly parallel. The other one is workflows, in which tasks belonging to the same job need to preserve the precedence constraints.

Abrishami et al. [51] propose two algorithms to schedule workflows on IaaS cloud with deadline constraints. The general idea is to assign sub deadlines to tasks based on the critical path and choose the cheapest service to finish the task within its sub deadline. Their idea is similar to this dissertation by first assigning deadlines to tasks, but they use critical paths while this dissertation uses the ratio of deadline over planned job makespan. Both of their algorithms are designed for a single workflow instead of a dynamic workflow stream as in this dissertation. Moreover, they model a homogeneous network environment in which the data transfer speed is the same among all the services. In this dissertation, chapter 4 considers the data transfer time as part of the task execution time while chapter 6 models the data transfer in a heterogeneous environment using the operator overlap technique [21].

MarShall et al. [52] present three simple provisioning policies to dynamically acquire VMs in the cloud to increase the site capacity. They target bags-of-tasks. The goal of their policies is to maintain the length of the job queue under some threshold. The total job wall time and estimated wait time are used to calculate the number of VMs needed. The goal (the queue length) they choose is a low level metric in this work, just like some resource utilization indicators, such as CPU utilization rate, I/O operations, etc. For many cases, it is not trivial to map the application performance to the resource indicators and determine the threshold value. The provisioning module runs in a loop periodically examining the job queue, executing a policy and performing cluster management functions, which is similar to the monitor-control loop idea in this dissertation. At the same time, their work also presents the architecture and implementation of a resource manager based on Nimbus toolkit [53], and discusses various logistical considerations, such as security and privacy.

In this dissertation, chapter 3 targets the batch-queue based applications that accept bags-of-tasks as the workload. Chapter 4, 5 and 6 target the SOA based applications that accept workflows as the workload.

Cloud application vs jobs. From the resource consumer perspective, the resource provisioning strategies may target different objects. For the end consumers, they normally focus on a single job or a set of jobs (job ensembles), which are known in advance. This is very similar to the grid scenario. The optimization is job-oriented. For the service providers who build their applications on the IaaS cloud, they focus on the whole application instead of individual jobs. The goals and constraints are applied to the application. The workload is dynamic.

Xu et al. [54] propose makespan- and cost-aware scheduling strategies for multiple workflows in the cloud. Their key idea is to model the time and cost quota using the ratio of single task's attributes over the whole workflow's variance sum, and try to make every task finish within its planned time interval without exceed the cost quota. However, their deadline and budget constraints are still only for individual worfklow jobs. Their work is not applicable to an application wide constraint.

Iosup et al. [55] propose three resource provisioning and scheduling policies to maximize the total utility from the submitted workflow ensembles given the deadline and budget constraints. Essentially, they develop a job admission procedure based on the information of individual workflow structure and the overall workflow ensembles composition. With incremental knowledge of the workload information, the performance of the provision/scheduling policies improves over one another. However, in their work, they only consider a single

VM type (a homogeneous environment). They also assume that the workload information is known before all the provisioning/scheduling decisions are made.

Service providers are the targets of this dissertation. Therefore, the cost constraint is applied to the whole application instead of every single job. As a continuously running application, it is not practical to identify the start and end of the service life cycle. The average job turnaround time and total application cost are all accumulated results in the observation period.

Application goals and constraints. The resource provisioning and allocation mechanisms may have different goals and constraints, such as minimizing cost, minimizing job response time, maximizing resource utilization, maximizing service profits, etc. Some works do not have specific goals and constraints. They focus on the interplay between two conflicting factors, such as the job makespan and cost.

Wu et al. [56] present a service level agreements (SLA) and VM mapping mechanism to minimize SLA violations and cost for service providers running on the IaaS cloud. One unique assumption of their model is that the service customers are requesting VM capacity directly instead of submitting jobs, which is determined based on a table of QoS parameters. Such assumption greatly simplifies the problem and essentially converts it to a bin-packing like problem. Specifically, they propose best-fit and worst-fit algorithms to take advantage of the partially used VMs. Their work is a complement to the job-based workload and one of the few works discussing the multi-tenancy environment (a VM is shared by more than one customer request).

Villegas et al. [57] analyze the performance and cost of different resource provisioning and allocation policy combinations. They assume a homogeneous environment and the workload is CPU-bound bags-of-tasks. They propose eight provisioning policies based on the queue size, execution time and waiting time, and four allocation policies, such as First Come First Serve (FCFS) and Shortest Job First (SJF). The performance of the provisioning and allocation policies are analyzed and ranked. One conclusion is that the allocation policy needs to be consistent with the provisioning decisions, which is also confirmed by the results in this dissertation.

Chen et al. [58] present two algorithms to balance the service provider profits and customer satisfactions using utility theories. They assume one instance hour is an execution slot and a job execution may spread among multiple slots. Resource provisioning and allocation decisions are made between adjacent execution slots. For each request, the best instance type will be chosen based on the profit or satisfaction rank for the next execution slot. The work is innovative to bring utility theory and customer satisfaction into the resource provisioning process. However, they have two assumption limitations. All jobs need to be served in every execution unit until it finishes, which means the algorithm essentially trades parallelism for waiting time, even for the non-urgent jobs. Second, the execution slot assumption is not practical in the current billing models. One instance hour is too long for an execution slot. This essentially means many partial instance hours could be wasted. Moreover, they assume a job can be restarted on a different machine type with negligible overhead.

Lee et al. [59] present a job scheduling mechanism to maximize the service provider's profit in the cloud. They assume every job is associated with a profit that the service provider can earn. The goal is therefore to maximize the overall application profit, in which some jobs may be greatly delayed or even ignored. They use queuing theory and probability theory to measure the potential profit for a task and develop two sets of profit-driven scheduling algorithms. The first set takes into account not only the profit achievable from the current service, but also the profit from other services being processed on the same service instance. The second set attempts to maximize service-instance utilization without incurring loss/deficit, which implies the minimization of costs to rent resources from infrastructure vendors. They assume a processor-sharing (multi-tenancy) environment, but only consider a single VM type.

The application goals and constraints studied in this dissertation include deadlines, job turnaround time, priorities and cost. For deadline-constrained problems, the goal is to minimize cost. For cost-constrained problems, the goal is to minimize job turnaround time.

Formal methods vs heuristics. The solutions generally fall into two categories. One is formal methods, which convert the resource provisioning problem to some mathematical optimization model. More specifically, this dissertation sees related work uses linear programming, queuing theory and control theory to estimate the future workload and determine the number for each VM type. The other category is heuristics. Normally, a greedy approach is applied in every step based on some performance metric and an optimized solution can be achieved in the end.

Bossche et al. [60] propose an binary integer programming approach to minimize job execution cost within deadlines in a hybrid cloud environment. They assume a job is composed of parallel tasks and an application is composed independent jobs with deadlines. The longest deadline determines the number of instance hour slots. The problem is then converted to choosing an instance from a cloud provider for each task at any time slot to minimize the overall execution cost. To support the hour slot setting, they assume the task is preemptive. Essentially, they target a set of jobs instead of an application. The solution shows slow performance when the problem size becomes large.

Pandey et al. [61] model the cost of an intrusion detection workflow execution on cloud resources using a Non-Linear Programming (NLP) solution. The NLP-model retrieves data partially from multiple data sources based on the cost of transferring data from those resources to a compute resource, so that the total cost of data-transfer and computation cost on that compute resource is minimized. However, they assume that the output data of each data is staged back to the cloud storage as the part of the task's execution, in which data locality among compute instances are not considered. Moreover, the cost minimization is job based instead of application based and deadline is not treated as an optimization goal.

Zhu et al. [62] design a resource provisioning framework for adaptive applications in the cloud. The key component of the framework is a dynamic resource provisioning algorithm based on control theory. They define a benefit function depending on the number of outputs, priority and cost. The CPU and memory are dynamically allocated using SVM regression to maximize the benefit function while meeting the deadline and budget constraints. Essentially, they provision the resources at a lower level than virtual machines. The assumption is that CPU and memory can be controlled in a finer granularity, which may not be practical because very few service providers can directly control the virtualization layer in the cloud data center.

In this dissertation, chapter 3 solves the auto-scaling problem for batch-queue based applications using the integer programming technique. The solution is to minimize the total VM cost while making sure the computing power is large enough to handle the workload. Chapter 4, 5 and 6 solve the auto-scaling problems using heuristics. Capacity is determined through deadline assignment, load vector and budget allocation techniques, while jobs are scheduled based on deadline, cost and cost-efficiency ranks.

Public cloud vs hybrid cloud. Hybrid cloud is a composition of two or more clouds (private, community or public) that remain unique entities but are bound together, offering the benefits of multiple deployment models. For security and legacy system design reasons, some computing infrastructure are kept on-premise. Cloud resources are acquired when the local resources are not able to handle the workload.

Dornemann et al. [63][64] extend an open source BPEL implementation to dynamically schedule the service calls of a BPEL process based on the target hosts' load. Essentially, they develop a provisioning component to dynamically acquire/release Amazon EC2 instances based on the service workload information. Their work focuses on the workflow engine implementation that can bring in cloud resources when local compute power is insufficient. However, they do not consider multiple VM types that are available in the cloud or consider either performance or cost as the optimization goal.

Bicer et al. [65] propose an algorithm to acquire cloud instances to support both time and cost sensitive execution for data-intensive applications executed in a hybrid cloud environment. They assume an application is composed of independent jobs which has the same amount of data to be processed and takes the same amount of execution time on a given instance type. They also assume a fixed-size local resource pool and a master node which assigns tasks between the local cluster and cloud instances. The idea is to increase the number of cloud instances and redistribute some cloud workload into the local cluster to reduce job execution time. Essentially, they are targeting a single job composing parallel tasks, in which the task is executed either in the local cluster or the cloud. However, they do not explicitly model the data transfer time for the stolen tasks which is a big drawback in this work.

Assuncao et al. [66] discuss different strategies to minimize job response time in a hybrid cloud environment. Depending on the job requirement and the scheduling strategy, a job is either routed to the site queue or the cloud queue for processing. The findings include that (1) Naive scheduling strategies can result in a higher cost under heavy load conditions. (2) The cost of increasing the performance of application scheduling is higher under a scenario where the site's cluster is under-utilized. (3) In addition, request backfilling and redirection based on the expansion factors (i.e. selective backfilling) show a good ratio of slowdown improvement to the money spent for using cloud resources. However the capacity determination problem in the cloud is not fully discussed in this work. It is not clear how the number of VMs for each job is determined. Data communication is also not considered in their scheduling strategies.

Calheiros et al. [67] present an architecture for coordinated dynamic provisioning and scheduling that is able to cost-efficiently complete applications within their deadlines by considering the whole organization workload at individual tasks level when making decisions. Their idea to create three queues for non-deadline jobs, deadline jobs and external jobs that will be executed in the cloud. When non-deadline and deadline queues are empty, external jobs can be executed using the local resources and when the non-deadline and deadline queues are full, more cloud resources need to be acquired to process the internal jobs in time. However they only consider a single VM type and do not discuss the data management issues which is one important aspect of the hybrid cloud. Also the workload they deal with is bags-of-tasks.

This dissertation assumes that the service provider build its application fully based on a public cloud. The job scheduling is considered among individual VMs instead of among local/remote computing sites. The job scheduling granularity is finer and the data transfer cost is explicitly modeled and defined. Extending the solutions to support the hybrid and federated cloud environments remains one of the future work.

Compute-intensive vs data-intensive. One important factor during the resource provisioning process is the modeling of intermediate data transfer and cost. Currently, most related work focuses on the compute-intensive workload and do not consider the intermediate data management. Or they assume the intermediate data is stored in a central storage system (such as Amazon cloud storage S3 [68]) and data uploading/downloading is part of the task execution.

Kllapi et al. [21] define operator graphs for data processing flows in the cloud and explore the trade-offs between the workflow job completion time and monetary cost. The key idea is that the operator graph model (S&F and PL) converts the data transfer process as a part of task execution. It also explicitly discusses task multi-processing and uni-processing on a shared container (a multi-tenancy environment). However, compared to this dissertation, they only model the data transfer time but not the cost. The cloud storage is not considered as a storage option and the optimization model can only be applied to a single workflow job.

Xie et al. [69] design a data placement strategy in a heterogeneous Hadoop cluster environment. Their idea is to distribute the data proportionally based on the local node's compute power. In other words, faster nodes get more workload. In this way, a large volume of data transfer is saved, because all the nodes (even with different computing speed) have similar task finishing time and can access their input locally. The idea of distributing data based on processing power to increase data locality is inspiring. However, the capacity determination question is not really answered in this work.

Yuan et al. [70] create an intermediate data dependency graph to record all the data sets information that have existed in the system. By comparing the regeneration cost and the storage cost of each data set, the system then decides whether the data set should be stored or deleted. In their model, they rely on the estimation of usage rate to determine both the regeneration and storage cost, and also a manually configured time threshold to check the stored data set. In their work, they focus on the shared data among different jobs. The management of immediate data generated during the job execution is not considered. Moreover they assume all the data will be stored in the cloud storage instead of the VM storage.

Shankar et al. [71] propose a workflow scheduling algorithm for data-intensive applications in a cluster environment. The scheduling criteria considers both the data transfer time and task execution time. For each workflow, it determines the best execution machine by calculating the total task runtime (including both the data transfer time and task execution time). In this way, the scheduling decisions take data locality into consideration when making the job placement decisions. However one big limitation is that every workflow job is treated as a single entity and it assumes all the tasks belonging to the same job will be executed on the same machine. They further extend the static algorithm by producing plans incrementally. The planning algorithm is split over multiple cycles. In each cycle, only a portion of each pending workflow is planned. The main goal of this work is to minimize the execution time for each individual job. It cannot accommodate deadline and budget constraints as discussed in this dissertation. Moreover, the resource pool is a cluster and the scheduling criteria is essentially best-effort, therefore the capacity determination problem is not really covered.

In this dissertation, chapters 3, 4 and 5 can be considered as auto-scaling solutions for compute-intensive applications. They assume the task input data is downloaded from some central storage before task starts and the output data is uploaded back to the storage after task finishes. The data transfer time is considered as part of the task execution time. Chapter 6 extends the model to data-intensive applications by explicitly modeling the performance and cost for data transfer/storage and proposes an intermediate data management policy that can cooperate with the resource provisioning and allocation strategies.

2.4 Industrial Solutions

Cloud providers and third-party cloud services [6][72][73][74] have developed schedule-based (e.g. time-of-the-day) and rule-based (e.g. CPU utilization thresholds) auto-scaling mechanisms to help cloud service providers to dynamically acquire/release resources. The mechanisms are simple and convenient. However, it is not always straightforward for the users to select the "right" scaling indicators and thresholds, especially when the application models are complex, and the resource utilization indicators are limited and very low-level. In other words, these trigger mechanisms do not really solve the performance-resource mapping problem, and sometimes the resource utilization indicators are not expressive enough to address user performance requirements directly. Moreover, these trigger mechanisms do not carefully consider the availability of different VM types, which should be chosen wisely based on the workload. More importantly, these trigger mechanisms do not answer the second question, i.e. resource allocation. Cloud users still need to allocate the resources to the computing activities by themselves. As shown in other related work and later chapters in this dissertation, inconsistent resource provisioning and allocation decisions could result in bad application performance and more cost. Finally, the trigger mechanisms are not aware of the user budget constraints. They are not easy to be applied to the budget limited applications.

2.5 Conclusion

Though self-servicing on-demand resource acquisitions give cloud users more power and flexibility, the resource ill-sizing problem does not disappear. It actually imposes new burdens. The service providers have to determine the right capacity in a dynamic manner. Therefore, the problem focus shifts from making the best efforts in the fixed-size resource pool to determining capacity and allocating acquired resources at the same time. The other significant change is the cost model definition. Every CPU consumed and every byte transferred are measured and billed. The model change drives researchers to reconsider the way that cloud resources are used in their solutions.

In summary, though this dissertation shares some common background with the related work, it does not try to resolve the problem. It is not a single factor that makes this dissertation different from the previous research, but the new features and challenges brought by the cloud that makes the auto-scaling problem a unique and unsolved problem. It targets a cloud application instead of a single job (or a set of jobs). It considers both the vertical scaling (VM type) and horizontal scaling (VM number). It covers both batch-queue based applications and workflow applications. Both formal methods and heuristic approaches are explored. The intermediate data management is incorporated in the resource provisioning and allocation process. The job scheduling decisions are made among VMs which is more fine-grained than the hybrid cloud research. Cloud practical issues, such as VM startup delay and partial instance hours are carefully considered.

Chapter 3

Auto-scaling for Batch-queue Based Cloud Applications

This chapter presents the auto-scaling solution for cloud applications based on the batch-queue model. It first details the assumptions and formalizes the problem. Next it develops an auto-scaling solution based on the integer programming technique. Finally, it evaluates the solution with both simulations and a real-life scientific application MODISAzure [22] running in the cloud.

3.1 The Batch-queue Based Cloud Application Model

The general and simple application model is the batch-queue model, in which the workload is bags-of-tasks (embarrassingly parallel jobs) that is submitted to a single job queue. The jobs are associated with deadlines, that indicate the preference of the job completion time. When the application is running in the cloud, service providers need an automatic mechanism to determine the right resource set. The acquired computing power should be large enough to handle the workload within the deadlines and but not more than necessary to cost extra money. The batch-queue model is shown in 3.1 and the assumptions are summarized as follows.

- The workload is independent jobs submitted into the job queue. Service providers do not know the incoming jobs in advance. The workload is dynamic.

- All jobs have the same performance goal, e.g. 1-hour job turnaround time deadline, from the time it is submitted to the time it should be finished.

- Jobs are served in the First Come First Server (FCFS) manner. Every instance can only process a single job at one time.
- VM instance acquisition requests can be made at any time, but it may take several minutes for a newly requested pending instance to be ready to use [39]. Such time is called VM startup delay.
- There could be different classes of jobs, such as compute-intensive jobs and I/O intensive jobs. A job class may have different execution times on different instance types. For example, a compute-intensive job can run faster on high-CPU machines than on high-I/O machines.

It is important to claim the FCFS assumption, which seems to contradict the classic real-time scheduling ideas. Because all the jobs have the same job turnaround time deadline and are treated fairly, a job always has an earlier deadline than jobs submitted later in terms of absolute time. This chapter does not assume the auto-scaling mechanism can control the job execution order or the job execution instance (i.e. no job scheduling support). Therefore, all jobs are assumed to be processed in the FCFS manner. In summary, the **goal** of the auto-scaling mechanism is to automatically provision cloud VM instances to enable all the jobs to be finished before deadlines with the minimum amount of money.

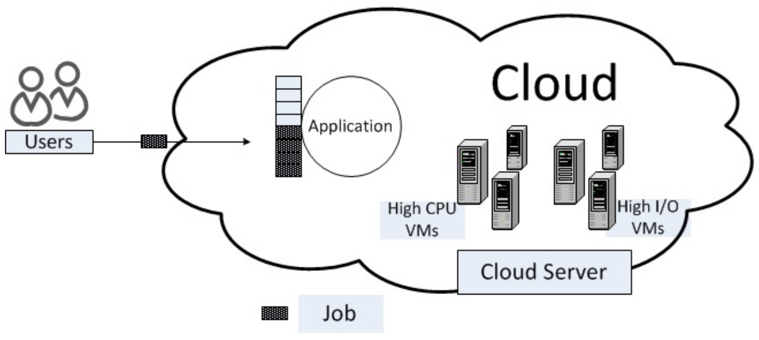

Figure 3.1: The batch-queue based cloud application model

3.2 Solution

Based on the problem description in the previous section, the problem is formalized and the implementation architectures in Windows Azure is presented in this section. One of the key insights to this problem is that, to finish all the submitted jobs before the deadline, the auto-scaling mechanism needs to ensure that the computing power of all acquired VM instances is large enough to handle the workload. The key variables are summarized in 3.1.

The system workload can be represented as a vector $W = [n_1 \cdots n_j]$. For each job class J_j, there are n_j submitted jobs. The computing power of instance I_i can be represented as a vector $P_i = [p_1 \cdots p_j]$. The idea

Variables	Meaning
J_j	the jth job class
n_j	the number of jobs (of job class J_j) submitted
V_v	the VM type
I_i	the ith instance (running or pending)
c_v	the cost per hour of VM type V_v
d_v	average startup delay of VM type V_v
s_i	the time already spent in pending status of I_i
$t_{j,v}$	average execution time of running job J_j on V_v
D	deadline (e.g. 1 hour or 100 seconds)
W	workload - jobs that **need** to be finished
P	computing power - jobs that **can** be finished

Table 3.1: Key variables in the batch-queue model

is to calculate the number of jobs that can be finished for each job class before the deadline on instance I_i. This dissertation uses the ratio of the deadline over the individual completion time (assume all the jobs are finished on that instance) to approximate the number of jobs that can be finished (equation 3.1).

$$P_i = (\frac{D}{\sum_j t_{j,type(I_i)} \times n_j} \times n_1, \cdots, \frac{D}{\sum_j t_{j,type(I_i)} \times n_j} \times n_j) \tag{3.1}$$

For the instance whose status is pending, its computing power is defined as equation 3.2, where s_i is the time already spent in starting the instance (pending status).

$$P_i = (\frac{(D - (d_{type(I_i)} - s_i))}{\sum_j t_{j,type(I_i)} \times n_j} \times n_1, \cdots, \frac{(D - (d_{type(I_i)} - s_i))}{\sum_j t_{j,type(I_i)} \times n_j} \times n_j) \tag{3.2}$$

Therefore, the total computing power of all the instances can be defined as $P = \sum_i P_i$. If $W > P$, which means as long as there exits at least one job class whose workload is greater than the processing power, the service provider needs to start more instances P'_i ($'$ means new instances) to handle the increased workload. The problem therefore becomes finding a VM instance combination plan, in which $\sum_i P'_i \geq W - P$. At the same time, the service provider also wants to minimize the cost for these newly added instances $Min(\sum_i c_{type(I'_i)})$.

When one instance I_s is approaching full hour operation, the service provider needs to decide whether to shutdown the machine or not. In such cases, the service provider can calculate the system computing power without instance I_s and compare it with the workload. If the computing power is still large enough to handle the workload $\sum_i P_i - P_s \geq W$, the instance can be terminated. To better explain the problem and solution, a simple example is shown in equation 3.3 and 3.4. The application accepts three job classes (J_1, J_2, J_3) and the cloud offers three VM types (V_1, V_2, V_3). Currently, the workload in the system is [60, 60, 60] and there are two running instances I_1 and I_2. The goal is to find a VM type combination $[n'_1, n'_2, n'_3]$ whose computing power is greater than or equal to target computing power (equation 3.3) and their cost is minimal among all the possible VM type combinations (equation 3.4). In this way, the problem is converted to an integer programming problem. The goal is to minimize the cost with computing power as the constraints.

$$\begin{bmatrix} J_1 \\ J_2 \\ J_3 \end{bmatrix} \qquad n'_1 \times \underbrace{\begin{bmatrix} 10 \\ 5 \\ 20 \end{bmatrix}}_{P_{J_1}} + n'_2 \times \underbrace{\begin{bmatrix} 10 \\ 20 \\ 5 \end{bmatrix}}_{P_{J_2}} + n'_3 \times \underbrace{\begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}}_{P_{J_3}} = \underbrace{\begin{bmatrix} x \\ y \\ z \end{bmatrix}}_{\sum P'} \geq \underbrace{\begin{bmatrix} 40 \\ 35 \\ 35 \end{bmatrix}}_{workload} = \underbrace{\begin{bmatrix} 60 \\ 60 \\ 60 \end{bmatrix}}_{W} - \underbrace{\begin{bmatrix} 10 \\ 5 \\ 20 \end{bmatrix}}_{P_{I_1}} - \underbrace{\begin{bmatrix} 10 \\ 20 \\ 5 \end{bmatrix}}_{P_{I_2}} \tag{3.3}$$

$$Min(c_1 n'_1 + c_2 n'_2 + c_3 n'_3) \tag{3.4}$$

This dissertation has designed and implemented the above cloud auto-scaling mechanism in Windows Azure [13]. Figure 3.2 shows the architecure of the implementation. The implementation includes four components. They are performance monitor, history repository, auto-scaling decider and VM manager. Performance monitor observes the current workload in the system, collects the information of actual job execution times and arrival patterns, and updates the history repository. VM manager works as the adapter between the auto-scaling mechanism and the cloud providers. It monitors all the pending and ready VM instances, and updates history repository with actual VM startup times. Moreover, it executes VM startup plans generated by auto-scaling decider and directly invokes cloud provider resource provisioning APIs. In this case, it is Windows Azure management API. The intention is that VM manager hides all cloud provider details and can be easily replaced with other cloud adapters. Such information hiding enhances reusability and customizability of the implementation when working with different cloud providers. History repository contains two data structures. One is the configuration file, which includes the application deadline, budget constraint and monitor execution interval information, etc. As shown in figure 3.2, service providers can dynamically control the behavior of the cloud auto-scaling mechanism by changing the configuration file. The other data structure is historical data table, which records the historical job execution time and arrival pattern information provided by performance monitor, and the instance startup delay information provided by VM manager. By maintaining historical data, the repository improves the input parameter accuracy and also helps the decider to prepare for workload surges early. Decider is the core of the cloud auto-scaling mechanism. Relying on the real-time workload and VM status information from performance monitor and VM manager, as well as configuration parameters and historical records from history repository, it solves the integer programming problem formalized in the previous section and generates a VM startup plan for VM manager to execute. The VM startup plan could be empty because the workload may be handled sufficiently by the exiting instances or it can contain instance type and number pairs to notify VM manager to acquire enough computing power. In current implementation, Microsoft Solver Foundation [75] is used to solve the integer programming problem. Acquiring instance actions are started by the decider. After every sleep interval, it invokes the logic to determine the VM startup plan. On the other hand, releasing instance actions are started by VM manager because it monitors which instance is approaching full hour operation and could be the potential shutdown target. But it has to ask decider to see whether remaining computing power is large enough to handle the workload. The implementation has been published as a library and plugged into the MODISAzure application [22]. The evaluation of the auto-scaling mechanism in this real-life scientific application can be found in the next section.

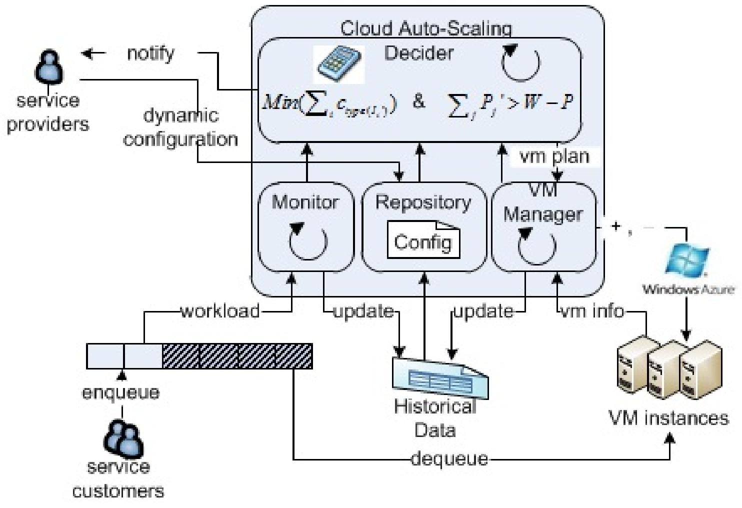

Figure 3.2: Architecture of cloud auto-scaling in Windows Azure

3.3 Performance Evaluation

This dissertation uses simulations to evaluate the performance of the auto-scaling mechanisms. Using simulations helps to control the input parameters, locate the key factors and duplicate the same testing environment. It covers a larger problem space than a specific application through different configuration combinations. It also helps to tune the application in a fine-grained manner and analyze the relationship between the application performance and the budget constraints. Moreover, it speeds up the evaluation process and saves the evaluation cost. Simulations will also be used in the following subproblems to evaluate the mechanisms' performance. The specific simulation architecture, configurations and workload patterns will be detailed further in each chapter.

In the batch-queue application model, three types of jobs are simulated. They are mix, compute-intensive and I/O intensive. At the same time, three types of machines are simulated. They are General, High-CPU and High-I/O machines. The simulation parameters are summarized in table 3.2, which are derived from the price tables and instance descriptions of Amazon EC2 [6]. For example, in EC2, a c1.medium instance costs twice as much as m1.small, but it offers five times more compute power than m1.small. In this case, mix jobs are assumed half computation and half I/O. The speedup factor of powerful machines is 4-5.

	Mix	Compute-intensive	Data-intensive
General ($0.085/h;600s delay)	Average 300s; Std 50s	Average 300s; Std 50s	Average 300s; Std 50s
High-CPU ($0.17/h;720s delay)	Average 210s; Std 25s	Average 75s; Std 15s	Average 300s; Std 50s
High-I/O ($0.17/h;720s delay)	Average 210s; Std 25s	Average 300s; Std 50s	Average 75s; Std 15s

Table 3.2: Job execution time on different VMs

3.3.1 Deadline

For the deadline performance goal, this dissertation considers two cases. The first case is stable workload with a changing deadline. It generates the workload based on table 3.2 and plots the job turnaround time in figure 3.3. Every data point in the graph reflects the job turnaround time information for every 5 minutes and the average, minimum and maximum turnaround times for all the jobs finished in that interval are recorded. The deadline is first set at 3600s, then changed to 5400s and finally switched back. The purpose is to evaluate the mechanism's reaction to dynamic user performance requirement changes. Figure 3.3 shows that more than 95% of the jobs are finished within the deadline and most of the misses happen at the second deadline change. This is mainly because the auto-scaling mechanism runs every 5 minutes and VM instances can only be ready 10-12 minutes later after the acquisition requests. In addition, the instantaneous instance utilization rate is also calculated. Job processing is considered utilized while all the other cases, such as VM pending and idling, are considered unutilized. The high utilization rate (average 94%) shows that the auto-scaling mechanism does not aggressively acquire instances to guarantee the deadline, and 6% of the time is spent on VM startups.

The second case is changing workload with a fixed deadline. In this test, the deadline is fixed at 3600s and three workload peaks are created. Base workload is 30 mix jobs per hour. The first workload peak adds another 300 mix jobs per hour. The second peak adds 300 compute-intensive jobs per hour, and the third one adds 300 data-intensive jobs per hour. The purpose of this test is to evaluate the auto-scaling mechanism's reaction to a sudden workload increase and job type change. Such a workload pattern is normally seen in large volume data processing applications, in which data computation and analysis is performed in the daytime, and data backups and movements are performed in the nights and on holidays. From figure 3.4, it shows that the deadline goal is well met for all three workload peaks. When the workload goes back to normal, the over-acquired instances during peak moments quickly reduce the job turnaround time. As more and more unnecessary instances are shutdown (approaching full hour operation), the job turnaround time goes back to the performance goal.

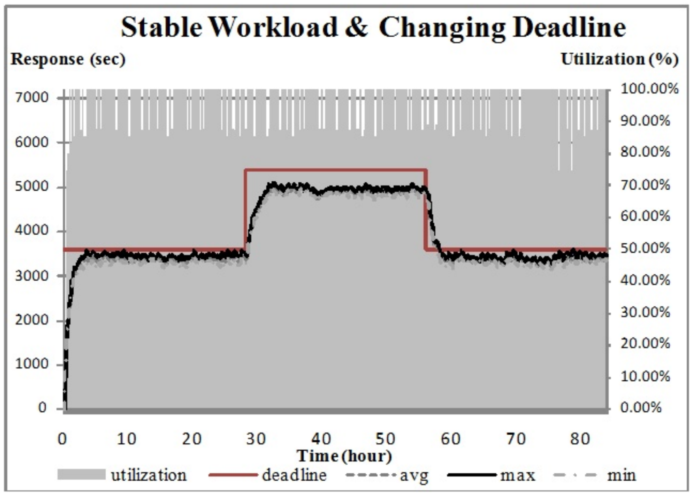

Figure 3.3: Stable workload with changing deadline

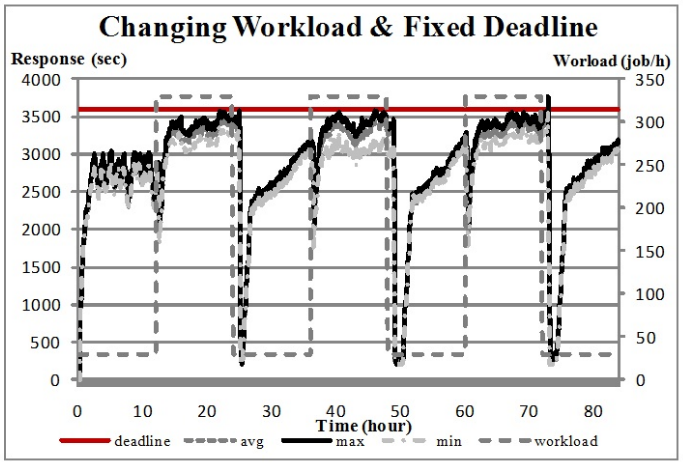

Figure 3.4: Changing workload with fixed deadline

3.3.2 Cost

Based on the same evaluation settings for the test of changing workload with fixed deadlines, this dissertation compares the cost of using different VM types. For example, choice #1 only uses General VM type, while choice #4 uses all the three VM types. The VM type combinations are illustrated in table 3.3. Figure 3.5 and figure 3.6 show the comparison results.

	VM Types	Total cost $ (% more than optimal)
Choice #1	General	98.52$ (43%)
Choice #2	High-CPU	128.86$ (87%)
Choice #3	High-I/O	129.71$ (88%)
Choice #4	General, High-CPU, High-I/O	78.62$ (14%)
Optimal	General, High-CPU, High-I/O	68.85$

Table 3.3: Cost of different VM choices

The theoretical optimal solution (essentially the theoretical lower bound) can be obtained by assuming the workload is known in advance and a job is always assigned to the most cost-efficient machines, e.g., compute-intensive jobs are assigned on High-CPU instances for processing. Figure 3.5 shows that by considering all available instance types (Choice #4), the auto-scaling mechanism can adapt to the workload changes and choose cost-efficient instances. In this way, the real-time cost is always close to the optimal. General instances always show average performance for all the three workload peaks, while High-CPU and High-I/O choices can only save cost on their preferred workload surges. Figure 3.6 shows the accumulated cost. Choice #4 incurs 14% more cost than the optimal solution and saves 20% compared to the General instance choice, 45% compared to the High-CPU and High-I/O only choices. Because of symmetry, High-CPU and High-I/O instances end up with almost the same cost. The General instance has lower cost on average, so in the long run, it outperforms High-CPU and High-I/O instances. By choosing appropriate instance types, choice #4 can incur less cost in all three workload peaks like the optimal solution. So it outperforms all the other choices. There are two reasons why the auto-scaling solution cannot make the optimal decisions. The auto-scaling mechanism does not know the future workload in advance and can only make local optimized decisions. Second, it cannot control the running instance for job processing (i.e. job scheduling).

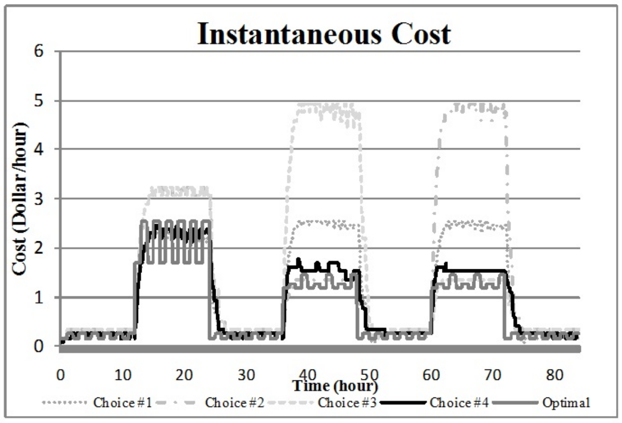

Figure 3.5: Instantaneous cost of changing workload & fixed deadline

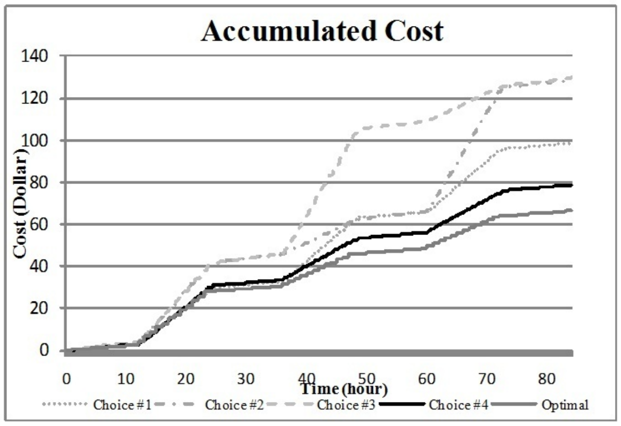

Figure 3.6: Accumulated cost of changing workload & fixed deadline

3.3.3 MODISAzure

In addition to simulations, this dissertation has applied the auto-scaling mechanism to a real-life scientific application MODISAzure [22] running in the cloud. MODISAzure is a cloud application built in Windows Azure [13] platform for large volume biophysical data processing. It integrates data from ground-based sensors with the Moderate Resolution Imaging Spectroradiometer satellite data. It is now used by biometeorology lab, UC Berkeley. This section first introduces the MODISAzure workload and the configuration parameters in the auto-scaling mechanism. MODISAzure workload can be understood in the following way. 200X indicates the year, Terra and Aqua represent satellite images, and (x-y) represents the period from day x to day y. For all the tests, all the available 15 tile images for a single day data processing are used. For example, Terra 2004 (10-12) means processing all 15 tiles of Terra images from 2004 Jan 10th to Jan 12th. This implies that totally 45 (15×3) jobs are submitted at once. In the evaluation, the actual job execution time ranges from 10 sec to 13 min (average 5 min) and all jobs are processed most cost-effectively on the small instances. The performance monitor interval is set as 1 min, the decider interval is set as 5 min and the initial average VM startup delay is set as 15 min.

The MODISAzure evaluation includes both moderate scale (up to 20 instances) and large scale (up to 90 instances) tests. In the moderate scale evaluation, two test cases are randomly selected. One is Terra satellite 2004 (10-12) and the other one is Aqua 2008 (30-32). The test results are shown in table 3.4, including both the performance and the instance hours consumed (i.e. cost). The table shows that 2- and 3-hour deadline goals are better met than 1 hour deadline for the same workloads. After investigating the VM instance startup history, this dissertation finds that this is largely because the actual VM startup delay is out of expectation. For example, in 1 hour deadline tests, the average startup delay is around 22 minutes. Some instances even take 50 minutes to be ready. There is little time left for the auto-scaling mechanism to react in such cases. On the contrary, in the longer deadline tests, the mechanism acquires fewer instances and hence the result is less affected by the startup delay variances. In both test cases, the theoretical computing power needed is 4 instance hours (all jobs are processed by a single instance). All tests actually acquire more than this, e.g. 9 or 10 instance hours for the 1-hour deadline tests. This is caused by the VM startup delay make up and inaccuracy of the initial job execution time configuration. With longer deadlines, such over acquisition is corrected because fewer instances are acquired and the job execution time is also updated by the historical table. Therefore, longer deadline tests incur less cost.

	1-hour deadline	2-hour deadline	3-hour deadline
Terra 2004(10-12) Total 45 jobs; 4C.H.* or 0.48$	18 min late; 9C.H. or 1.08$	8 min early; 6C.H. or 0.72$	20 min early; 5C.H. or 0.6$
Aqua 2008(30-32) Total 45 jobs; 4C.H. or 0.48$	15 min late; 10C.H. or 1.2$	20 min early; 7C.H. or 0.84$	29 min early; 5C.H. or 0.6$
* C.H. - compute hours (or instance hours)			

Table 3.4: MODISAzure moderate scale evaluation

For the large scale (up to 90 instances) MODISAzure evaluation, two tests are performed and the results are shown in table 3.5. Similar to the moderate scale evaluation, longer deadline tests show better results. Again, unexpected VM startup delay is the dominating factor. Windows Azure shows longer VM startup delay and larger variances in the cases of large number of instance acquisitions. For example, in the Terra & Aqua 2006 (1-75) 2-hour deadline test, the average VM startup delay is 40 minutes and there's one instance which is still not ready 2 hours later. For the 2006 (1-125) 2-hour deadline test, the decider calculation shows 95 instances are needed, which is beyond the test resource limit. This job is successfully identified and denied.

	2-hour deadline	4-hour deadline
Terra and Aqua 2006(1-75) Total 1125 jobs 93C.H.* or 11.16$	20 min late 170C.H. or 20.4$	6 min early 132C.H. or 15.84$
Terra and Aqua 2006(1-150) Total 2250 jobs 185C.H.* or 22.2$	Admission denied	22 min early 243C.H. or 29.16$
* C.H. - compute hours (or instance hours)		

Table 3.5: MODISAzure large scale evaluation

To better demonstrate the auto-scaling mechanism's working details, this dissertation presents the instance acquisition and release information for the test case - Terra & Aqua 2006 (1-75) 4-hour deadline in figure 3.7. This test includes 1125 jobs in total and is submitted at time 0. As shown in the figure, after around 4 minutes, the decider started 34 instances (instance 1 - 34) to handle the workload. The real instance acquisition time took much longer than configured. Therefore, around 1.5 hours later, the decider started another 6 instances (instance 35 - 40) to make up for such unexpected startup delay. After approaching 2 full hour operation, these 6 instances were shutdown due to the decreased workload. After all jobs are finished, instance 1 to instance 34 were shutdown when they approached 4 hour operation. At that time, only instance 0 was kept alive to maintain the service availability. In this case, the theoretical computing power needed is 93 instance hours. The actual computing power consumed is 132 hours with 36 hours spent on the VM startup time. Both moderate and large scale tests show that longer deadlines have better performance and incur less cost. This is because longer deadline tests are less affected by the VM startup delay and have more chances to use the updated job execution time.

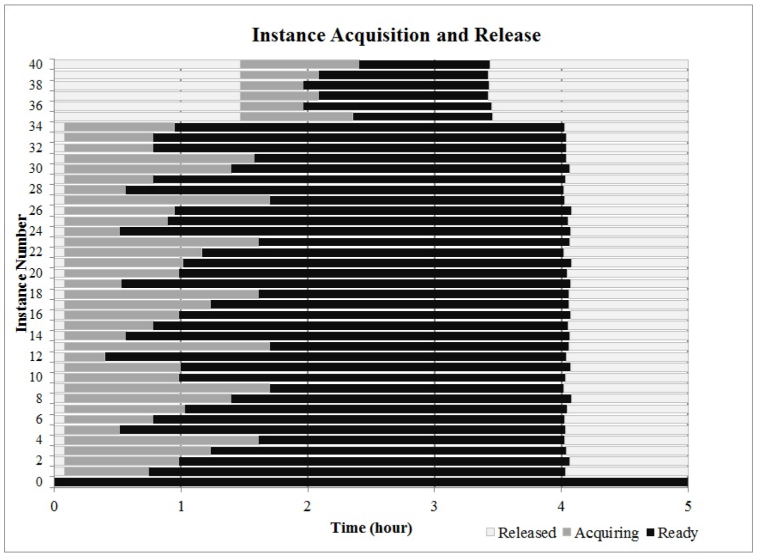

Figure 3.7: Instance acquisition and release (Terra & Aqua 2006 (1-75) 4-hour deadline)

3.4 Conclusion

This chapter presents an integer programming based auto-scaling solution to solve the deadline-constrained resource provisioning problem for batch-queue based cloud applications. The key idea is to make sure the computing power is always large enough to handle the workload and convert it to an integer programming problem with cost minimization as the goal and job deadline as the constraint. The results show that performance- and cost-aware cloud auto-scaling mechanisms can help service providers to achieve the desired performance while reducing the cloud spending. Choosing cost-efficient instances that can adapt to workload changes is a useful technique to save cost. In the changing workload and fixed deadline test, it saves 20.2% - 40.1% cost compared to a fixed machine type choice. However, the weaknesses of the mechanism are also acknowledged: 1) Only a single queue and a single job deadline are supported. These restrictions limit the types of the applications this mechanism can be applied to. 2) The auto-scaling mechanism cannot control the instances on which jobs are executed. Job scheduling is not supported. In other words the solution only answers the ”capacity determination” question not the ”resource allocation” question. This work has been published in the 2010 11th IEEE/ACM international conference on grid computing (Grid 2010) [20]. The next chapter presents an auto-scaling solution for the workflow application model with individual job deadlines and explicitly answers the job scheduling question.

Chapter 4

Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows

The previous chapter introduces a batch-queue based cloud application model and presents an auto-scaling solution using the integer programming technique. The auto-scaling mechanism calculates the instance computing power based on the ratio of the deadline over the total execution time, and acquires machines if the computing power is less than the workload. The results have shown that performance- and cost-aware cloud auto-scaling mechanisms can help users to achieve desired performance goals while reducing the cloud spending. Choosing cost-efficient instances that can adapt to the workload changes is a useful technique to save cost. However, the weaknesses of the previous model are also acknowledged:

- The workload is bags-of-tasks in which jobs do not have dependencies. All the jobs share the same deadline (from its submission time to the completion time). These restrictions limit the types of the cloud applications the auto-scaling mechanism can be applied to.

- The auto-scaling mechanism cannot control the instances on which jobs are executed. In other words, the second question in the resource provisioning and allocation problem - job scheduling - is not answered.

- More general and representative cloud applications and workload patterns are needed to evaluate the auto-scaling mechanism and cover a larger problem space.

One limitation of the previous chapter is that the application architecture is based on a single queue and the workload is bags-of-tasks. For many applications, jobs have much more complex structures. They may share the same service components and have different execution paths. Essentially, the applications are based on the Service Oriented Architecture (SOA) and the jobs can be treated as workflows [23]. A workflow consists of a sequence of connected steps. It is normally represented as a direct acyclic graph (A specific example will be shown later). Sometimes, one or more service components could be the potential performance bottlenecks and cannot handle the workload in time. In the cloud, these service components become the basic scaling units. More machines can be acquired to run the services to increase the processing power. In addition, the workflow jobs may have individual deadlines depending on the user preferences and job characteristics. The auto-scaling solution presented in this chapter will deal with all these requirements.

4.1 The Cloud Application Model Based on Service Oriented Architecture

4.1.1 Assumption

- Cloud application, job and task
  - A cloud application consists of several function service components and a job is composed of sub tasks (simply referred as tasks from now on) with precedence constraints. All jobs are submitted into the entry service component. They flow from one service component to another. In other words, jobs may share the same service components, but have different execution paths and therefore consist of different tasks (a task can be considered as one execution instance of a service component).
  - Jobs are not dependent on each other. However, tasks belonging to the same job must be processed based on the output-input dependencies (execution flow). Together with the previous assumption, this assumption extends the cloud application from the batch-queue model to the SOA model. The workload is workflows instead of bags-of-tasks.
  - Based on the execution paths, jobs can be categorized into different classes that may have different levels of importance and urgency. Service customers can specify different deadlines for different job classes.
  - Every service component can be considered as a queue that holds incoming tasks. Service components can be classified as compute-intensive, data-intensive or neither (or even more fine-grained). Thus, tasks may perform differently on different VM types.

    - Based on the task property and deadline requirements, tasks can be reordered and redistributed on different types of instances for processing. This is an important difference from the previous chapter and it implies that job scheduling is now explicitly considered as a separate process.

- Cloud resources and pricing

    - The cloud provider offers several types of VMs with different hardware configurations (CPU, memory, disk, etc.). Service providers can choose appropriate VMs based on their needs (e.g. workload). Cloud VMs are billed based on the cost per time quantum scheme. In current real-world practice, VM instances are priced by hours. Partial instance hour consumption is always rounded up as one hour [6][11][13]. In addition VM instances are not necessarily priced linearly to the processing power. In other words, for a particular task, a more expensive machine does not mean a faster machine. Though VMs can be acquired at any time, it may take some time for the instances to be ready to use (cloud VM startup delay [39][76]).
    - There could be different classes of tasks, such as compute-intensive and data-intensive tasks. This property is inherited from the service component where it is processed. A task may have different execution times on different instance types. For example, a compute-intensive task can run faster on a high-CPU machine than on a standard machine. In other words, a task may prefer an expensive VM type (high hourly cost) compared to a cheap VM type (low hourly cost) when considering the greatly reduced execution time. Further, this dissertation assumes that the auto-scaling mechanism can estimate the task execution time on different types of VMs in advance.

Figure 4.1 shows one example of a SOA based cloud application, which consists of 11 service components. The application background is introduced as the second motivating example in chapter 1. An insurance company offers online quotes to its customers, categorized as non-members, silver members and gold members. Non-member jobs consist of the data validation and quote calculation steps using a very basic model, while silver and gold member jobs go through more data collection steps, such as the credit history and health record collection modules, and are evaluated based on more complex risk assessment models. These requests share some service components but have different execution paths. They are described as workflows. The requests from potential customers are given fast responses to attract more business, and the requests from existing customers may take longer but are offered more accurate results. The requests therefore have different deadlines. The data collection steps are data-intensive tasks that run more cost-efficiently on high-I/O instances, while the model building and premium calculation steps are compute-intensive tasks that run more cost-efficiently on high-CPU machines. There could be other job classes (not shown in the figure), such as the daily and weekly report generation requested by the operation department, and the data mining or sales trend analysis performed by the research department. This cloud application needs an auto-scaling mechanism to help acquire instances and schedule the workload. The goal is to meet job deadlines and minimize the cloud cost, therefore to maximize their profits from cloud adoption.

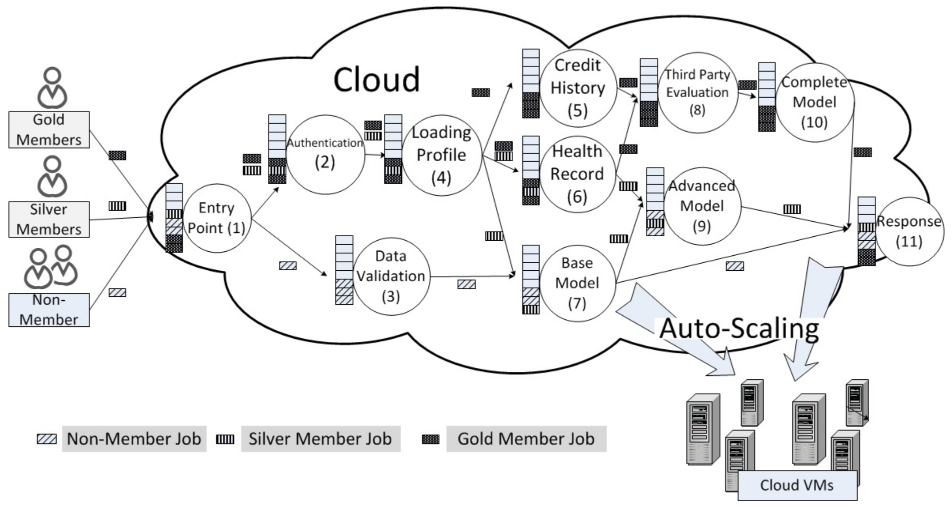

Figure 4.1: The SOA based cloud application model

4.1.2 Problem Definition

This section formally defines the cloud application, cloud resources and the auto-scaling problem.

Definition 1 - Cloud Application. A cloud application consists of a set of service components (equation 4.1). A service component is an abstraction of a processing module/component in an application, such as the order submission and data persistence steps for an online shopping site, or the data reduction and compression steps in a scientific workflow. Without loss of generality, all the jobs are assumed to be submitted into the entry service unit S_0.

$$app = \{S_i\} \tag{4.1}$$

Definition 2 - Job Class. A job class J_j includes two properties (equation 4.2). The first one is a direct acyclic graph (DAG) which describes its workflow, such as the insurance quote request for the gold members in the above example. A job is an execution instance of a job class. It can be considered as a set of tasks with precedence constraints and each task is processed at some service component. The second property of a job class is the deadline property in the form of time duration, e.g. one hour, from the time a job instance is submitted to the time it should be finished.

$$J_j = \{DAG_j(S_i), deadline_j | S_i \in app\} \tag{4.2}$$

For example, the job class of non-member insurance quotes can be defined as $\{(S_1,S_3)(S_3,S_7)(S_7,S_{11})$, 1 min$\}$, which means the request should go through the entry point, data validation, basic model and response steps within 1 minute. In the application, there could be several job instances of the same job classes submitted. In the following text, this dissertation uses job_{J_j} to represent an instance of job class J_j and $job_{J_j}^{S_i}$ to represent the task at service component S_i.

Definition 3 - Cloud VM. The cloud provider may offer different types of VM instances that are suitable for different types of workload. These VMs have different processing power (the number of cores, memory, disk space etc.) and prices (e.g. ranging from \$0.085/hour to \$2.10/hour for Amazon EC2 [6]). A VM type is defined with three components - the estimated processing time for each service component, the cost per time quantum (e.g. 1 hour) and the estimated instance acquisition lag (equation 4.3). It is assumed that there is no performance inference among VMs.

$$VM_v = \{[t_v^{S_i}], c_v, lag_v\} \tag{4.3}$$

The performance component $[t_v^{S_i}]$ is a vector, in which each element is the estimated processing time for tasks from service component S_i. Such task execution time on different types of VMs can be estimated by using existing performance estimation techniques [49] (e.g. analytical modeling [77], empirical [78] and historical data [79]). Therefore, the cost of running task $job_{J_j}^{S_i}$ on VM_v is calculated as $t_v^{S_i} \times c_v$.

Definition 4 - Workload. It is assumed that the service provider does not know the incoming service requests in advance. Therefore, the workload is defined as all the jobs that have been submitted into the application. They are either waiting to be processed or partially processed. At some time t, the tasks waiting at every service component are used to represent the workload W_t (equation 4.4).

$$W_t = \sum_{S_i} \sum_{J_j} \sum_{job} job_{J_j}^{S_i} \tag{4.4}$$

Definition 5 - Goal. The goal of the auto-scaling mechanism is to minimize the total running cost C in the observation period. One important difference of this dissertation from the related research is that the target is a continuously running cloud application instead of a single job or job ensembles. It is not practical to define or identify the start and end of the service life cycle. Therefore, the cost is the accumulated cost in the observation period (like the quarterly financial reports), which is the total cost of all the acquired instance hours (equation 4.5).

$$Min(C) = Min(\sum_{v} c_v N_v) \tag{4.5}$$

Definition 6 - Constraint. The constraint is to finish all the submitted jobs before their deadlines (equation 4.6). Deadlines are soft deadlines. They are not hard deadlines as in the real-time system. Deadline misses are allowed. They are considered as indicators to reflect the execution speed of the cloud application or the user's preference on the job completion time.

$$\forall job_{J_j}, turnaround(job_{J_j}) < deadline(J_j) \tag{4.6}$$

Definition 7 - Output. The auto-scaling mechanism needs to make two decisions. The first decision is to determine the number of instances for each instance type VM_v at some time point t (i.e. scaling plan, equation 4.7) and the second decision is to determine the execution instance for each task at some time point t (i.e. scheduling plan, equation 4.8).

$$Scaling_t = \{VM_v \rightarrow N_v\} \tag{4.7}$$

$$Schedule_t = \{job_{J_j}^{S_i} \rightarrow VM_v^i\} \tag{4.8}$$

4.2 Solution

The cloud application continuously accepts jobs submitted by the service customers, and the workload is changing all the time. Therefore, the auto-scaling mechanism needs to keep monitoring the dynamic workload information as well as the progress of submitted jobs, and then make fast scaling and scheduling responses. This is a repeated process instead of a one-time process. Therefore, the solution is based on a monitor-control loop. Every time inside the loop, a scaling decision and a scheduling decision are made based on the latest updated information. Because cloud VMs are currently billed by instance hours (not by the exact consumption time), the scaling and scheduling decisions should avoid partial instance hour waste. Moreover, unlike the fixed-size resource environment [80][81][82][83][84], as long as there is unhandled workload, the auto-scaling mechanism can acquire a VM instance and place a task on it.

As shown in the remainder of this section, this dissertation makes the scaling and scheduling decisions step by step. It first calculates the number of VMs needed for each VM type based on the workload. Next, it determines if two or more existing VMs can be consolidated. It finally schedules tasks on the active VMs using the Earliest Deadline First (EDF) algorithm.

4.2.1 Preprocessing

To reduce the runtime overhead of the auto-scaling mechanism and accelerate dynamic scaling plan generation, the auto-scaling mechanism pre-analyzes the job classes and calculates deadlines for each sub task using the following techniques.

Step 1 - Task bundling. Task bundling treats adjacent tasks that prefer the same instance type as one task and forces them to run on the same instance. Therefore, it can save data transfers by using the temporary results stored locally. Figure 4.2 shows one example of task bundling. Both Task 6 and Task 8 run most cost-efficiently on high-CPU machines, so the auto-scaling mechanism will treat these two tasks as a single task Task 6'. In this chapter, it only bundles the tasks that prefer the same type of instances and have the one-to-one task dependencies. The trade-off between large data movements and different VM type choices, as well as complex task dependencies, will be considered in chapter 6.

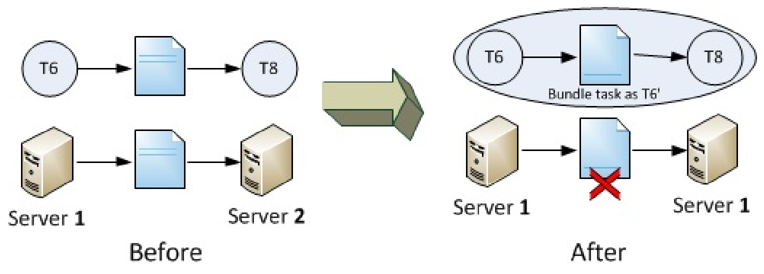

Figure 4.2: Task bundling

Step 2 - Deadline assignment. Deadlines are associated with jobs not tasks. When a job is submitted, deadlines are further assigned to individual sub tasks. If every task can be finished by its assigned deadline, then the job will be finished within its deadline. Deadline assignment in a DAG is introduced and detailed in [49]. A simple strategy is to calculate the shortest job makespan by assuming all the tasks do not wait for resources and then extend the task execution time by the ratio of deadline over the job makespan. In [49], they assign deadlines proportionally by the fastest task execution time and then search for the cheapest service for each task. This dissertation however assigns individual deadlines based on the task processing time on their most cost-efficient machines. The reason is that the cloud VM is not necessarily priced linearly to its processing power and a more expensive machine does not mean a faster machine. If the initial deadline assignment cannot make the job finish in time, this dissertation further uses a heuristic introduced by [48] to locate a feasible plan. The idea is to calculate the job makespan for the initial deadline assignment, and if the job can be finished within the deadline, the process stops. If not, it tries to schedule each task on a faster but more expensive machine and calculate the new job makespan to see which task upgrading reduces the makespan the most with the same amount of money. In other words, it upgrades the task with the highest cost-efficiency rank (equation 4.9) to a faster machine. It repeats such process until the job makespan is within the deadline. Through the deadline assignment process, the auto-scaling mechanism breaks the task precedence constraints and treats each task independently. Figure 4.3 shows one example of deadline assignment.

$$Cost\_Efficiency\_Rank = \frac{makespan_{before} - makespan_{after}}{cost_{after} - cost_{before}} \tag{4.9}$$

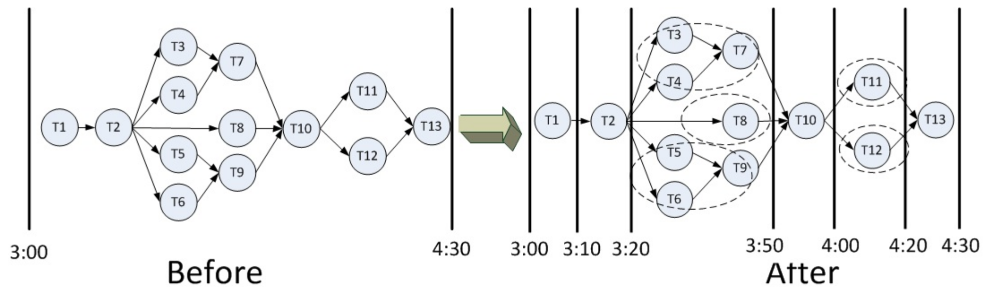

Figure 4.3: Deadline assignment

In this step, some other techniques can be used to further tailor this approach to accommodate the cloud characteristics. Unlike the utility computing environment [48][50][49], as long as there are services available, the scheduling mechanism can reserve a time slot and place the task on the service. In the cloud, although there are unlimited resources and a service provider can acquire an instance at any time, however, it may not be always wise to do so whenever a task needs to be processed, especially when the task could waste a significant portion of a purchased instance hour. Therefore, reducing task concurrency is a way to improve instance utilization rate and reduce cost. Figure 4.4 illustrates this idea. Assuming tasks T3, T4 and T5 are tasks shorter than one hour, by processing them sequentially instead of in parallel, three instance hours can be saved. It uses breadth-first search to combine parallel tasks, and the search stops when some task has to change its originally scheduled machine type to finish before deadline. It stops searching because this strategy is only a job level optimization, and it should not affect the global scheduling decisions. The overall deadline assignment algorithm is shown in algorithm 1.

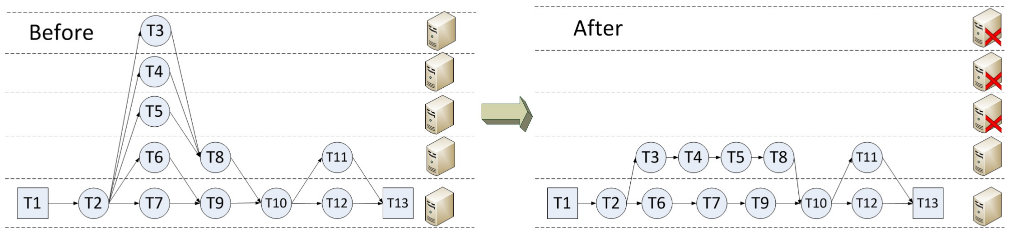

Figure 4.4: Parallelism reduction

Algorithm 1: Deadline assignment

```
Result: Schedule plan $S = task_i \rightarrow VM_v$
Generate the cheapest schedule $S = task_i \rightarrow VM_{cheapest}$ ;
while true do
    if makespan(S) < deadline then
        return S;
    else
        for each $task_i$ do
            $S_i = S - (task_i \rightarrow VM_v) + (task_i \rightarrow nextFasterVM(task_i))$;
            $SpeedUp_i = (makespan(S) - makespan(S_i))/(cost(S_i) - cost(S))$;
        end
        $index = subscript(max(SpeedUp_i))$ ;
        $S = S_{index}$;
    end
end
```

4.2.2 Dynamic Scaling-consolidation-scheduling

Inside each monitor-control loop, the dynamic scheduling and scaling decisions are made using the most updated information. The auto-scaling mechanism recalculates task deadlines to determine the instance number, consolidate partial instance hours and schedules tasks using the earliest-deadline-first algorithm (EDF).

Step 3 - Scaling. To determine the number of VMs, this dissertation introduces a metric called *load vector*.

Definition 7 - Load Vector. A load vector (LV) is defined for each task. After deadline assignment, an execution interval $[T_0, T_1]$ is scheduled for each task, and the task execution time on VM_v is t_v. Therefore, the task's load vector LV_v is defined as $[t_v/(T_1 - T_0)]$ indexed from T_0 to T_1. Intuitively, the vector indicates the number of the machines needed to finish the task on VM_v between the time interval. Because a task cannot be further divided and run in parallel, if the ratio is greater than 1, the task cannot be finished in time. For example, as shown in figure 4.5, assuming the execution interval for task T1 is from 3:00PM to 4:00PM, and it is estimated to run for 15 min on VM_1. It means that the task needs to use 1/4 instance hour on VM_1 between 3:00PM and 4:00PM. Another task T2 needs to be finished between 3:15PM and 3:45PM and also consumes 15 min on VM_1. In total, the auto-scaling mechanism can use one instance to process both T1 and T2. The finest granularity is 1 minute when defining the load vector.

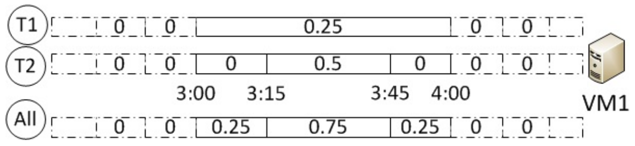

Figure 4.5: Load vector

The auto-scaling mechanism calculates the load vector for each task and adds them together. Every VM type has one load vector and one task can only contribute to the load vector of the scheduled machine type. If the auto-scaling mechanism can ensure that the number of existing machines is always greater than or equal to the load vector at any time, all the tasks will finish within the assigned execution interval. Because cloud VMs may take some time to be ready to use, for scaling-up cases, task load vectors are calculated between interval $[T_0 + lag_v, , T_1]$ instead of $[T_0, T_1]$. Scaling-down decisions are also based on the load vector. The auto-scaling mechanism is aware of the acquisition time of every instance. When one instance is approaching full hour operation and the number of instances is greater than the load vector, it can be shutdown. VM churn is another factor which may affect the decision of shutting down an instance. Too frequent VM acquisitions/releases will degrade the performance of the auto-scaling mechanism if the instance acquisition lag cannot be accurately estimated. The effects of inaccurate parameter estimation are shown in the evaluation section. Note deadline assignments need to be recalculated periodically, because some tasks may be finished earlier than their original assigned deadlines and reassignment can allow later tasks to run on cheaper machines.

Step 4 - Instance consolidation. It is optimal if all tasks can be executed on their most cost-efficient instances and all instances are fully utilized. However, it is not always feasible to make sure there are no wasted partial instance hours when considering the arrival times and execution times of the tasks. Sometimes, a decision needs to be made to run tasks on their non-cost-efficient machines to consolidate partial instance hours, because consolidating instance hours can help users to reduce the overall cloud cost. This process is called instance consolidation. Figure 4.6 below illustrates this idea. T11 and T12 originally are scheduled on a high-CPU instance and a standard instance, respectively. Because both tasks only consume a partial instance hour and there are no other tasks sharing the instances at the same time, a smart scheduling decision is to consolidate the two tasks on the same standard machine and save one high-CPU instance hour (although T11 runs slower and costs more on a standard machine). Of course, a "consolidated task" must still be finished before its original deadline. The process is described in algorithm 2.

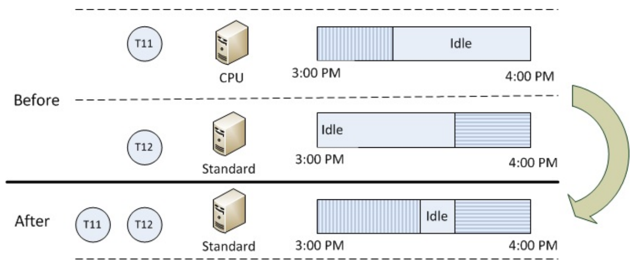

Figure 4.6: Instance consolidation

Algorithm 2: Instance consolidation

```
Data: LoadVector $LV_v$ and exiting instance number $N_v$ for each VM type $VM_v$
Result: Updated LoadVector $LV_v$ after instance consolidation
for each $VM_v$ whose $LV_v > N_v$ do
    for each $VM_n$ whose $LV_n <= N_n$ do
        if $InstanceNum(LV_n + LV_n(TopTask[VM_v])) \leq N_n$   AND   tasks following TopTask[$VM_v$]
        do not change scheduled VM then
            $LV_v$ - $LV_v$(TopTask[$VM_v$]);
            $LV_n$ - $LV_n$(TopTask[$VM_v$]);
            schedule TopTask[$VM_v$] to $VM_n$ instances;
        end
    end
end
```

Step 5 - Dynamic scheduling. After determining the number of instances for each VM type, the auto-scaling mechanism uses the Earliest Deadline First (EDF) algorithm to schedule tasks on each VM type. After deadline assignment and instance consolidation, every task is scheduled to a VM type. Tasks are sorted by their deadlines for each VM type, and the task with the earliest deadline is scheduled when an instance becomes available. Through dynamic scaling, the task facing deadline misses can be found in time and the auto-scaling mechanism can immediately acquire instances to execute the task. In other words, dynamic scaling ensures that the load vector is always less than 1 for every instance type (equation 4.10). It is known that EDF is the optimal scheduling strategy in such cases [85]. Therefore, all the tasks will finish before assigned deadlines and so is the whole workflow job. Algorithm 3 describes the overall auto-scaling solution.

$$\sum_i \frac{t_i}{T_{end\_i} - T_{start\_i}} < 1 \tag{4.10}$$

Algorithm 3: Auto-scaling

```
while $true$ do
    update $LV_v$ for each task $t_i$;
    get existing $N_v$ for each VM type $VM_v$;
    $LV_v \leftarrow$ InstanceConsolidation($LV_v$ , $N_v$);
    for $each\ VM_v\ where\ NumNeeded(LV_v) > N_v$ do
        acquire($VM_v$ , $NumNeeded(LV_v) - N_v$);
    end
    for $each\ VM_v\ where\ NumNeeded(LV_v) < N_v$ do
        for $each\ instance\ I_i\ of\ type\ VM_v$ do
            if $I_i$ approaches full hour operation  AND  $NumShutdown < N_v - NumNeeded(LV_v)$
            then
                $shutdown(I_i)$;
                $NumShutdown++$;
            end
        end
    end
    schedule the tasks based on EDF;
    wait for the next monitor interval;
end
```

4.3 Evaluation

This dissertation evaluates the presented auto-scaling mechanism using three types of applications with four workload patterns. Using simulation helps to control the input parameters and locate the key factors in the auto-scaling mechanism. It also helps to analyze the relationship between the mechanism performance and the budget constraints. Moreover, it speeds up the evaluation process and saves the evaluation cost. This dissertation first compares with two baseline approaches on cost and instance utilization. Then, it analyzes the effects of the workload volume and the mechanism's capability to handle inaccurate input parameters, (e.g. estimated task running time and the instance acquisition lag). Finally, it evaluates the mechanism's overhead.

4.3.1 Application, Workload and VM

The three types of representative applications are Pipeline, Parallel and Hybrid. Pipeline applications are simple multiple stage applications in which tasks need to be processed one by one with precedence constraints. Parallel applications feature a high degree of potential concurrency, because there are only limited precedence constraints. Hybrid applications are mix of pipeline applications and parallel applications. Task dependencies in Hybrid applications can be very complex. Figure 4.7 illustrates the pipeline application, the parallel application and the hybrid application used in the evaluation. They are adopted from [49].

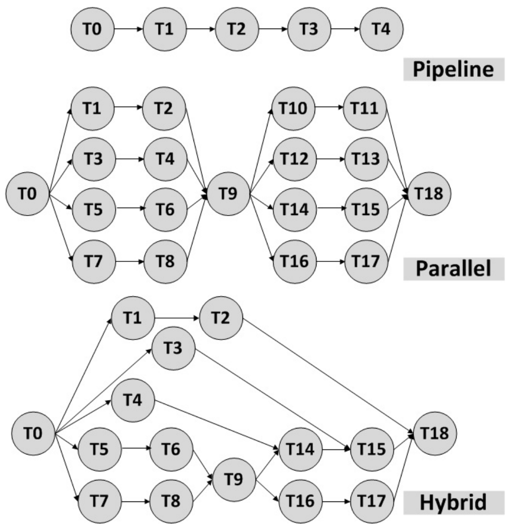

Figure 4.7: Application models

The four representative workload patterns (figure 4.8) in the cloud environment are Stable, Growing, Cycle/Bursting and On-and-Off [9][86]. Each of these workloads represents a typical application or scenario. For example, the Growing workload pattern represents a scenario in which a news or video suddenly becomes popular and brings in more and more users to hit the button. The workload keeps increasing very fast. The Cyclic/Bursting workload represents the workload pattern of an online retailer. Daytime has more workload than the night and holiday shopping seasons may handle more traffic than normal. The On-and-Off workload pattern represents the work to be processed periodically or occasionally, such as batch processing and data analysis performed daily or weekly in a research department. These applications have relatively short active period, after which the service can be switched off or be maintained at the lowest service level. In the evaluation, the task execution time on different types of VMs are randomly generated (the task execution time and distribution are generated based on [87][88]) and 200 combinations are generated for each workload pattern. All the test results have shown similar performance. Therefore the information of the task running time is not detailed here. For each application and workload pattern, this dissertation simulates a 72-hour period for each test with four different deadlines - 0.5 hour, 1 hour, 1.5 hour and 2 hour. The accumulated cost in the observation period is recorded.

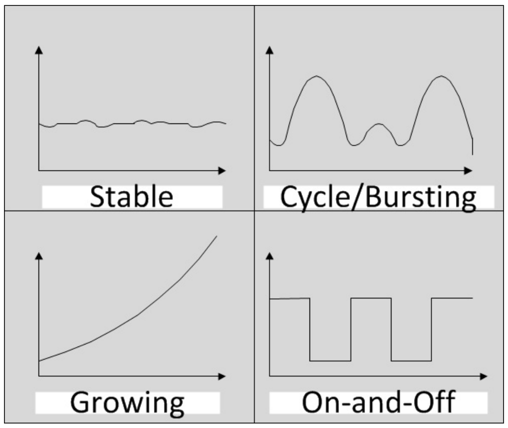

Figure 4.8: Workload patterns

In addition to the three application models and four workload patterns, this dissertation simulates four types of VMs: Micro, Standard, high-CPU and high-I/O instances. The prices (table 4.1) of these VMs are borrowed from Amazon EC2 [6].

VM Type	Price
Micro	\$0.02/hour
Standard	\$0.085/hour
High-CPU	\$0.68/hour
High-I/O	\$0.50/hour

Table 4.1: VM types and prices

4.3.2 Cost and Resource Utilization

It is a challenge to establish a baseline to compare with this dissertation. Existing research either targets a batch-queue model (which does not handle task dependencies) or a single workflow instance (not a stream of submitted jobs). Moreover, cost-efficiency (deadline and cost) is usually not a first-class concern. Although the rule-based trigger mechanism is a reasonable baseline. However, it is really hard to determine the scaling indicators and thresholds by this dissertation itself. As argued in chapter 2, the trigger mechanism does not really solve the performance-resource mapping problem and any rules this dissertation picks could result in the risk of an “unfair game”.

Therefore, this dissertation decides to extend two existing approaches, Greedy [49] and GAIN [48], as the baseline. This section compares the approach presented in the previous section (denoted as Scaling-Consolidation-Scheduling (SCS)) with these two approaches. These two approaches are originally designed for the cost-aware single workflow execution in the utility computing environment. This dissertation extends these two approaches to enable them to support continuous workflow submissions and make them aware of the instance hour billing model in the cloud environment. First, they treat each submitted job independently using their original algorithms. They reserve instances for each job and acquire more instances when the number of active instances is insufficient. Second, acquired instances can only be released when they approach full hour operation and no jobs need them any more. In this way, all job requests are served in a timely manner and the number of wasted partial instance hours is reduced as well. Note, the original GAIN approach starts with the cheapest plan and iteratively improves the plan based on the cost-efficiency rank until the budget cap is reached. This dissertation changes the break condition so that the estimated job finishing time is before the deadline. For the Greedy algorithm, it always tries to find the cheapest instance among all the live instances first. If there are not sufficient instances, it then acquires the cheapest instance from all the available instance types.

The test results with different workload patterns are shown in figure 4.9, figure 4.10, and figure 4.11. For each application model and deadline, this dissertation shows the total running cost and the average instance utilization. In all the cases, it assumes the task running time and instance acquisition lag (6.5 min) can be accurately estimated. Therefore, SCS, Greedy and GAIN all can finish the jobs before user specified deadlines. As shown in the figures, for almost all the cases, SCS incurs the least cost and has the highest instance utilization (higher utilization implies fewer idle instance hours) compared to the Greedy and GAIN approaches. The cost savings ranges from 9.8% to 40.4%.

When the deadline is short, these three approaches tend to have similar performance, because all tasks are forced to run on their fastest machines to finish the job within the deadline. In other words, all three approaches generate very similar scheduling/scaling plans and there is not much space for cost saving optimization. When the deadline is longer (the “scheduling slack time” is greater), SCS can save the most cost, the Greedy approach performs the worst and the GAIN approach performs in between. The Greedy approach always chooses the cheapest machine instead of considering cost-efficiency for each task. In this case, many tasks actually cost more running on the cheapest machines than on their preferred machines. Therefore, the total cost is high. This result explains the importance of choosing suitable VM types for different workloads. The GAIN approach however always schedules the tasks on their preferred cost-efficient instances. In this way, it achieves cost optimization for each job and saves more cost than the Greedy approach, but it has lower instance utilization rate than the Greedy approach, which implies more partial instance hours are wasted. The SCS approach does not only take advantage of the task-level cost-efficiency but also better utilizes the partial instance hours through instance consolidation. Therefore it saves more cost than the GAIN approach and also has higher instance utilization rate.

Among the four workload patterns, the Growing case has the highest utilization, because as more and more jobs are submitted very quickly, all instances are filled with tasks and partial instance hours become fewer and fewer. Generally speaking, when the workload volume is high enough, task level cost-efficiency could dominate the overall cost. For this reason, the SCS and GAIN approaches can always beat the Greedy approach. When the deadline is longer, such cost-saving benefits become clearer, because the Greedy approach will place most tasks on the cheapest machines and therefore incur more cost than shorter deadlines.

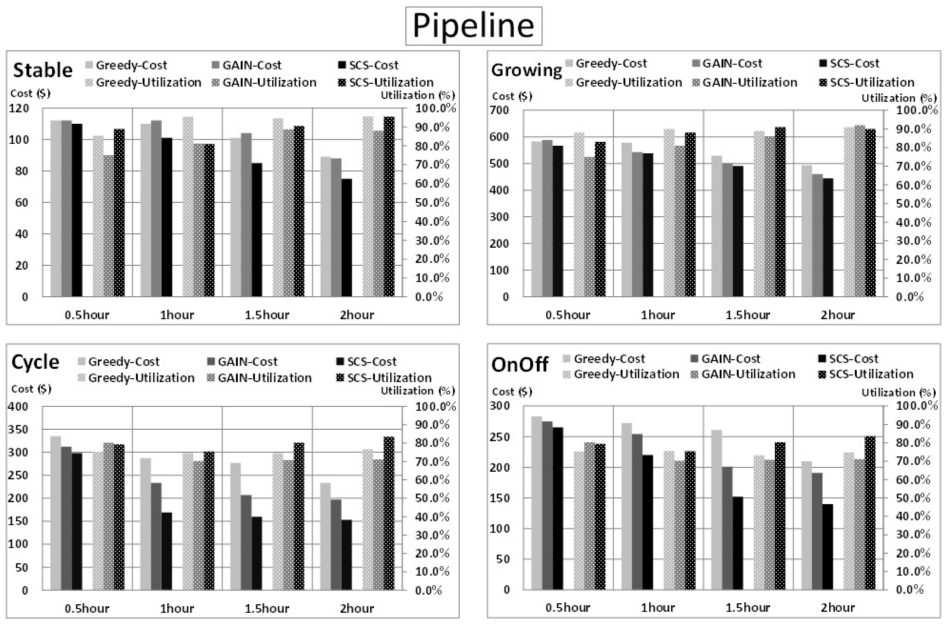

Figure 4.9: The performance for pipeline applications

4.3.3 Heavy Workload vs Light Workload

In the extreme cases, the workload volume can be very low, such as a single workflow instance. In such cases, the idea of placing tasks on their preferred cost-efficient instances may not always save money, because instances are not fully utilized and there could be a large portion of unutilized instances hours. In other words, the benefit of instance consolidation overweighs the cost-efficiency of individual tasks. However, when the workload volume is high enough to fill all the instances, placing tasks cost-efficiently also implies global cost-efficiency. This is because all the jobs can be processed with the minimum cost and there are very few idle instances. To illustrate this point, this dissertation shows the evaluation results of the pipeline application

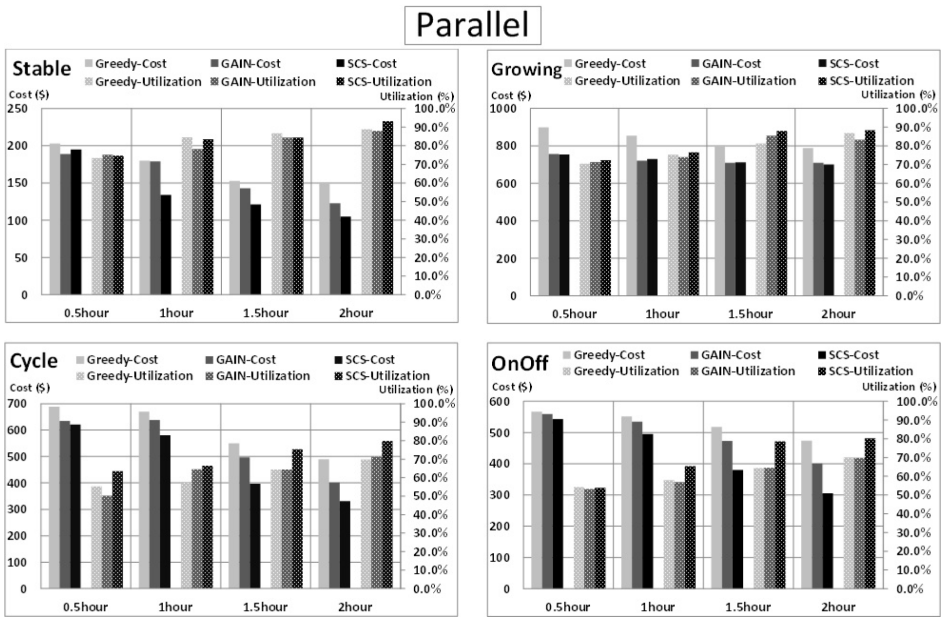

Figure 4.10: The performance for parallel applications

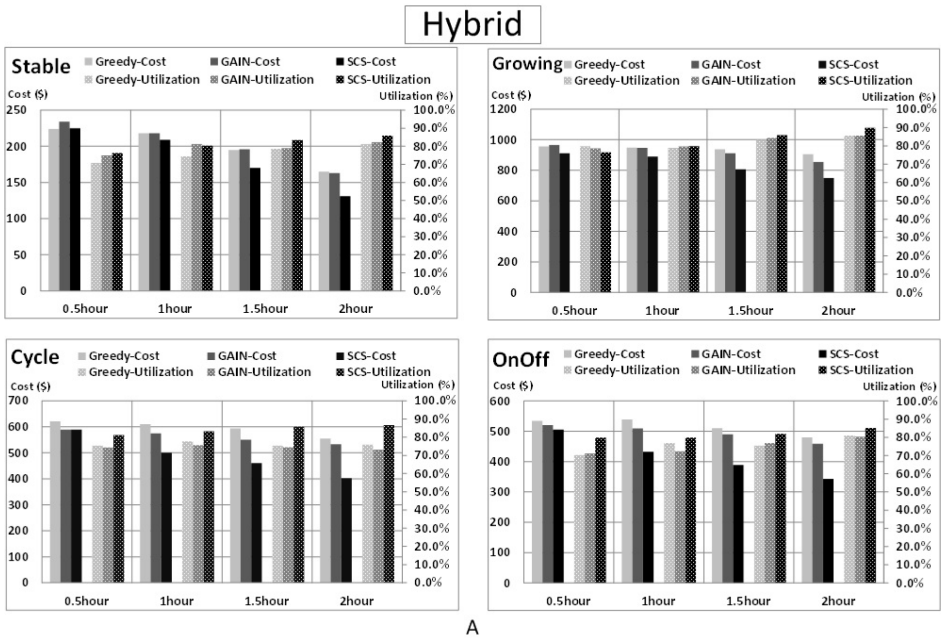

Figure 4.11: The performance for hybrid applications

with both low (X) and high (10X) workload volume using 1 hour deadline. From figure 4.12, it is shown that the Greedy approach works better than GAIN when the workload volume is low and performs worse when the workload volume is high. This is because it schedules as many tasks on the cheapest machines as possible, which is actually an instance consolidation strategy. The SCS approach works better than the Greedy and GAIN approaches in both low volume and high volume workload environment. It handles the low volume cases through instance consolidation and takes the advantage of task level cost-efficiency when the workload volume becomes high. Since parallel and hybrid applications show the similar performance, their results are not discussed here.

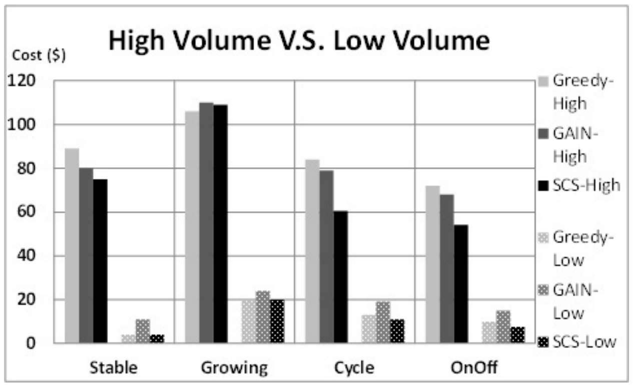

Figure 4.12: Heavy workload and light workload

4.3.4 Sensitivity to Inaccurate Parameters

This dissertation assumes that the task running time and the instance acquisition lag can be estimated. Under such assumptions, it is shown that all deadlines can be met through the dynamic scaling and EDF scheduling approach. However, accurate estimation of task execution time are not always available in practice and the lag of instance acquisition is actually not under a user's control. This section evaluates the auto-scaling mechanism's capability to handle inaccurate input parameters. It first allows the real task running time to be the estimated running time with ±20% error and tests the deadline non-miss rate for the pipeline application with 0.5 hour deadline (figure 4.13). The dynamic scaling nature of SCS handles the inaccurate task runtime estimation pretty well. It can complete more than 90% jobs within their deadlines, which is much better than the other two approaches. In fact, as the users allow longer deadlines, SCS can have an even higher deadline non-miss rate. Next, this dissertation allows the estimated instance acquisition lag with ±20% error and tests the deadline non-miss rate for the pipeline application with 1 hour deadline (figure 4.14). The instance acquisition lag can affect the mechanism's performance more than the task running time estimation (deadline non-miss rate is below 80%), because the auto-scaling mechanism reacts to the dynamic changes through instance acquisition, which is the core function of the auto-scaling mechanism. In the extreme cases, in which the acquired instance will never be ready to process user tasks, all jobs will not be finished and miss the deadline. Another factor to determine whether the instance acquisition lag is acceptable or not is the user's performance requirement. If the user has a very urgent deadline and needs to acquire new instances immediately, 1 minute and 10 minute lag do not make a significant difference when the criteria is no deadline misses. However, if the deadline is 2 hours for a short job, even if the instance acquisition lag is 30 minutes, the job can still be finished in time.

For all the workload patterns, the Growing case is the worst performance case. This is because it acquires more instances than all the other test cases, and therefore is affected more by the inaccurate lag estimation. This test case shows that the instance acquisition lag plays a very important role in the performance of an auto-scaling mechanism. Workload prediction techniques are needed to prepare early for instance acquisitions. Reducing the frequent operations (e.g. parallelism reduction) of instance acquisition and release is also beneficial.

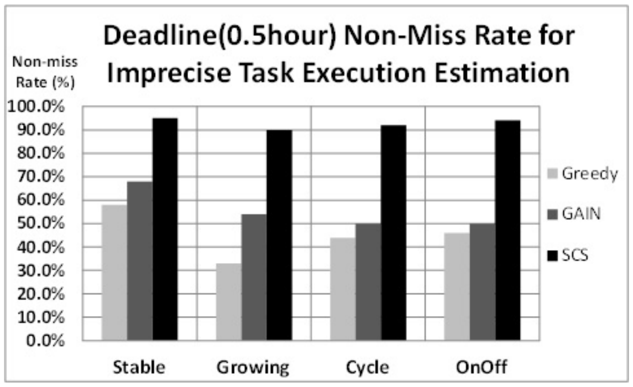

Figure 4.13: Inaccurate task execution time

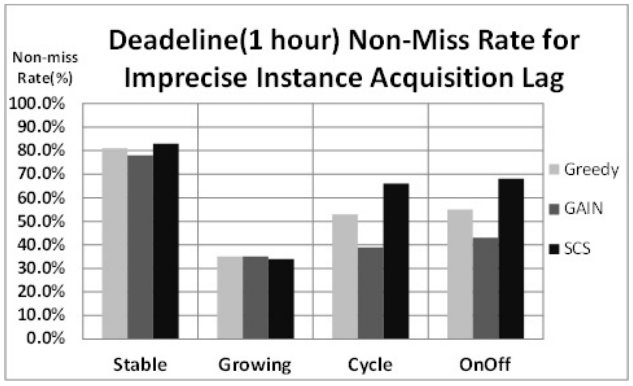

Figure 4.14: Inaccurate instance acquisition lag

4.3.5 Mechanism Overhead

Finally, this dissertation evaluates the mechanism overhead. In this test, it ignores the overhead of runtime information monitoring and update, such as the number of newly submitted jobs and the progress of running tasks. In practice, such information monitoring functionality may be provided by the cloud providers, such as Windows Azure Diagnostics [89] and AWS CloudWatch [90] or implemented by the application itself.

The overhead therefore largely depends on the way it is implemented. Here this dissertation only focuses on the performance of the core scheduling and scaling algorithm instead of the complete monitor-control loop. The test is run on a desktop with Intel P4 2.4G CPU, 4G memory and 512G storage. It measures the time of updating load vectors, consolidating partial instance hours and making scaling/scheduling decisions for different number of jobs (from 10 to 100000). The test uses 100 hybrid job classes with 16 different VM types. In figure 4.15, it is shown that the overhead is low and the performance scales linearly based on the job number (note, the X-axis is exponential). The low mechanism overhead is achieved through the following two techniques. The most time-consuming part, deadline assignment, can be done in preprocessing and the result can be cached after the first calculation. The later jobs of the same class do not really need to recalculate the deadline assignments each time. Secondly, the other time-consuming part, updating the Load Vector, can be implemented using pair-wise data structures instead a big array for each task, which saves both memory and computation time.

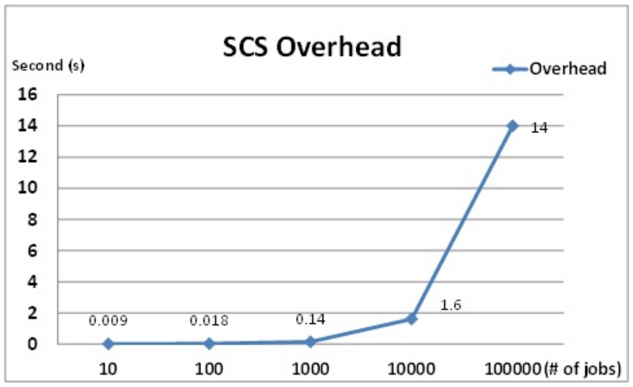

Figure 4.15: SCS overhead

4.4 Conclusion

This chapter presents an auto-scaling solution for SOA application models. It is different from the batch-queue model in the previous chapter. The workload are workflows instead of bags-of-tasks. It helps the service provider to finish the submitted jobs before user specified deadlines in a cost-efficient way. The solution is based on a monitor-control loop that can adapt to dynamic changes, such as workload bursting and delayed instance acquisitions. Evaluation results show that it can help to reduce costs for various applications models and workload patterns. The cost-saving ranges from 9.8% to 40.4% compared to the two baseline approaches. The instance consolidation process does not only improve the instance utilization rate but also reduces partial instance hour waste. It efficiently handles both the high and low workload volume, and successfully considers job-level and global-level cost-efficiency together. Moreover, The mechanism shows good tolerance to the inaccurate parameters. The monitor-control loop can help handle inaccurate task execution time and instance acquisition lag estimations. It makes fast responses to dynamic changes. The mechanism overhead is also largely reduced through preprocessing and caching.

This part of the work has been published in the 2011 international conference for high performance computing, network, storage and analysis (SC 2011) [29]. In the next chapter, this dissertation considers the other dimension of the optimization problem - minimize the job turnaround time within the budget constraints.

Chapter 5

Auto-Scaling to Minimize Job Turnaround Time within Budget Constraints in Cloud Workflows

The previous chapter solves the auto-scaling problem for deadline-constrained SOA based applications on the IaaS cloud. The idea is to assign deadlines to tasks and calculate the VM numbers using load vectors. The tasks are then scheduled based on their sub deadlines. The solution can help service providers with unlimited budget to minimize application cost (hence maximize profit) and meet job deadlines. Job turnaround time and cost are two conflicting goals for a cloud application. This chapter tackles the other dimension of the optimization problem - maximizing the application performance within the budget constraints. It aims to help the budget limited/sensitive service providers to make resource provisioning and allocation decisions.

5.1 Problem Definition

This dissertation uses another motivating example to illustrate the cost-constrained case. The department of environmental science plans to conduct the watershed modeling research in the cloud. The domain scientists first collect and process a large amount of data from multiple field observation sites. Then they perform model calibrations to determine the appropriate equation parameters to describe the watershed conditions. They also perform Monte Carlo simulations to predict future watershed movements. As a budget sensitive research organization, they decide to move the computing infrastructure into the cloud and process the research workload using cloud resources. The IT team therefore builds the computing service in the cloud and runs the application with a budget constraint (e.g. \$10/hour). This application will process workflow jobs submitted by the domain scientists and researchers. The IT team is the service provider and the domain scientists are the service customers. One important goal for the IT team is to get the fastest performance for all the jobs within the limited budget. In other words, they need an auto-scaling mechanism to minimize the job turnaround time and also make sure the running cost does not exceed the budget constraint.

This cost-constrained auto-scaling problem shares some assumptions with the previous deadline-constrained problem, such as the IaaS cloud resource definition, workload definition, etc. To highlight the differences of the two problems and avoid duplicating the same contents, this chapter only presents the assumption changes in the following text. The key difference is that the goal of the auto-scaling mechanism becomes minimizing job turnaround time instead of minimizing application cost.

Definition 1 (change) - Job Class. A job class J_j includes two properties (equation 5.1). The first one is a direct acyclic graph ($DAG_j(S_i)$) which describes its workflow. The second one is the priority (p_j) which indicates the job classs importance. A larger number p_j implies a higher priority. A job with high priority should be finished earlier compared to the job with low priority (assuming the two jobs have the same workflow and are submitted at the same time). Compared to the previous subproblem, the job classes are now associated with priorities instead of deadlines. Service customers can submit several instances of the same job class. For example, two students can submit the same Monte Carlo simulation jobs. In this case, there are two jobs job_1 and job_2 submitted in the cloud application, and they belong to the same job class J_{mc}. For each job, this dissertation assumes that the job submission time (job_{submit}) and the finish time (job_{finish}) can be recorded. The job turnaround time is defined in equation 5.2.

$$J_j = \{DAG_j(S_i), p_j | S_i \in app\} \tag{5.1}$$

$$job_{turnaround} = job_{finish} - job_{submit} \tag{5.2}$$

Definition 2 (change) - Constraint. The cloud application has a budget constraint in terms of dollars/hour (B). In other words, the cost of all the running instances at any time t belonging to the cloud application cannot exceed the budget constraint B. Compared to the previous chapter, the constraint is application level budget instead of job level deadlines.

$$\sum_i cost(vm_i)_t \leq B \tag{5.3}$$

Definition 3 (change) - Goal. Because the target of this dissertation is an application in which the workload is a stream of workflow jobs and is not known in advance, the weighted average job turnaround time (equation 5.4) is defined as the performance metric to represent the overall application execution speed (performance) in the observation period. A shorter average job turnaround time means faster execution (better performance) of the application. The goal of the auto-scaling mechanism is therefore to minimize the weighted average of the job turnaround time.

$$Min(weighted(job_{turnaround})) = Min(\sum_{job}(p_j \times job_{turnaround}/\sum_{job}(p_j))) \tag{5.4}$$

5.2 Solution

This section presents two solutions for the auto-scaling problem with budget constraints in the cloud. These two solutions make scaling and scheduling decisions in different orders. They are named as scheduling-first algorithm and scaling-first algorithm separately.

5.2.1 Scheduling-First Algorithm

The scheduling-first algorithm makes resource allocation decisions first and then makes resource provisioning decisions. The idea of scheduling-first algorithm is to first allocate the service provider budget to each individual job based on the job priority and then schedule as many tasks as possible to their fastest execution VMs within the job budget constraint. When the VM type for each task is determined, the auto-scaling mechanism will acquire the VM instances based on the scheduling plan.

Step 1 - Distribute budget. The application budget is allocated to individual jobs based on priority. High priority jobs have bigger budgets and therefore more tasks are scheduled on the faster machines. While low priority jobs have smaller budgets and more tasks run on the slower machines. A simple budget distribution scheme is to allocate the service provider budget to individual jobs proportionally based on their priorities. For example, if there are three jobs with priority 1, 2, 3 and the service provider budget is \$6/hours, then job_1 gets \$1/hour, job_2 gets \$2/hour and job_3 gets \$3/hour. This means that the cost of the running machines allocated for job_1 cannot exceed \$1/hour. Note, in this dissertation, the budget is in the form \$/hour instead of \$/job. The auto-scaling mechanism targets the cloud application as a whole instead of every single job. Therefore, the individual budget B_j for job_j at any time t is defined as follows (equation 5.5).

$$B_j = B \times \frac{p_j}{\sum_j p_j} \tag{5.5}$$

Step 2 - Schedule tasks. For each job, the auto-scaling mechanism further allocates the budget B_j to the tasks that are running ($task_r$) and the tasks that are ready and waiting to run ($task_w$). The machine types for the running tasks are known and their cost can be calculated easily, which is just the sum of the cost of all the running machines for that job. The cost is denoted as $cost(task_r)$. The idea is to allocate the remaining budget ($B_j - cost(task_r)$) to the waiting tasks. If the remaining budget is less than or equal to 0, or there are no waiting tasks, the task scheduling process is stopped. Otherwise, the auto-scaling mechanism allocates the remaining budget to the waiting tasks and schedules them on their fastest machines until there are no waiting tasks or the budget is insufficient.

Step 3 - Consolidate budget. After step 2, there may be remaining budget available for some individual jobs. Some budget is left because there are no waiting tasks. Some budget is left because the budget is not big enough to acquire a new VM. The remaining budget of each job is returned to the system. The consolidated budget can be further used to purchase new machines for waiting tasks that are ready but not scheduled in step 2. This is like another round of task scheduling, however this time, the algorithm does not distribute the budget to individual jobs, but uses the budget to directly purchase machines for high priority tasks among all the jobs. This implies that a job may actually be allocated with a budget bigger than calculated in step 1, and only high priority jobs have such advantages.

Step 4 - Acquire instances. After the budget is distributed to every job and the scheduled VM type for each task is determined, the auto-scaling mechanism will acquire the VMs based on the scheduling plan. If there are idle instances for the scheduled VM type, the tasks will run on the idle instances first. If there are no idle instances for the VM type, the auto-scaling mechanism acquires more instances and schedules the tasks on the newly acquired instances. Because the scheduling plan is determined within the budget constraint, the auto-scaling mechanism makes sure that the running cost will not exceed the budget constraint.

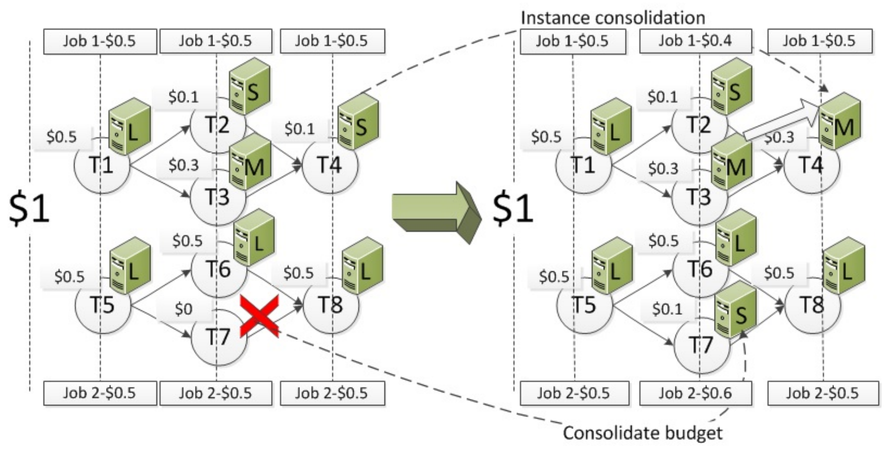

Figure 5.1: The scheduling-first algorithm

To better understand the idea of the scheduling-first algorithm, this dissertation uses the above example (figure 5.1) to illustrate the four steps. Without loss of generality, it assumes that there are two jobs with the same priority submitted to the cloud application. There are three VM types available, large (L) with price \$.5/hour, medium (M) with price \$.3/hour and small (S) with price \$.1/hour (Note tasks have different execution times on different VM types. Larger machines do not imply faster execution). The system budget is \$1/hour so each job gets \$.5/hour. Therefore, T1 and T5 are both scheduled on the large machines. When T1 and T5 are finished, the auto-scaling mechanism further allocates job budget to the remaining tasks. In the example, T2 is scheduled on a small machine and T3 is scheduled on a medium machine. At the same time, T6 is scheduled on a large machine and T7 does not have sufficient budget. After budget consolidation process, job_1 returns \$.1/hour to the system and the returned budget can be used to schedule T7 on a small machine (see the "consolidate budget" dash line). At last, T4 is scheduled on a small instance and T8 is scheduled on a large instance. Though T4 originally should run on a small machine but it is actually scheduled on a medium machine that finishes T3. This process is called instance consolidation that tries to save partial instance hours. It will be further detailed later.

Algorithm 4: Scheduling-first algorithm

```
allocate budget to individual jobs based on priority;
for each job do
    waitTasks ← GetReadyTasks(job) ;
    while $B_j > cost(VM_{cheapest})$ && $waitTask.size > 0$ do
        $tempTask \leftarrow waitTasks.pop$;
        schedule tempTask on $VM_{fastest\_now(tempTask)}$;
        $B_j \leftarrow B_j - cost(VM_{fastest\_now(tempTask)})$;
    end
    remainTasks.add(waitTasks);
    $B \leftarrow B + B_j$;
end
while $B > cost(VM_{cheapest})$ && $remainTasks.size > 0$ do
    tempTask = remainTasks.popByPriority();
    schedule tempTask on $VM_{fastest\_now(tempTask)}$;
    $B \leftarrow B - cost(VM_{fastest\_now(tempTask)})$;
end
```

5.2.2 Scaling-First Algorithm

The idea of the scaling-first algorithm is to first determine the type and the number of the cloud VMs within the budget constraint and then schedule the submitted jobs on the acquired resources based on job priority to minimize the weighted average job turnaround time. Different from the scheduling-first algorithm, the scaling-first algorithm first makes resource scaling decisions and then makes job scheduling decisions.

Step 1 - Determine the VM type and number. To determine the type and the number of the cloud instances, the scaling-first algorithm assumes all the tasks are scheduled on their fastest machines and calculates the cost C_{fast} of the cloud resources needed within the next hour. Because the system only has budget B, the ratio B/C_{fast} can be used to acquire the cloud resources proportionally for each VM type. For example in figure 5.2, there are three jobs submitted into the cloud application. all the tasks are assumed to be scheduled on their fastest machines within the next hour. T1 will be scheduled on a large machine running for 0.5 hour; T2 will be scheduled on a large machine as well but for 1 hour. However within the next hour, it will run on a large machine for only 0.5 hour because it starts after T1 finishes. Similarly, T4 will consume 0 hour on a small machine because it cannot start within the next hour. The auto-scaling mechanism applies the same instance consumption calculation for all the three jobs. When adding together the instance consumption of all the three jobs, the total cost C_{fast} is \$2.2/hour (3 large instances, 2 medium instances and 1 small instance. Note instance number needs to round up). Assuming the system budget is \$1.4/hour, proportionally (B/C_{fast}), the scaling-first algorithm can acquire 2 large instances, 1 medium instance and 1 small instance (Instance number needs to round down considering the budget constraint).

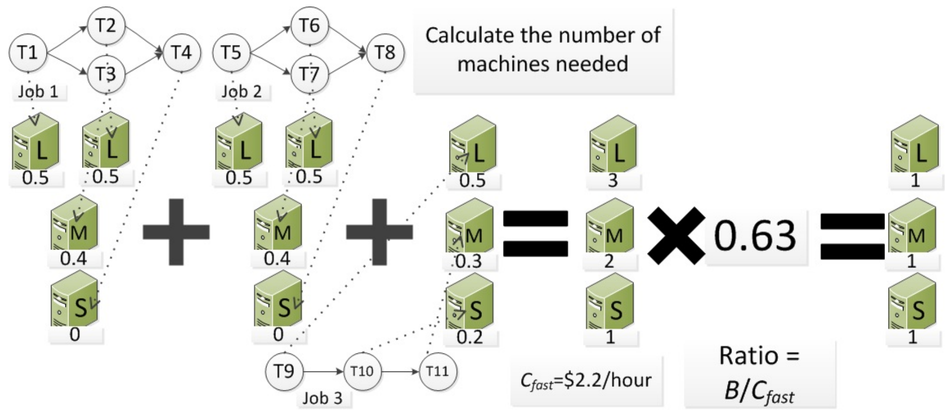

Figure 5.2: Determine the VM type and number

Step 2 - Consolidate budget. Instances are acquired proportionally to the ratio of budget over fastest cost (B/C_{fast}). However, there could be some remaining budget available since the number of instances can only be an integer (rounding down effects). In some cases, the remaining budget can be large enough to purchase more machines. As shown in figure 5.3, based on the ratio B/C_{fast}, the auto-scaling mechanism acquires one instance for each VM type and the actual cost is \$0.9/hour (0.5+0.3+0.1). Therefore, the remaining budget \$0.5/hour can be used to purchase one more large machine. In other words, the auto-scaling mechanism finally acquires 2 large machines, 1 medium machine, and 1 small machine.

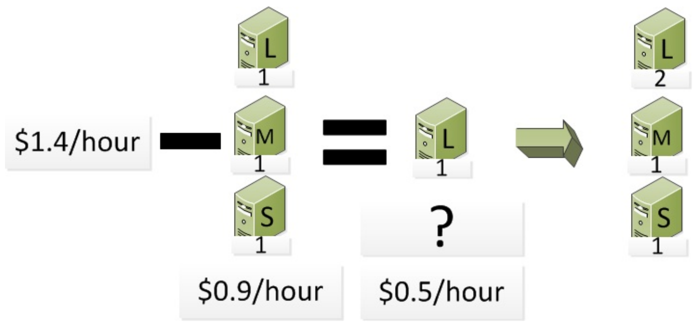

Figure 5.3: Consolidate budget

Step 3 - Schedule jobs. When the resources have been acquired, jobs are scheduled on their fastest machines based on priority. For each VM type, there is a priority queue for the waiting tasks. Tasks that are ready to run will enter into the priority queue of its preferred (fastest) VM type and compete for the next available instance based on its priority (inherited from their jobs). For example in figure 5.4, only 2/3 machines can be acquired because of the budget constraint. All the tasks enter into the priority queue for each VM type. Because job_2 has the highest priority, all its tasks are in the front. While job_3 has the lowest priority, all its tasks are in the back. Job_1's tasks are in the middle.

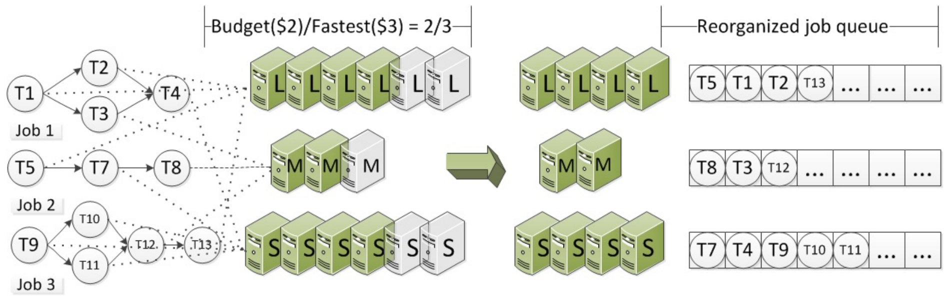

Figure 5.4: The scaling-first algorithm

The scaling-first algorithm assumes that all the tasks will run on their fastest machines and calculates the cost to acquire all these fastest machines at the same time. Because of the budget constraint, the auto-scaling mechanism can only acquire a fraction based on the ratio of B/C_{fast}. When the acquired instances are ready, the individual tasks will compete for their fastest VMs based on the job priority.

Algorithm 5: Scaling-first algorithm

```
$fastConsumption \leftarrow [0, ..., 0]$;
for each job do
    calculate the instance consumption ($jobConsumption$) for the next hour;
    $fastConsumption \leftarrow fastConsumption + jobConsumption$;
end
$cost_{fast} \leftarrow calculateCost(fastConsumption)$ ;
$capableConsumption \leftarrow fastConsumption \times (B/cost_{fast})$ ;
while true do
    if $runningCost + cost(VM_{cheaptest}) > B$ then
        goto scheduling;
    end
    for each VM type $VM_v$ do
        if $size(VM_v) < capableConsumption[VM_v]$ && $runningCost + cost(VM_v) <= B$ then
            acquire one instance of $VM_v$;
            $B \leftarrow B - cost(VM_v)$;
        end
    end
end
label: scheduling;
for each VM type $VM_v$ do
    schedule the tasks on idle instances based on priority;
end
```

5.2.3 Instance Consolidation

In the cloud, VM instances are billed by hours. Partial instance hours are charged as full hours. The idea of mapping the tasks to their fastest machines is to minimize the job execution time as much as possible. However, this strategy does not consider the unutilized partial instance hours. In such cases, one effective strategy is to schedule tasks to the unutilized VMs to improve the resource utilization and reduce the job turnaround time. The instance consolidation processes for both scheduling-first and scaling-first algorithms are similar. There are basically two cases. If some instance is idle and some ready tasks are waiting for their scheduled VM types, the auto-scaling mechanism can try to place the ready task on this idle machine. If the idle instance is faster than the originally scheduled machine type and there is no task that will be ready before this task is finished, the auto-scaling mechanism can go ahead and schedule the task on this faster idling machine. If the idle instance is slower than the originally scheduled VM type or there could be tasks (that is scheduled on the idle instance) becoming ready before this task is finished, two conditions need to be checked, scheduling the task on this idling machine can finish this task earlier, in other words, the longer execution time saves more time than waiting for the originally scheduled faster VM type. Second, scheduling the waiting task on the slower machine will not affect the execution of future tasks. The following figure illustrates these two cases.

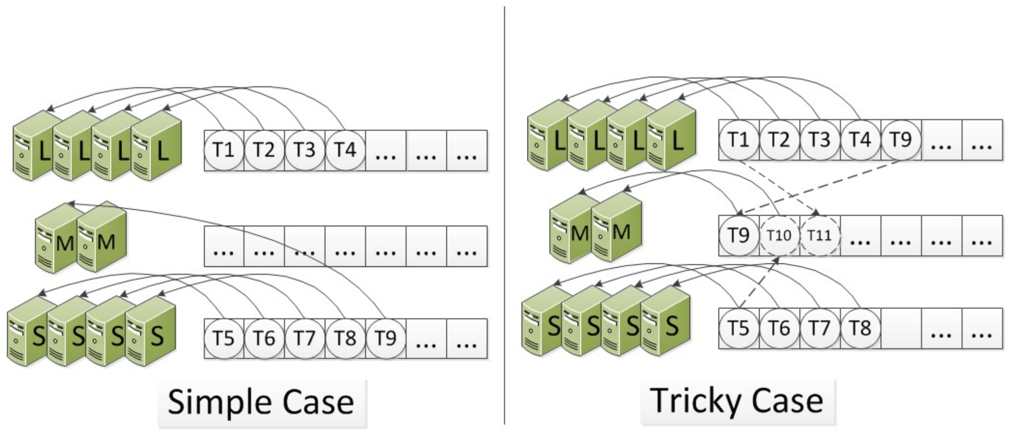

Figure 5.5: Instance consolidation

As shown in the above figure 5.5, in the simple case, T9 is originally waiting for a small instance. However there are two medium idle instances available. In such case, the auto-scaling mechanism will schedule T9 on one medium machine to speed up the job execution (assuming T9 runs faster on the medium machine) and improve instance utilization. In the tricky case, T9 is originally scheduled on a large machine. However, before the auto-scaling mechanism moves its execution on an idle medium machine, it needs to check two conditions. The first one is that moving T9 on a medium machine will finish the task earlier than letting it wait for a large machine. The second condition is that, during the execution of T9, it will not delay the tasks which are scheduled on the medium instances. In the example, there is only one task T10 that will be ready when T9 is executing on one medium machine. The task T10 (that is ready after T5 is finished) will be scheduled on the other medium machine. However, because the task T11 can only start after T1 is finished and T9 will finish its execution at that time, there will be no conflicts in this case. Note, only jobs that have been submitted are considered in the instance consolidation.

Algorithm 6: Instance consolidation

```
for each task in waitTasks do
    for $VM_v$ that $exe(task, VM_v) < exe(task, VM_{schedule})$ do
        if no conflicts on incoming tasks then
            schedule the task on one idle instance of $VM_v$;
            goto check;
        end
    end
    for $VM_v$ that $exe(task, VM_v) > exe(task, VM_{schedule})$ do
        if $exe(task, VM_v) < waitingTime + exe(task, VM_{schedule})$ && no conflicts on incoming tasks
        then
            schedule the task on one idle instance $VM_v$;
            goto check;
        end
    end
    label: check;
    if there is no idle machines then
        break;
    end
end
```

5.2.4 VM Startup and Shutdown

In the cloud, service providers can request VM instances at any time. However, it may take several minutes or even longer for an acquired instance to be ready to use. The VM startup time varies by cloud providers, image size, data center location, instance type, the number of requested instances at the same time, and other factors [39][76]. In the scheduling-first algorithm, the real execution time should be refined as $lag_v + exe(task, VM_v)$ if the task needs to be scheduled on a newly acquired machine. However, for the scaling-first algorithm, the tasks are placed in the priority queue and the waiting time is less predictable, therefore, the execution time is not refined in this algorithm. In the evaluation section, inaccurate VM startup time estimation will be discussed.

The instance shutdown decisions are simpler to make than acquiring an instance. Because instances are billed by full hours, in both solutions, the auto-scaling mechanism will check all the idle instances that are approaching full hour operation. If there are no tasks scheduled on them (scheduling-first) or the number of instances is more than needed (scaling-first), the VM can be shutdown.

5.2.5 Budget Allocation Schemes

This dissertation assumes the budget constraint is in the form of dollars per hour instead of dollars per job. This is because the budget is assigned to a continuously running application, in which the start time and end time cannot be clearly identified. This is a big difference from the per-job level budget constraint (in the form of dollars per job). It is a more reasonable assumption from the service provider perspective because they are targeting the overall application performance, instead of a single job or a set of jobs that is known in advance [21][54][55]. Further, this dollars per hour form is also consistent with the current prevailing hourly pricing scheme in the cloud [6][11][13]. This dissertation proposes three budget allocation schemes to convert a budget for a long time horizon to hourly based budget. (1) The running application can distribute the total budget evenly in its service life cycle. If a yearly budget x is approved by the finance department, the daily budget and hourly budget will be $x/365$ and $x/8760$. (2) If the application experiences seasonal behaviors, budget can be allocated based on the workload. For example, holiday season and weekends normally experience less workload compared to regular business hours therefore a smaller hourly budget can be allocated. (3) If workload prediction technique can be used, more fine-grained and accurate budget allocation strategies can be developed. This essentially converts the budget form from \$/hour to \$/job.

5.2.6 The Overall Algorithm

As a component for a continuously running application, the auto-scaling mechanism works like a monitor-control loop. It runs periodically to collect updated workload and VM information, and makes dynamic scaling and scheduling decisions in response to the changes. The pseudo code of the overall mechanism can be found in algorithm 7.

Algorithm 7: Auto-scaling

```
while true do
    collect the runtime information (including VM & workload);
    run scheduling-first or scaling-first algorithm;
    run instance consolidation;
    shutdown idle instances if they approach full-hour operation;
    wait for the next monitoring interval;
end
```

Essentially, the scheduling-first algorithm makes local job scheduling decisions first by distributing the system budget to each job while the scaling-first algorithm makes resource acquisition decisions first by looking at the overall workload type within the next hour. The job priority affects the amount of budget allocated to each job in the scheduling-first algorithm and therefore, high priority jobs have large budgets and run faster. While the job priority affects the task waiting time for their preferred resources in the scaling-first algorithm, therefore high priority jobs have shorter waiting time and run faster. Since jobs are associated with priorities which can hugely affect the budget and resources allocation decisions, job starvation is always a problem. Job starvation prevention mechanisms, such as aging, can be applied to ensure old and low priority jobs to be handled in time. However, this is not the focus of this dissertation, so it is not further discussed.

5.3 Evaluation

To make it consistent and comparable, this chapter uses the same representative application workflows (figure 4.7) and workload patterns (figure 4.8) to evaluate both scheduling-first and scaling-first algorithms. The five VM types used in the evaluation can also be found in table 4.1 in the previous chapter. This chapter will not duplicate the testing environment descriptions. It simulates a 72-hour experiment period with different combinations of workload patterns and application workflows. The task execution time on different VM types is randomly generated (the execution time of every single task ranges from 3 minutes to 4 hours on different VMs [87][88]). Every combination is tested for 200 times with randomly generated execution times. The average performance of the auto-scaling mechanisms are reported in the following sections.

5.3.1 Job Turnaround Time

This section records the weighted average job turnaround time of the two auto-scaling mechanisms and compare them with the strategy of choosing a fixed machine type (standard machine). Their performances are shown in figure 5.6, figure 5.7 and figure 5.8.

For all the workflow applications and workload patterns, both scheduling-first and scaling-first algorithms have shorter average job turnaround time than choosing a fixed VM type. They can reduce the job turnaround time from 9.6% to 45.2%, depending on the amount of available budget. When the budget is small (e.g. $5/hour), there is not too much budget available to choose faster VM types, and therefore the performance improvement of the scheduling-first and scaling-first algorithms is small. When the budget becomes larger, the advantages of the two algorithms are clearer. This is because a larger budget allows more tasks to run on faster machines and the overall job turnaround time can be reduced more compared to small budget. This result confirms the importance of choosing appropriate instance types based on the workload and scheduling tasks on their preferred machines.

In current experiment settings, it is shown that when the budget is small, the scaling-first algorithm works better (shorter job turnaround time) than the scheduling-first algorithm. When the budget is large, the scheduling-first algorithm works better. The reason behind this result is the trade-off between the waiting time for the resources and the task performance degradation on slower machines. The scheduling-first algorithm starts as many tasks as possible, which essentially reduces the task waiting time. In contrast, the scaling-first algorithm always tries to schedule tasks on their fastest machines, which reduces the task execution time. The results show that budget \$15/hour and \$20/hour are close to the threshold for current experiment settings. When the budget is below this threshold, the waiting time for faster machines is shorter than the performance degradation on the slower machines, so the scaling-first algorithm wins. When the budget is above the threshold, the performance degradation on slower machines becomes smaller than the waiting time for faster machines, so the scheduling-first algorithm wins. Moreover, in the growing workload pattern, the scaling-first algorithm always beats the scheduling-first algorithm, this is because the budget is always below the threshold considering the continuously increasing workload. Therefore, the relationship between the budget and VM prices largely determines the performance of the two algorithms, because the relationship essentially determines the winner of task waiting time and task execution time.

The performance differences of the two algorithms are also related to the arrival patterns of the workload. If there are many jobs arriving at the cloud application around the same time, the scaling-first algorithm works better than the scheduling-first algorithm. The reason is that the scaling-first algorithm has a global view and can acquire instances based on all the tasks running within the next following hour. However, the scheduling-first algorithm will acquire instances based on the ready tasks only. The scheduled VM types may not be cost-efficient picks for the following tasks in the workflow. On the other hand, when the jobs do not arrive in a batch pattern, the scheduling-first algorithm works better than the scaling-first algorithm. This is because the scheduling-first algorithm makes job wise decisions so it does not affect the global scaling decisions that much. However, in such cases, the scaling-first algorithm may make aggressive decisions based on a few jobs submitted in the system. This could result in bad choices of VM types for the future workload.

5.3.2 Instance Consolidation

The cloud instances are billed by hours. Partial instance hours are charged as full hours even that the instances are idling. The idea of instance consolidation is to take advantage of such unutilized instance hours by scheduling tasks on them. This section shows the reduction of job turnaround time and the improvement of

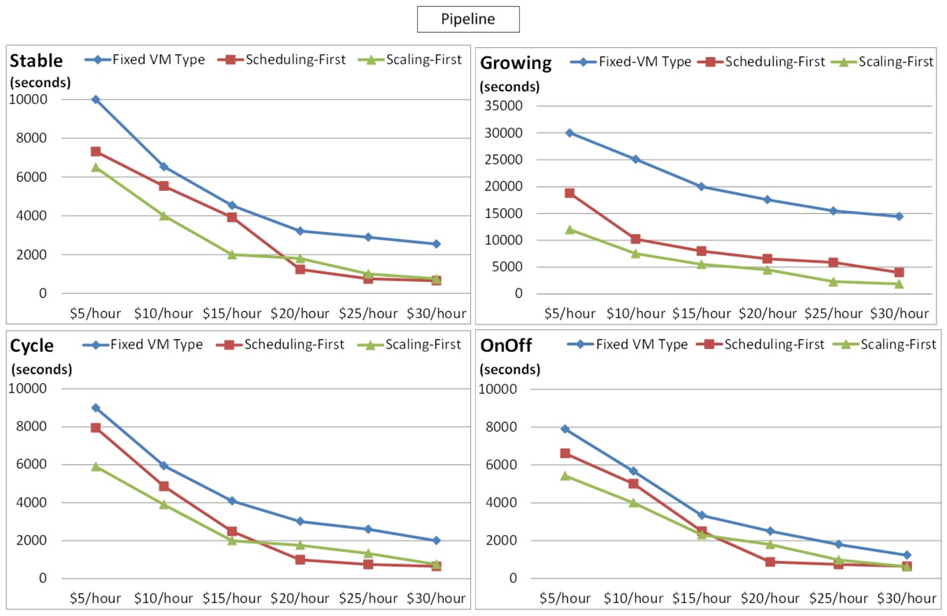

Figure 5.6: The job turnaround time for pipeline applications

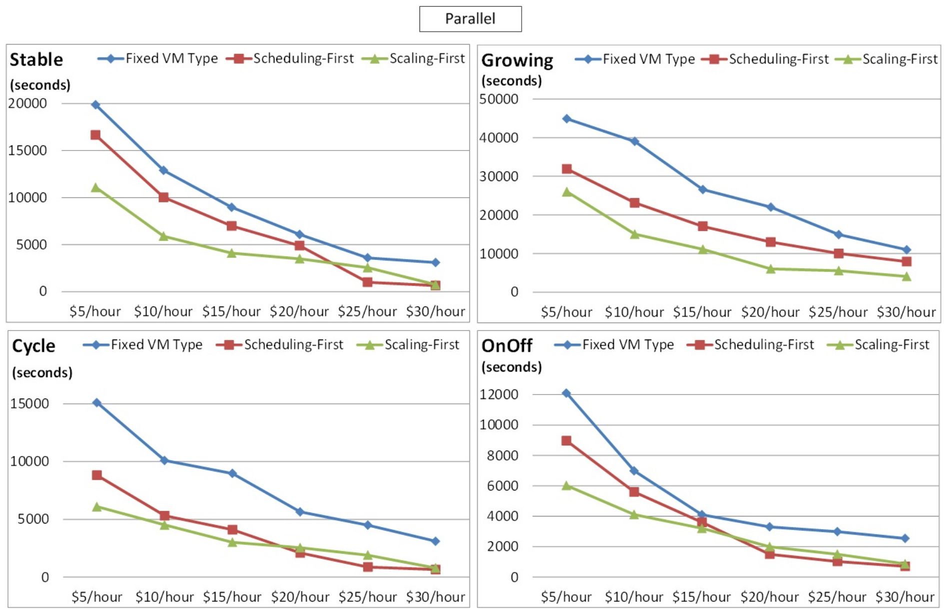

Figure 5.7: The job turnaround time for parallel applications

resource utilization contributed by the instance consolidation process. Since the four workload patterns show similar performance, here it only shows the figures for the hybrid application with the cycle workload pattern. From figure 5.9 and figure 5.10, it is shown that instance consolidation helps to improve resource utilization and more importantly reduce the job turnaround time. For both scheduling-first and scaling-first algorithms, when the budget is small, the improvement contributed by instance consolidation is low, this is because the workload is large enough to feed the acquired resources and there is not too much space for instance

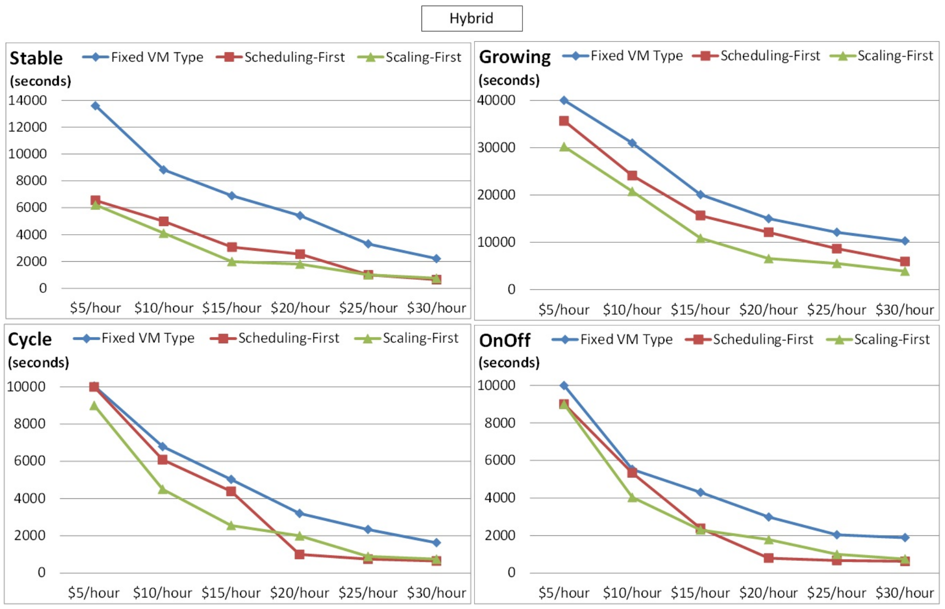

Figure 5.8: The job turnaround time for hybrid applications

consolidation. When the budget is large, the improvement contributed by instance consolidation is also low, this is because most of the tasks are processed on their fastest machines and there is no need to consolidate instances. When the budget is in between, the benefit of instance consolidation is clearer. For example, in the budget range between \$15/hour and \$25/hour, utilization rate improvement ranges from 2.2% to 19.9% while the job turnaround time improvement ranges from 9.0% to 35.1%. Generally speaking, compared to the scaling-first algorithm, the scheduling-first algorithm benefits more from the instance consolidation process. This is because the scheduling-first algorithm is a job wise resource allocation scheme and takes less global runtime information into the decision making process. Instance consolidation therefore is a strong complementary strategy to fill those unutilized instance hours.

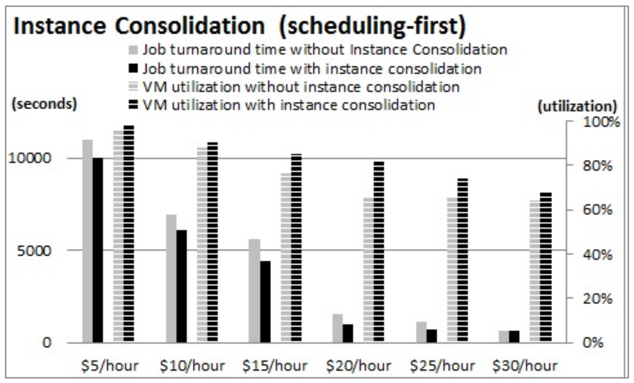

Figure 5.9: Instance consolidation for scheduling-first algorithm (hybrid + cycle)

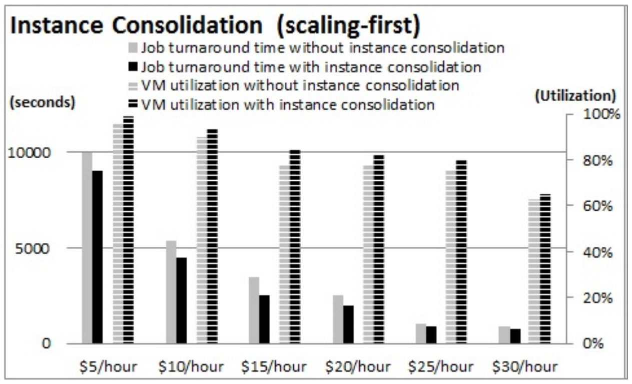

Figure 5.10: Instance consolidation for scaling-first algorithm (hybrid + cycle)

5.3.3 Sensitivity to Inaccurate Parameters

Similar to the previous chapter, the task execution time and the VM startup delay are assumed known to the auto-scaling mechanism. However, such information may not be always available or accurate in practice. This section tests the auto-scaling mechanism's sensitivity to these parameters. For both algorithms, it first allows the estimated task execution time to be centered around the real task execution time with ±20% error and then it allows the estimated VM startup delay to be centered around the real VM startup delay with ±20% error. The testing results are shown in the following two figures. In both cases, it dose not show significant performance degradation. The performance difference ranges from -10.2% to 16.7% in the current experiment settings. This implies that the proposed auto-scaling mechanisms can handle inaccurate parameters well. Moreover, this test keeps increasing the estimation error of the task execution time to see if it can find some threshold which could significantly degrade the mechanism's performance. It finally finds that when the estimation error reaches around ±60%, the results would become significantly worse. The job turnaround time will increase more than 48.3%. The main reason is that the algorithms cannot correctly rank the fastest machines among different VM types and the tasks treat slower machines as their preferred machine type. Therefore, as long as the auto-scaling mechanism can correctly rank the machine type by speed, inaccurate task execution time will not significantly hurt the performance. In practice, the cloud VM startup time is much more stable compared to the task execution time [39], so the stress tests on the VM startup delay are not performed.

5.3.4 Mechanism Overhead

This section evaluates the overhead of the two algorithms. Particularly, it reports two overhead components, the overhead of the core algorithm (before instance consolidation) and the overhead of the instance consolidation (ic). The test runs on a desktop with Intel 2.4G quad core CPU, 4G memory and 512G (7200rpm) hard

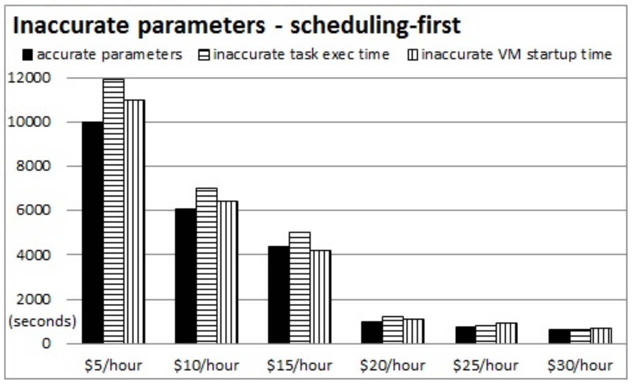

Figure 5.11: Sensitivity to inaccurate parameters (20%) for scheduling-first algorithm (hybrid + cycle)

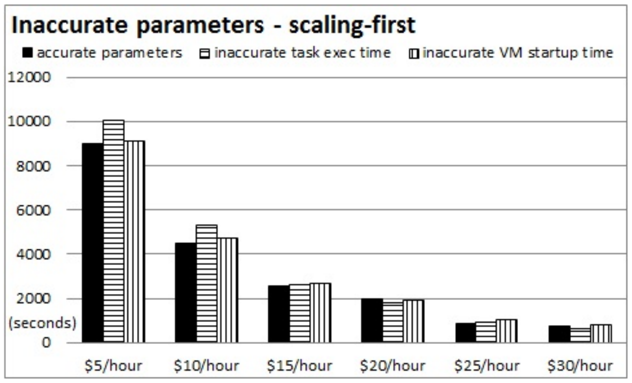

Figure 5.12: Sensitivity to inaccurate parameters (20%) for scaling-first algorithm (hybrid + cycle)

disk. The running time of the two algorithms with different number of jobs (100000 jobs and 500 job classes maximum) are recorded. From figure 5.13, it is shown that the scheduling-first algorithm has bigger overhead than the scaling-first algorithm and the majority of the overhead comes from the instance consolidation process. In current implementation, the task scheduling plan (in the scheduling-first algorithm) and the instance hour consumption (in the scaling-first algorithm) only need to be calculated once for each job class. The results can be stored in the cache for future look up. The same calculation for the incoming workflow jobs is therefore avoided and the mechanism overhead is greatly reduced. So the results show that the overhead of the core algorithms is not high and it scales linearly with the number of jobs. However, the instance consolidation process is largely dependent on the runtime information (i.e. submitted jobs and running machines) which needs to be updated and recalculated every time. Therefore, the instance consolidation process dominates the overall mechanism overhead. And the scheduling-first algorithm has higher overhead because the instance consolidation process plays a bigger role compared to the scaling-first algorithm.

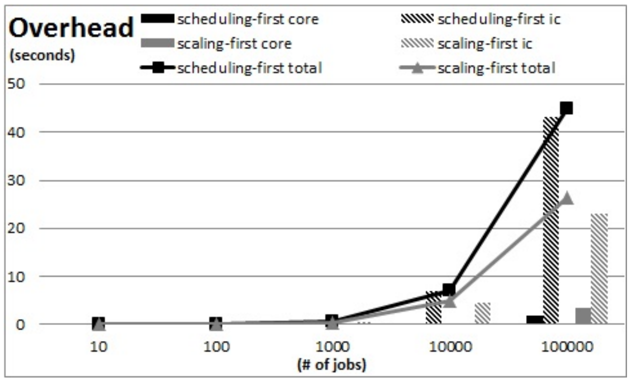

Figure 5.13: The overhead of scheduling-first and scaling-first

5.4 Conclusion

Compared to the previous chapter, this chapter solves the other dimension of the auto-scaling problem for cloud workflows - maximizing application performance (minimizing job turnaround time) within the budget constraints, which is helpful for the budget limited service providers. From the solution perspective, there are several improvements. It explicitly considers the scaling and scheduling as two separate processes. The scheduling-first and scaling-first algorithms make resource provisioning and allocation decisions in different orders. The scheduling-first algorithm allocates the application level budget to individual jobs and schedules as many tasks as possible. The scaling first algorithm acquires the cloud resources by looking at the overall workload and schedules tasks based on their priorities. This is a trade-off between waiting time and execution degradation. The scheduling-first algorithm shows better performance when the budget is high while the scaling-first algorithm shows better performance when the budget is low. Instance consolidation process is a good complementary strategy to the scheduling-first algorithm which creates more instance hour fragments than the scaling-first algorithm. It is shown that instance consolidation process can help to increase the resource utilization rate by 2.2% - 19.9% and reduce job turnaround time by 9.0% - 35.1%. However instance consolidation is also the major contributor to the mechanism overhead because the scheduling decisions need to be recalculated every time when the runtime information is updated. The auto-scaling solutions show good tolerance (-10.2% to 16.7%) to the inaccurate parameters (±20% error), as long as the the ranks (e.g. cost, execution time) among VMs can be correctly identified, the two algorithms can acquire correct VM types to process the workload.

This work has been submitted to a conference and it is under review [30]. The next chapter extends the application model to the data-intensive environment. It explicitly models the performance and cost for intermediate data storage and transfer.

Chapter 6

Auto-Scaling and Intermediate Data Management for Data-Intensive Applications

Data-intensive applications are playing a very important role in both the business world and scientific community. People buy and sell millions of items using the online trading websites, share news, photos and experiences through the social network services, and work together through the collaboration platforms [91]. Scientists in high-energy physics (e.g. CyberShake [92], ATLAS [93]), astronomy (e.g. Montage [24], SDSS [94]), and bio-informatics (e.g. BIRN [95], BLAST [96]) need to process and manage large volume of data to perform simulations and conduct scientific research. In both areas, a large amount of data may be captured, cleaned, transformed, transported, stored, accessed and visualized to extract useful knowledge and important findings. While high processing power machines are still necessary to speed up the application execution, the involvement of large data sets means that efficient data access from disks and data movement across servers is becoming an essential part of computation.

One great challenge for the applications running on the IaaS cloud is to determine the size of the resource pool and allocate the acquired resources in a cost-efficient way. Chapter 3, 4 and 5 have presented several auto-scaling solutions for both batch-queue based applications and SOA based applications. However, the problem becomes even more complex when a large volume of intermediate data is generated during the execution, such as in the data-intensive workflows. The auto-scaling problem in the data-intensive context has not been sufficiently addressed yet and is the focus of this chapter. Intermediate data is the temporary data generated directly or indirectly from the input data but is not included in the final output. The aggregate amount of intermediate data could be in a ratio up to 10:1 or increases quadratically compared to the input data [97]. Though it is short-lived and may only be accessed by a limit number of tasks, it is critical to the successful completion of a client request. Moreover, as the data size increases, the intermediate data movement and storage decisions could dominate the whole application's performance and cost. This essentially requires that the auto-scaling solution should not only consider the number of VMs to process the workload but also consider the way to organize, transfer and store the intermediate data during the application execution.

Data scheduling has been extensively studied in the grid environment. While the previous research works have presented many ideas on reducing data transfer overhead and speeding up job execution, they are not sufficient enough to be applied directly in the cloud. The closest related work falls into three categories. (1) Some works [98][99] discussed the ideas of duplicating the data among geographically distributed computing sites to reduce data transfers. However the granularity is to map a whole workflow job to an execution site and the data considered for duplication is the input data not the intermediate data. (2) Some works discussed workflow scheduling with deadline and budget constraints [48][49][50]. Though the individual task scheduling is explicitly modeled and considered, however the data transfer performance and cost are not included. (3) Some other works explored data driven workflow scheduling based on data locality in a cluster environment [71][100]. They missed the capacity determination question to take advantage of the dynamic scalability feature in the cloud. In summary, to the best of our knowledge, this dissertation does not see resource provisioning solutions to minimize application cost within the deadline constraints which also considers the intermediate data management. In the previous related research, the job placement granularity is based on workflow jobs instead of sub tasks, the intermediate data management is basically ignored, and the deadline and cost constraints are not always included in the decision process.

In the previous three chapters, it either assumes that the intermediate data size is small enough to be ignored (tasks are all compute intensive), or as in the related work [55], it assumes the intermediate data is stored in a central storage system which can be accessed all the VMs. The data transfer (upload/download to/from the central storage, such as Amazon S3 [68]) time is considered as part of the task execution. These assumptions work well in practice. However, from the performance and cost perspectives, such design may slow down the application execution and incur more cost. Figure 6.1 is one example. If every intermediate data file needs to be staged in to the execution VM before the task starts and staged out to the cloud storage after the task finishes, the data transfer size will be 2x and the total cost will be all the data transfer in/out cost (for both VMs and cloud storage) plus the cloud storage cost - $cost^{out}_{VM1}(x)+cost^{in}_{cloud}(x)+cost^{store}_{cloud}(x)+cost^{out}_{cloud}(x)+cost^{in}_{VM2}(x)$. However, if the intermediate data can be stored in the local VM and transferred directly among the computing nodes, the data transfer size will be 1x and the total cost is reduced to $cost^{out}_{VM1}(x) + cost^{in}_{VM2}(x)$. Moreover, if two tasks sharing the same data can execute on the same machine, all the data transfer operations will be saved.

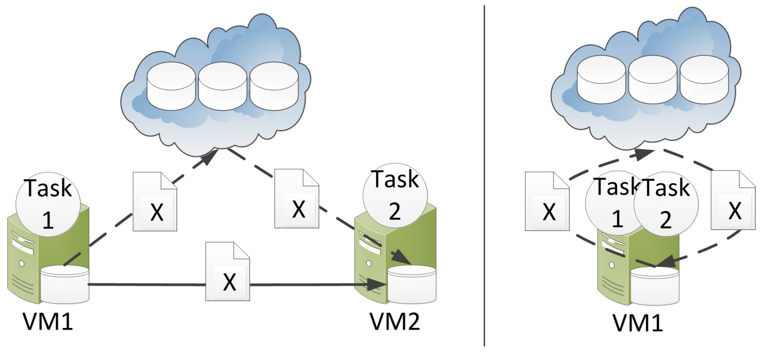

Figure 6.1: The benefits of storing intermediate data locally

Another important factor is the data locality. In figure 6.2, though the task has shorter execution time on VM1 (plan1) and longer execution time on VM2 (plan2), however, considering the data transfer penalty, plan 2 is in fact a better choice because faster execution on VM1 is offset by the data transfer overhead and it has overall shorter runtime on VM2. This example illustrates the importance of considering data locality in the job scheduling process.

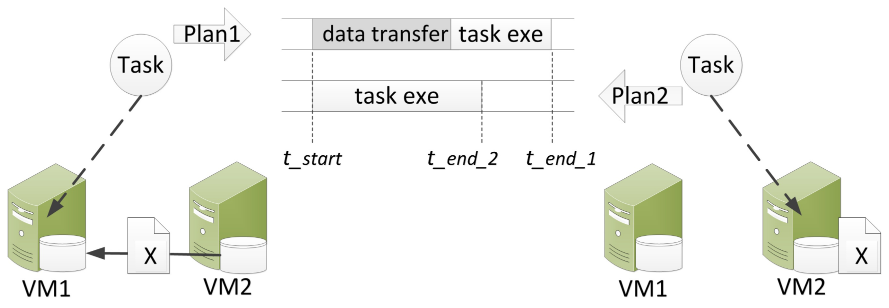

Figure 6.2: An example of data locality aware scheduling

For data-intensive applications, the auto-scaling problem becomes even more challenging because now it needs to answer three questions - resource provisioning, job scheduling and intermediate data management. There are three components in the circular reference problem that can affect each other. Particularly, how to incorporate the data locality information in the resource provisioning and allocation decisions and make the intermediate data management policies to cooperate are the two keys to the auto-scaling solution. This chapter will explicitly model the performance and cost for intermediate data transfer and storage. In addition to the data locality aware resource provisioning and allocation strategies, it presents an intermediate data management policy to minimize application cost while meeting job deadlines.

6.1 Problem Definition

Similar to the previous chapter, this chapter only details the assumptions that are different from the deadline-constrained auto-scaling problem (chapter 4) to highlight the unique aspects of this subproblem. The main change lies in the modeling of cloud storage, network and intermediate data.

Definition 1 (new) - Cloud Storage. There are two types of storage in the cloud - VM storage and cloud storage. VM storage is the storage associated with the acquired VMs (e.g., EC2 m1.small VM type has 160G instance storage [101]). The data stored on the VM disk is only accessible when the VM is active and disappears when the VM is terminated by the cloud application. VM storage does not cost extra money. Cloud storage is a shared central storage system offered by the cloud providers and is independent of the acquired VMs, such as Amazon S3 [68] and Google Cloud Storage [102]. The data in the cloud storage is persistent and high available, and can be accessed by all the VMs at all times. For cloud storage, this dissertation defines the storage cost as c_s per time quantum. The data transfer in cost is c_{cloud}^{in} per unit (e.g. GB) and the data transfer out is c_{cloud}^{out} per unit (e.g. GB). Similarly, the data transfer in and transfer out cost for a VM is $c_{VM_v}^{in}$ and $c_{VM_v}^{out}$. In practice, c^{in} and the data transfer cost within the same data center could be free as the cloud providers try to attract customers and promote their cloud storage services.

Definition 2 (new) - Network. Cloud storage and VMs are connected through network. The bandwidth of transferring data into the cloud storage is defined as b_{cloud}^{in} (e.g. GB per second) and the bandwidth of transferring data out is defined as b_{cloud}^{out}. Similarly, this dissertation defines the bandwidth for each VM type as $b_{VM_v}^{in}$ and $b_{VM_v}^{out}$. It also assumes the network transfer speed among the cloud storage and VMs are bounded by the smaller bandwidth. For example, the bandwidth between the ith VM type and the jth VM type is defined as b_{ij} and it is bounded by the smaller bandwidth of the two VMs, i.e. $b_{ij} = min(b_i^{in}, b_j^{out})$. Similar to the operator overlapping idea in [21], when there are n multiple uploaders or downloaders, the bandwidth for each individual client is $min(b_i^{in}/n, b_j^{out})$ or $min(b_i^{out}/n, b_j^{in})$.

Definition 3 (new) - Intermediate Data. One great difference and contribution of this chapter is that the performance and cost for the intermediate data storage and transfer are explicitly modeled in the resource provisioning and allocation process and it is not considered as part of the task execution. Specifically, three types of data are defined in the application. They are input data, intermediate data, and output data. The input data is directly consumed by a job, e.g. the raw data collected in the sensors. The output data is the final output of a job, such as the final sky image generated by the Montage worklfows [24]. Normally, the input and output data can only be stored at some specific locations because of the security considerations and legacy system designs. They are not the focus of this dissertation and their data transfer and storage policies are not included in the model. This dissertation focuses on optimized data placement strategies for the intermediate data. The intermediate data is the temporary data generated during the job execution, whose data transfer and storage decisions could dominate the application's performance and cost. An intermediate data file is defined as f_i and all the tasks that use f_i as inputs are defined as $suc(f_i)$. It needs to be transferred to the execution VM before a dependent task can start. This dissertation assumes the intermediate data will be removed automatically when it is no longer needed (i.e. all the tasks in $suc(f_i)$ are finished) [103]. The saved disk space can be used for newly generated temporary intermediate data.

Definition 5 (change) - Goal. The goal of the auto-scaling solution is to minimize the application cost while meeting job deadlines in the observation period. The cost includes three components. They are the computing cost which is the sum of all the acquired instance hours, cloud storage cost which is the cost incurred when the intermediate data is stored in the cloud storage, and data transfer cost which is the cost incurred when transferring the intermediate data among VMs and cloud storage.

$$Cost = \sum_{vm} c_{vm} + \sum_{f_i} c_s \times duration(f_i) + \sum_{f_i} (c_{cloud}^{in} + c_{cloud}^{out} \times ntimes + \sum_{vm} c_{vm}^{in} + \sum_{vm} c_{vm}^{out}) \times size(f_i) \quad (6.1)$$

Definition 6 (change) - Output. In addition to the scaling and scheduling decisions, the auto-scaling mechanism now needs to answer the third important question - data placement plan, i.e. determine the storage and transfer plan for every intermediate data file generated during the execution.

$$Scaling_t = \{VM_v \rightarrow N_v\} \quad (6.2)$$

$$Schedule_t = \{job_{J_j}^{S_i} \rightarrow VM_v\} \quad (6.3)$$

$$DataPlacement_t = \{f_i \rightarrow vm \quad or \quad cloud\ storage\} \quad (6.4)$$

6.2 Solution

For resource provisioning, this dissertation uses extended deadline assignment and load vector techniques to determine the VM number for each instance type, which also takes the data locality information into consideration compared to chapter 4. For job placement, it develops three strategies to schedule tasks on the acquired VMs. The strategies have different levels of consistency with the resource provisioning decisions. For intermediate data management, it designs a data prefetching strategy to fully utilize the VM local storage and reduce the waiting time for data transfers.

6.2.1 Resource Provisioning

This dissertation uses the deadline assignment and load vector [29][49][55] techniques to determine the size of the resource pool, i.e. the type and the number of the cloud VMs. The idea is to first assign deadlines to each task. If every single task can finish before its sub-deadline, the whole workflow job can then meet its deadline requirement. However different from chapter 4, instead of determining the task execution interval by using the ratio of deadline over job makespan, this chapter uses level-based deadline assignment. First, the task level $level(t_i)$ for task t_i is defined as equation 6.5, in which $pred(t_i)$ is the set of tasks that t_i depends on. The level of entry task $t_{S_{entry}}$ is 0.

$$level(t_i) = \begin{cases} 0, & if pred(t_i) = \emptyset \\ max_{p \in pred(t_i)} level(p) + 1 & otherwise \end{cases} \tag{6.5}$$

Next tasks are grouped by levels. For tasks at the same level that also share at least one same input, they can be grouped. The idea is that grouped tasks will be assumed to run on the same type of machines, therefore, they have a higher probability to run on the same machine and the data locality advantage can help to save the data transfer time and cost. Formally, tasks t_i and t_j can be grouped together if they belong to the same level and share one or more input files (equation 6.6). Task grouping combines tasks into task groups. Essentially it turns a set of nodes into a large node and treats them as a single task.

$$\begin{cases} t_i \in group_m \;\; \& \;\; t_j \in group_m & if \;\; level(t_i) = level(t_j) \;\; \& \;\; input(t_i) \cap input(t_j) \neq \emptyset \\ t_i \in group_n & if \;\; t_i, t_j \in group_m \;\; \& \;\; t_j \in group_n \end{cases} \tag{6.6}$$

For each VM type VM_v, the auto-scaling algorithm defines the sum of all the task execution times as the task group' execution time on VM_v (equation 6.7). In the deadline assignment step, for each task group it uses the VM_v with minimum cost as the default VM type (equation 6.8), and for each task level, it defines the longest task group execution time as the task level's execution time (equation 6.9).

$$exe(group_m, VM_v) = \sum_{t_i \in group_m} exe(t_i, VM_v) \tag{6.7}$$

$$exe(group_m) = min_{cost(group_m, VM_v)} \{exe(group_m, VM_v)\} \tag{6.8}$$

$$exe(level_i) = max_{group_m \in level_i}\{exe(group_m)\} \tag{6.9}$$

After determining the execution time for each level, sub-deadlines ($dl(level_i)$) can assigned to each level proportionally (equation 6.10). Compared to chapter 4, one important difference in this chapter is that deadline assignment needs to take data movement time into consideration. Therefore, the auto-scaling mechanism introduces a parameter - communication ratio (cr) - to represent the portion that data movement could take during the job execution. A smaller cr implies a more compute-intensive job and a larger cr implies a more data-intensive job. This parameter will be further discussed in the evaluation section.

$$dl(level_i) = (dl \times (1 - cr)) \times exe(level_i) / \sum_i exe(level_i) \tag{6.10}$$

Deadline assignment essentially determines an execution interval $[d_{start}, d_{end}]$ for each task. The load vector lv for task t_i is defined as $lv(t_i) = exe(t_i)/(d_{end} - d_{start})$. Intuitively, it means task t_i needs lv instance hours from d_{start} to d_{end} to finish before deadline. For each VM type, the load vectors of all the tasks are added together based on the time index (equation 6.11). The load vector describes the number of machines needed for any time interval. The overall resource provisioning strategy can be found in algorithm 8.

$$LV_{VM_v} = \sum_{t_i \rightarrow VM_v} exe(t_i)/(d_{end}(t_i) - d_{start}(t_i)) \tag{6.11}$$

Algorithm 8: VM scaling strategy

```
calculate $level(t_i)$ for each task $t_i$;
for $level_j = 0$ ... $level_{max}$ do
    for each task $t_i$ at $level_j$ do
        group tasks through $breadth-first-search$ and equation(6.6);
    end
    calculate $exe(level_j)$ based on equation(6.7, 6.8, 6.9);
end
initialize $loadVector_v = [0 ... 0]$ for all VM types $VM_v$ ;
for $level_j = 0$ ... $level_{max}$ do
    calculate deadline $dl(level_j)$ for $level_j$ based on equation 6.10 ;
    for task $t_i$ at $level_j$ do
        $lv(t_i) = exe(t_i, VM_v)/(dl(level_j) - dl(level - 1))$;
        $loadVector_{vmtype(t_i)} += lv(t_i)$;
    end
end
for each $VM_v$ do
    acquire $n = loadVector_v - num(VM_v)$ machines if $n > 0$;
end
```

6.2.2 Job Scheduling

When the VM instances are acquired and ready to use, tasks will be scheduled for execution. In this chapter, three job scheduling strategies are developed.

The first strategy is *deadline-first* (**DF**). Without considering the input file locations and the task execution time, deadline-first always schedules the ready task with the earliest deadline when a machine becomes idle. All the remote input files will be first transferred to the idle machine's local VM storage and then the task starts execution. In DF, the actual execution VM type can be different from the VM type planned in the resource provisioning phase (equation 6.8). It depends on the dynamic runtime information.

The second strategy is *planned-deadline-first* (**PDF**). Same as DF, planned-deadline-first always schedules the task with the earliest deadline first. However, the important difference is that the idle machine needs to be the same VM type as decided in the resource provisioning phase. Essentially, PDF could improve data locality and reduce the data transfer time. It schedules jobs in a more consistent way with the resource provisioning process.

The third strategy is *cost-deadline-first* (**CDF**). While DF always gives the top priority to the urgent tasks and PDF always first considers the VM type, CDF tries to strike a balance between the deadline and cost. The general idea is when a machine becomes idle, if there are urgent deadlines, the urgent tasks will be executed first, otherwise, the cheap tasks will be executed. Note, CDF works like PDF. It only allows tasks to run on the same VM type as determined in the resource provisioning phase. To determine whether urgent tasks exist, this dissertation introduces a parameter called urgency threshold u. If there are no tasks with urgent deadlines, i.e., $\forall t_i\ exe(t_i)/(dl(t_i) - now) < u$, the task with the smallest cost rank will be scheduled.

For an idle vm, the cost rank of a ready task t_i is defined as follows (equation 6.12). All the input files that are needed by t_i are defined as $input(t_i) = \{f_1, \ldots, f_j, \ldots, f_n\}$. The cost rank is calculated as the ratio of the sum of remote file size over the sum of all input file size. Essentially, CDF means the task which has the highest data locality ratio (smallest cost rank) will be executed when there are no urgent requests.

$$costRank(t_i) = \sum_{f_r \in input(t_i)\ \&\&\ f_r \notin vm} size(f_r) \Big/ \sum_{f_j \in input(t_i)} size(f_j) \tag{6.12}$$

For all the three job scheduling strategies, if a task may miss its deadline, it needs to acquire one instance by itself and starts its execution on the new instance. In other words, if the task has not been scheduled by its acquisition deadline $acquistionDL(t_i)$ (equation 6.13), a new machine (the same type as determined in the resource provisioning phase) will be started. When calculating the acquisition deadline, the intermediate data transfer time is also included. Through the $dl(t_i)/dl$ ratio, it gives later tasks in the workflower bigger time buffers, especially the last one, because it determines whether the whole job can be finished before deadline or not. The overall job scheduling strategy can be found in algorithm 9.

$$acquisitionDL(t_i) = dl(t_i) - size(input(t_i)) \times (dl(t_i)/dl)/b_{VM_v} - exe(t_i, VM_v) \tag{6.13}$$

Algorithm 9: Job scheduling strategy

```
if one vm becomes idle then
    pop the top ready task $t_i$ based on DF or PDF or CDF;
    move files $input(t_i)$ to $vm$ and skip all the files stored locally;
    execute task $t_i$ on $vm$;
end
for task $t_j$ in readyTasks do
    if currentTime >= acquisitionDL then
        acquire 1 $vm$ of $VM_v$ determined in equation(6.8);
        execute $t_j$ on $vm$;
    end
end
```

6.2.3 Intermediate Data Management

One important fact about the intermediate data is that it always needs to be transferred to the execution VM before the task starts. Therefore, an improvement that can be made is to prefetch the intermediate results asynchronously to the predicted execution VMs to actively increase the data locality and reduce the data transfer waiting time. In this way, the intermediate data prefectching strategy could in turn affect and cooperate with the resource provisioning and allocation process, especially for CDF. The data prefetching strategy needs to answer two questions (1) which intermediate file to transfer and (2) to where. For an intermediate file f_i on one VM instance, all the tasks that need to consume f_i are defined as $suc(f_i)$. Further this dissertation uses the earliest task deadline to define the intermediate data deadline (equation 6.14). The file with the earliest deadline will be first prefetched. The current implementation assumes every VM can prefetch the intermediate data simultaneously when the a task is running on it. However data prefetching has lower priority than the normal intermediate data transfer for task execution. It stops when a normal data transfer is started (either on the source or the target data prefetching VM) to save bandwidth. The reason is that task execution is more urgent and the data consumed by the task is required immediately, while data preteching is a complementary strategy and may not prefetch the right file to the right location.

$$dl(f_i) = min_{t_j \in suc(f_i)}\{dl(t_j)\} \tag{6.14}$$

When the prefetched file f_i is determined, it will choose the destination VM in the following way. The VM type VM_v will be the same as determined in the resource provisioning phase. Across all the VMs vm_j of

VM_v, it will calculate the total size of the files that belong to $\cup_{t_j \in suc(f_i)} input(t_j)$ on vm_j. f_i will be sent to the VM with largest file size. In other words, the data prefetching strategy prefers the machine that contains more related intermediate files. If there are two or more machines containing the same file size (e.g. 0), the VM with largest remaining disk storage will be chosen as the destination. Such strategy greatly increases the chances that grouped tasks run on the same machine and works consistently with the task grouping idea in the deadline assignment step. Both the execution cost and job runtime will be reduced.

$$VM_{maxsize(\cup_{t_j \in suc(f_i)} input(t_j))}$$

Like [103], this dissertation assumes the unreferenced intermediate files will be removed immediately, i.e. for f_i, all the tasks belonging to $suc(f_i)$ have finished. In this way, the local VM storage is saved for newly generated intermediate files. If the VM storage is still not large enough, the files with latest deadline will be pushed to the cloud storage. The logic is that files with earlier deadlines will be used in the future with a higher probability. It is a waste of time and money to upload it to the cloud storage first and then download it to the VM (figure 6.1). In such cases, cloud storage serves as an infinite backup storage space. The overall intermediate data management strategy can be found in algorithm 10.

Algorithm 10: Intermediate data placement strategy

```
sort all intermediate data by their deadlines;
choose file $f_i$ with the earliest deadline;
for each vm of $VM_v$ do
    $remain\_local\_size[vm]$ = calculate remaining disk size on $vm$;
    $sum\_file\_size[vm] = 0$;
    for $f_j$ in $\bigcup_{t_k \in suc(f_i)} input(t_k)$ do
        if $f_j \in vm$ then
            $sum\_file\_size[vm] += size(f_j)$;
        end
    end
end
if there are two vms with same max(sum_file_size) then
    $destinationVM = vm_{max(remain\_local\_size)}$;
else
    $destinationVM = vm_{max(sum\_file\_size)}$;
end
```

6.2.4 VM Shutdown

Different from chapter 4, instance shutdown needs to consider both the task execution and the intermediate data on an instance. There are two conditions to check. (1) When the VM instance is approaching the full hour operation, there are more active machines than the load vector determined in the scaling phase. (2) There is no unconsumed intermediate data on the VM or there are extra copies of the data on other VMs. If there does exist unconsumed data, a selection needs to be made between the cloud storage and VM storage. Essentially, it needs to compare the cost of the two options. The VM will not be terminated if it is cheaper to keep the data on the VM.

Assuming the file size is $size(f_i)$ and the estimated storage time is t_s. The cost of using the cloud storage and VM storage are defined in equations 6.15 and 6.16. When the cost of using cloud storage is cheaper, the local files will be transferred to the cloud storage and the VM is shutdown. Otherwise, the VM will be kept alive even there is no running tasks on it. Though the cloud storage cost is generally cheaper than the VM, however, if the file size is really large, the VM local storage may be preferred because of the data transfer overhead.

$$C_{cloud} = c_s \times size(f_i) \times t_s + (c_{VM_v}^{out} + c_{cloud}^{in}) \times size(f_i) + c_{VM_v} \times size(f_i)/b_{VM_v,cloud} \tag{6.15}$$

$$C_{VM} = c_{VM_v} \times t_s \tag{6.16}$$

6.3 Evaluation

6.3.1 Experiment Setup

Cloud and Application Simulator

The cloud is simulated using CloudSim [104], including five VM types and one cloud storage. Their configuration and prices are borrowed from the Amazon Web Service [6] and related performance studies [105][106]. The details are listed in table 6.1 and table 6.2. On top of the cloud, there are three components. They are workload generator, workflow engine and auto-scaling controller. The workflow engine accepts the submitted jobs from the workload generator, simulates the application execution. It also logs the job information, e.g. job finishing time.

Type	Price	CPU	Mem	Disk	Bandwidth
micro	$0.02/h	2 CU	613M	80G	50MBps, $0.15/G
small	$0.08/h	1 CU	1.7G	160G	50-60MBps, $0.15/G
high-CPU	$0.66/h	20 CU	7G	1690G	60-80MBps, $0.15/G
high-mem	$0.45/h	6.5 CU	17.1G	420G	60-80MBps, $0.15/G
extra-large	$1.3/h	33.5 CU	23G	1690G	80-100MBps, $0.15/G

Table 6.1: VM types

Storage	Data Transfer-in	Data Transfer-out	Transaction
$0.15/G-Month	60MBps, $0.15/G	60MBps, $0.15/G	$0.01/1000ops

Table 6.2: Cloud storage

Workflows and Workload

Three representative data-intensive workflows from [25] are used as the workload (figure 6.3). Montage [24] is used by NASA as an open source toolkit to generate custom mosaics of the sky. LIGO [107][108] is used to analyze the data obtained from the coalescing of compact binary systems by the Laser Interferometer Gravitational Wave Observatory. CyberShake [92] is used by the Southern California Earthquake Center to characterize earth quake hazards in a region. The workflow jobs are in the Directed Acyclic Graph in XML (DAX) format and generated the using the workflow generator from [25]. In addition to the three real-life synthetic workflows, this section continues to uses four representative workload patterns from the previous two chapters (figure 4.8). The four workload patterns are stable, cycle, growing and on-and-off. Each of these workloads represents a typical application or scenario. The workload generator constantly reads in the workflows jobs in the DAX format and submits the jobs to the workflow engine based on the four patterns.

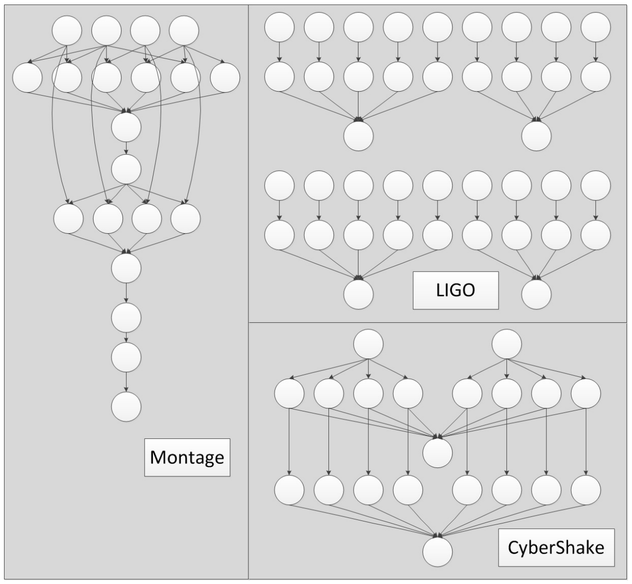

Figure 6.3: Data-intensive workflows

Observation Period

The auto-scaling controller monitors the job execution, determines the VM acquisition and release, and notifies the workflow engine job scheduling and data prefetching decisions. Different from other research works, one unique aspect of this dissertation is that the target is a continuously running application instead of a single job or job ensembles. In practice, it is hard to identify the start and the end of an application's life cycle. Therefore both the workload and the cost are essentially accumulated results in the observation period. This dissertation evaluates the performance of the auto-scaling solutions in a 72-hour period with 12 workflow-workload combinations (3 workflows and 4 workload patterns). However, since the four workload patterns show similar performance, this dissertation here only discusses the cycle workload pattern in the following sections, unless some workload pattern shows unique results. The cycle workload patterns submit 90 jobs per hour with the high peak around 180 jobs per hour and low peak around 120 jobs per hour. A complete cycle is 12 hours and there are 6 cycles in the 72-hour period.

6.3.2 DF vs PDF vs CDF

This dissertation first compares the performance of the three job scheduling algorithms - deadline-first (DF), planned-deadline-first (PDF) and cost-deadline-first (CDF) without data prefetching. For each job class, it assumes all the service components run on their fastest machines and calculates the job turnaround time as the baseline (1x) (the data transfer are considered as part of the task execution time). It then repeats the evaluation with four different deadlines - 1x, 2x, 3x and 4x to compare the auto-scaling mechanism's performance under different urgency settings. Particularly, the communication ratio (cr) is set to 0.4 [25][109] and the urgency threshold (u) is set to 0.7. This is the default setting in all the evaluations unless otherwise stated and these two parameters will be discussed in detail in the last sub section. The evaluation results of the three algorithms are shown in figure 6.4.

No matter the deadline is urgent or not, the DF algorithm always incurs more cost (12.9% - 50.3%) than the other two algorithms. The reason is that the scheduling policy is not consistent with the resource provisioning policy. A large portion of the tasks are not scheduled on the VMs as planed in the resource provisioning process. This mismatch makes more jobs miss the deadlines even a higher cost is incurred. In DF, a task may run on a more expensive but even slower machine and the data locality is totally ignored. On the contrary, PDF and CDF always incur less cost. Their scheduling strategies are consistent with the scaling decisions. When deadlines are less urgent (e.g. 3x and 4x), CDF shows better performance and it can save up to 29.1% cost compared to PDF. This is because CDF can explicitly compare the data transfer cost and choose the cheaper tasks to execute. Such advantage is clearer when more tasks fall below the

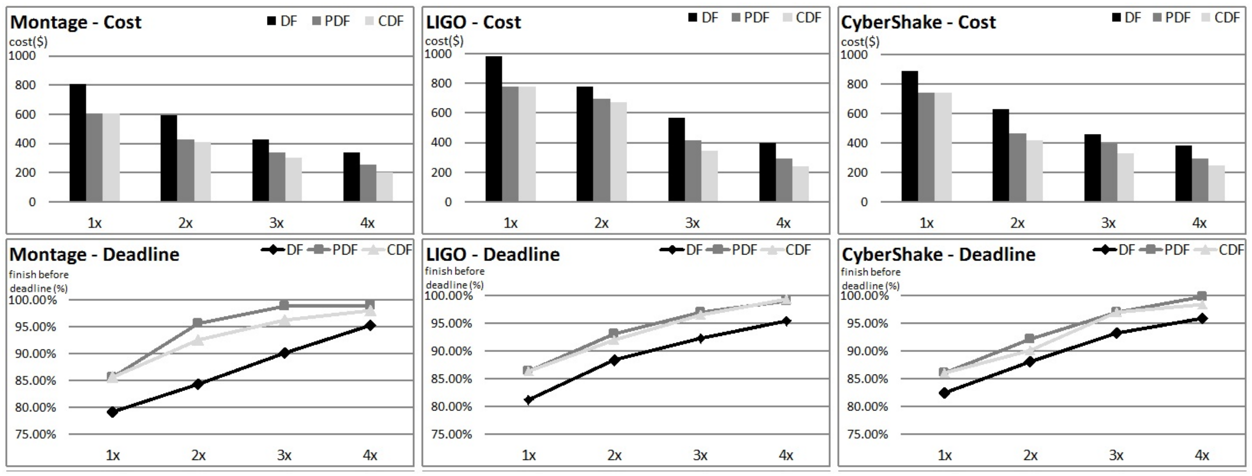

Figure 6.4: DF vs PDF vs CDF without data prefetching (Cycle)

urgency threshold (i.e. longer deadlines). These results confirm that resource provisioning and allocation are dependent on each other and they need to cooperate to achieve overall good performance. This is a very important principle when designing the auto-scaling mechanisms in the cloud. Moreover, data transfer cost plays an important role in data-intensive applications. Taking advantage of the data locality information can help to improve the auto-scaling mechanism's performance.

As shown in the lower half of figure 6.4, DF does not only incur the most cost and but also has the highest deadline miss rate. Though urgent tasks are always scheduled first, it does not mean they are always finished early. The immediately available machine could in fact be a slower machine because the VM type is not suitable for the task and it has low data locality. In other words, urgent tasks have shortest waiting time in DF but they may have overall longer runtime compared to PDF and CDF. When the deadline is urgent, CDF and PDF essentially work in the same way, therefore, the two algorithms also have the same deadline miss rate. When the deadlines are less urgent (e.g. 3x and 4x), PDF can finish slightly more jobs before their deadlines than CDF. This is because PDF always gives top priority to the urgent tasks, while CDF starts to gives more opportunities to the cheap tasks, which may cause deadline misses. Note, though CDF can make cheaper tasks run faster by considering data locality, it still cannot beat PDF in terms of deadline misses. This is a trade-off between deadline and cost which will be further discussed in the later sub section. Note, for all the three algorithms, the deadline miss rate is pretty high (above 20%) when the deadline is 1x. This is because 1x is really a tight deadline setting which is the lower bound of the job makespan. When deadlines become less urgent, the deadline miss rate of all the three algorithms are largely reduced (more than 95% jobs can finish before deadlines). The mechanism's acquisition deadline can help to find the most urgent tasks facing deadline misses and acquire new machines as long as it has enough response time.

To quantify the consistency between the resource provisioning and allocation process, this dissertation calculates the VM type match rate in the scaling and schedule phases. For a task, if it is executed on a VM type as planed in the scaling phase, it is considered as a match. Table 6.3 shows the match rate for the montage application with two different deadlines (1x and 3x). For DF, longer deadlines have higher match rate because fewer VMs are running at the same time. PDF and CDF always have 100% match rate because the tasks are only allowed to run on the same VM type. Further, this dissertation calculates the average data locality ratio ($\sum size(f_{local}) / \sum size(f_{all})$). DF has the lowest data locality ratio for both deadlines. While PDF and CDF have the same data locality ratio when the deadline is urgent, CDF can improve more than PDF when deadline is less urgent because of the cost rank based scheduling idea. These numbers explains the importance of decision consistency and data locality awareness in the resource provisioning and allocation process and why PDF and CDF have better performance than DF.

	DF	PDF	CDF
1x deadline	38.2%	100%	100%
3x deadline	54.6%	100%	100%

Table 6.3: VM type match rate in scaling and scheduling (Montage)

	DF	PDF	CDF
1x deadline	13.3%	45.2%	45.2%
3x deadline	19.5%	54.6%	62.3%

Table 6.4: Data locality ratio (Montage)

6.3.3 Data Prefetching

Data prefetching moves the intermediate data to the task execution VM in advance to save data transfer time. It also cooperates with the resource provisioning and allocation process to schedule jobs based on the data locality. This section quantifies the benefits of data prefetching for all the three job scheduling strategies. The evaluation results (same tests as the previous section) with data prefetching strategies are shown in figure 6.5.

For all the three job scheduling algorithms, data prefetching helps to make more jobs finish before their deadlines. The improvement ranges from 2.0% to 8.7%. Such improvement results from asynchronous data transfer which gives more time buffer to the task execution. Moreover, PDF and CDF benefit more from data prefetching than DF. This is because the data prefetching strategy only moves the files to the VM types as determined in the deadline assignment phase. PDF and CDF therefore have higher consistency while many data prefetches in DF are wasted because the task is executed on a different VM type. When deadlines are less urgent, both PDF and CDF can finish more than 97% jobs before their deadlines. However, CDF

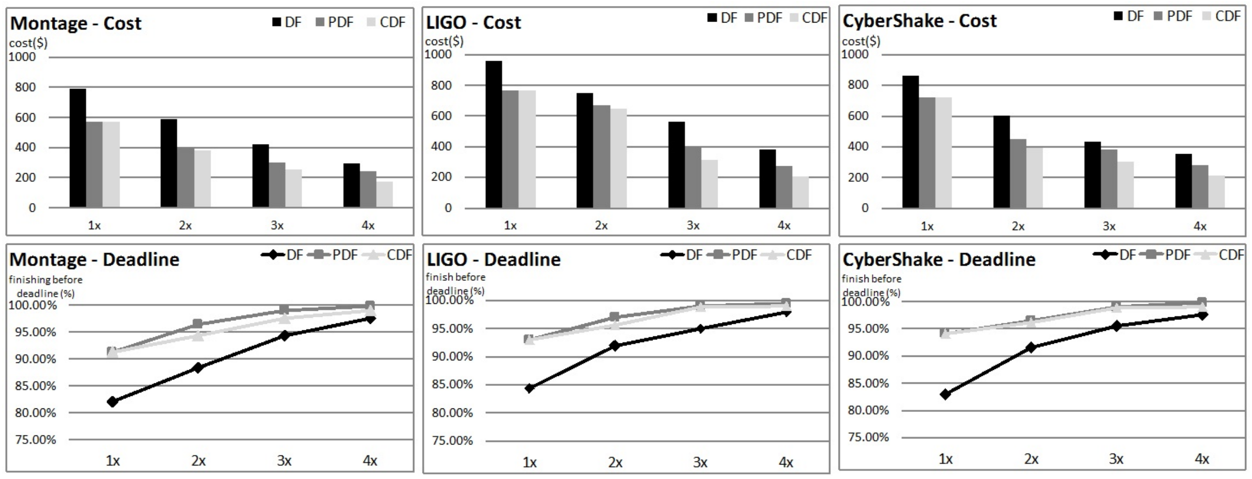

Figure 6.5: DF vs PDF vs CDF with data prefetching (Cycle)

can save more cost than PDF and this cost saving benefit is even bigger than the previous section (no data pretching). This is because data pretching always chooses the VM with the highest data locality ratio as the target VM, which is the same as the CDF cost rank idea. Such consistency therefore can help CDF to maximize its cost saving benefits. This test shows that the data preteching strategy needs to be consistent with the resource scaling decisions (DF vs PDF&CDF) and also needs to be consistent with the job scheduling decisions (PDF vs CDF) to maximize its benefits of reducing job turnaround time and saving cost.

One important fact about data prefetching is that the anticipated target machine may not be always “correct” (the same as the actual task execution machine). Particularly, this dissertation defines that it is a successful data prefetch if the data is consumed by at least one dependent task running on the target machine. Table 6.5 shows the rate of successful prefetching for the montage application with 1x and 3x deadlines. Clearly, CDF has the highest data prefetch success rate, this number confirms the consistency between the job scheduling and data prefetching processes. Similar to the previous section, the data locality ratio is also calculated. The results (table 6.6) show that the data prefetching strategy can improve the data locality ratio for all the algorithms. When the deadline is less urgent (3x), CDF’s data locality ratio is improved for the most.

	DF	PDF	CDF
1x deadline	15.1%	22.7%	24.8%
3x deadline	18.4%	32.6%	42.1%

Table 6.5: Rate of successful data prefetching (Montage)

	DF	PDF	CDF
1x deadline	20.4%	48.2%	48.2%
3x deadline	28.2%	63.7%	74.3%

Table 6.6: Data locality ratio with data prefetching (Montage)

6.3.4 The Benefits of VM Storage

One contribution of this dissertation is to take advantages of the VM local storage in the auto-scaling solution. Storing the intermediate data on the VM local storage can help to save cloud storage space and data transfers, and also improve the data locality. This section shows the benefits of VM storage. In many related works, the intermediate results are assumed staged in to the local VMs from the cloud storage before the task starts and staged back to the cloud storage when the task finishes. In this way, the system implementation is simple and the application does not need to worry about the limited disk space. However such design does not only suffer from the performance degradation due to the large amount of data transfers between the VM and cloud storage, but also pays extra for the storage and bandwidth. Figure 6.6 compares this design idea (referred as "Cloud Storage Only") with this dissertation (referred as "VM Storage and Cloud Storage") for all the three applications with PDF scheduling policy (without data prefetching) and 3x deadline. It shows that using the VM local storage can save up to 47.1% cost (LIGO case) compared to the cloud storage only design. The cost increase largely depends on the amount and duration of the intermediate data. In fact, in the cloud storage only design, the data stage-in and -out transfer also reduces the time allocated for task execution which forces more tasks to run on faster and expensive machines. It therefore increases the overall application cost. Therefore, the application that generates the most intermediate data - LIGO - suffers the most in both the performance (7.1% deadline miss rate increase) and cost.

In practice, the cloud providers may offer free data transfers among the VMs and the cloud storage within the same data center. Therefore, this experiment also compares the two designs with 0 data transfer cost (the two bars on the right for each application in figure 6.6). It is shown that the free data transfer can save around 20% cost and the difference between the two design ideas is smaller. This confirms that data transfer is a large cost component for the overall application cost. The cloud storage offers infinite and high available storage space, and can be easily accessed by the application clients. However from the cost perspective, cloud storage may not be a good option to store the short-lived intermediate data, even the data transfer cost can be ignored. It may also incur more job deadline misses because more time has to be spent on the data transfer. The VM local storage capabilities should be carefully planned and utilized.

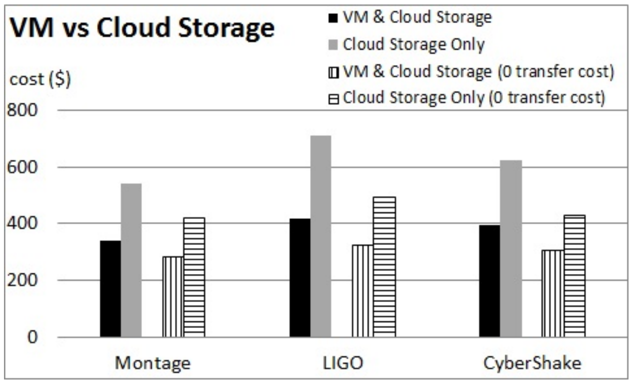

Figure 6.6: VM storage vs Cloud storage

6.3.5 Communication Ratio & Urgency Threshold

In the resource provisioning phase, the deadline assignment step considers the ratio of compute time and data movement time, and assigns deadlines based on the compute time only (see equation 6.10). In this way, it purposely allocates time buffers for transferring the intermediate data. A large cr implies a data-intensive job while a smaller cr implies a compute-intensive job. Though cr can be collected and estimated from past execution history, it may not always accurately reflect the ratio between communication and computation. This test (figure 6.7) shows that data prefetching can help to handle inaccurate settings of this communication ratio parameter. When the application developers simply ignores the data transfer time (e.g. a small cr setting - 0.1, 2x deadline for Montage), it is shown that the deadline miss rate can reach 52.3% percent (only 47.7% jobs can be finished in time). The reason is that tasks are scheduled on the cheaper and slower machines because a larger execution interval is allowed. The unexpected data movement overhead could greatly delay the task execution and cause deadline misses. A small cr setting certainly helps to save cost, but it can largely increase the number of deadline misses as well. When a larger cr is set, tasks are given shorter execution intervals and forced to run on faster machines. In such cases, the application cost is increased because more expensive machines are acquired. The data prefetching idea can help to alleviate this inaccurate estimation problem. The asynchronous data transfer could buy more time for task execution. For example, when cr is set a small value (e.g. 0.1), data prefetching can help another 25.3% jobs to finish before the deadline. As cr goes large, the benefit of reducing deadline misses becomes smaller, however it could still save cost because more machines can be shutdown and cheaper machines can be used. Essentially, communication ratio can be used to help service providers to speed up the application execution.

Urgency threshold (u) reflects the application's preference between the performance and cost. When u is small, CDF essentially behaves in the same way as PDF, data locality is largely ignored. In such cases, nearly all the jobs (more than 99% jobs) can finish before their deadlines, because urgent tasks are always given the top priority. However the application cost is also the highest because the data locality advantages are not taken and more machines are acquired because of the acquisition deadline misses. When u is set to a large number (i.e. prefer cost saving to performance), the cost is reduced but more jobs can miss their deadlines. As u increases, the benefit of data prefetching on cost saving also becomes clearer, because more tasks can take advantage of the data locality aware scheduling. The urgency threshold setting offers the service providers opportunities to make trade-offs between the performance and cost.

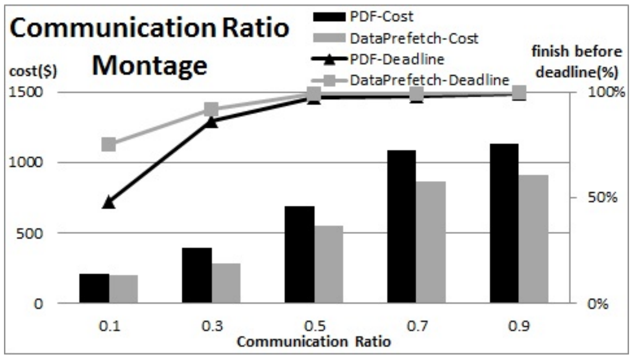

Figure 6.7: Communication ratio

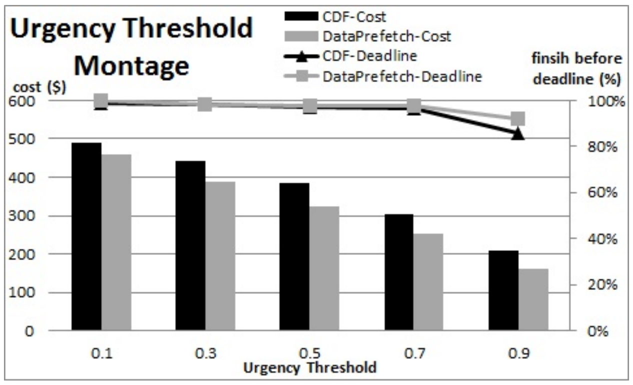

Figure 6.8: Urgency threshold

6.4 Conclusion

Data-intensive applications are playing a very important role in both the business world and scientific community. During application execution, a large volume of intermediate data can be generated. The intermediate data management decisions can therefore dominate the whole application's performance and cost. This chapter designs an auto-scaling solution for data-intensive applications. The major contribution of this chapter includes (1) It extends the workflow application model (chapter 4 and 5) to the data-intensive context. (2) It designs an auto-scaling solution that incorporates the resource provisioning, job scheduling and intermediate data management processes. (3) The data prefetching strategy can take advantage of the VM local storage and data locality information to reduce data transfer time and cost. It also helps more jobs to finish before the deadlines through the asynchronous data movement. The evaluation results show the importance of the consistency among the three subproblems. PDF works better than DF because jobs are always executed on the same VM type as determined in the resource provisioning phase. CDF works better than PDF because it explicitly considers the data locality information to further reduce the data transfer time and cost. Particularly, data prefetching is most beneficial to CDF because it prefetches the intermediate data based on data locality which shares the same idea with CDF's cost rank calculation. Moreover, from the cost perspective, VM local storage can help to save cloud storage cost and unnecessary data transfers. Data prefetching can help to handle inaccurate communication ratio settings (especially for smaller settings) and urgency threshold allows service providers to make trade-offs between performance and cost.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The cloud has become an important computing platform and attracted many businesses and individual users. It offers users on-demand resources and saves unnecessary capital investment. The maintenance cost is largely reduced. In the cloud adoption process, dynamic scalability and low cost are the two key enablers and benefits. However this does not mean that the cloud adoption process is simple and the dynamic scalability feature can be used in an appropriate way. Only when users can correctly determine the computing capacity and utilize the cloud resources in a cost-efficient way, can the advantages of the cloud be realized. While resource over-provisioning may cost users more than necessary and offset cloud cost-saving advantages, resource under-provisioning can hurt application performance and turn away customers. This dissertation targets this important resource provisioning and allocation problem in the cloud through auto-scaling solutions.

The auto-scaling solutions assume the applications run on the IaaS cloud, which offers the most general and flexible compute resources. Cloud VMs are with different hardware configurations (e.g. CPU, memory, disk, etc.) and they are treated as the most fine-grained scalable computing capacity in this dissertation. Cloud VMs have different prices and they are not necessarily priced linearly to their computing power. The OS images can be customized and saved. They serve as the templates for dynamically scalable components. The cloud storage serves as a persistent, high available storage system that can be shared by all the clients.

This dissertation considers two types of application architectures. They are batch-queue based applications and SOA based applications. The batch-queue based applications accept bags-of-tasks as the workload, while SOA based applications accept workflow jobs, such as complex business processes and scientific workflows. These two types of applications are general and abstract enough to cover most application architectures and it is challenging to handle task dependencies in workflow jobs.

The cloud makes the divisions of resource providers and users to be even more fine-grained. This dissertation defines three roles in the cloud usage scenarios - cloud providers, service providers and service customers. The services providers build their applications using the cloud resources. They only need to focus on their expertise and the areas of interest. Depending on the nature of the services and the goals of the service providers, this dissertation specifically deals with two cases - the unlimited budget case and limited budget case.

The two dimensions of the auto-scaling problems are deadline and cost. Deadline is not treated as hard deadlines in the real-time systems. Deadline misses are allowed. Deadline is an indication of service customers' preference on the job completion time and reflects the application execution speed. Cost is one of the most important factors when service providers develop and migrate their applications in the cloud. It serves as either the goal or the constraint in the four subproblems. Meeting deadlines with minimum cost and maximizing application performance within the budget constraints are the two problems this dissertation solves.

Chapter 3 solves the auto-scaling problem for batch-queue based cloud applications using integer programming techniques. Chapter 4 presents a dynamic scaling and scheduling solution to minimize application cost within the deadline constraints for cloud workflows. The solution assigns deadlines to sub tasks and uses load vectors to represent the computing capacity needed. Chapter 5 presents the scaling-first and scheduling-first algorithms to maximize application performance within the budget constraints. The two algorithms make resource provisioning and resource allocation decisions in different orders. Chapter 6 explicitly considers the performance and cost of the data transfer and storage in the auto-scaling problem. It develops a data prefetching strategy to manage the intermediate data.

Cloud workload may prefer different types of VMs. Provisioning cost-efficient VMs is a very important principle to speed up job execution and reduce application cost. Choosing the cheapest machines cannot always help to save cost. The right metric should consider task specific execution time and cost.

Load vector is a simple and innovative technique to represent the computing capacity needed. While deadline assignment breaks task dependencies, it includes both the task waiting time and execution time information. The communication ratio setting allows users to specify the degree of the data-intensiveness for an application and the urgency ratio enables users to express their preference between the performance and cost.

Instance consolidation process is a good complementary strategy to improve resource utilization and reduce partial instance hour waste. It is an efficient process to alleviate the instance hour fragmentation problem and enables the auto-scaling mechanism to handle both the high and low volume workload.

The dynamic nature of monitor-control loop enables the auto-scaling mechanism to make the resource provisioning and allocation decisions based on the updated information. It could find the urgent tasks and longer than expected VM startup delays in time to make sure the resource is always adequate. Therefore, it also helps to handle the inaccurate parameters.

The three questions to be answered by an auto-scaling solution is resource provisioning (capacity determination), resource allocation (job scheduling) and intermediate data management (data placement). The three decisions are a circular reference problem, which are dependent on each other. The more consistent they are, the more cost can be saved and the more the job turnaround time can be reduced. In addition, the order of making resource provisioning and allocation decisions can also affect the auto-scaling mechanism's performance. The scaling-first algorithm works better in low budget ranges while the scheduling-first algorithm works better in high budget ranges.

Data prefetching is an efficient intermediate data management strategy to reduce data transfer time and cost. It can affect the resource provisioning and allocation decisions as well. The local VM storage proves to be a good option to store temporary and short-lived intermediate data. It saves unnecessary data uploading and downloading to the central storage and enables the auto-scaling mechanism to take advantage of the data locality information.

7.2 Limitation and Future Work

Other application goals In addition to deadlines, job turnaround time and cost, there could be other performance metrics that service providers and customers are interested in, such as throughput - the number of the completed jobs should be maximized; utilization - the percentage of the idle resources cannot be higher than a threshold; satisfaction - all the service requests should be finished in a way to maximize the overall customer satisfaction; utility - the accumulated application profits should be maximized in the observation period. The performance metrics can be very application specific.

Cost model In addition to the cost per time quantum (dollars per hour) pricing scheme, cloud providers also have other pricing models. One of them is the reserved instance billing model. It works like a subscription service offered by the phone companies. Cloud users pay a fixed cost in advance and get a discount on the service usage. In this way the long term price of the cloud resources is reduced compared to the regular on-demand resources. Another billing model is the auction market. Essentially, cloud users could bid for the cloud resources. If the real-time price (varies depending on demand and supply) is lower than the bidding price, cloud users could gain the access to the virtual machines. Such virtual machines are called spot instances. Spot instances can lower the computing costs for time-flexible, interruption-tolerant tasks (e.g. web-crawling or Monte Carlo simulations), because spot prices are often significantly less than on-demand prices for the same instance types. Extending the current solutions to support these cost models is an interesting direction. The application cost can be further reduced because of the cheaper resource price. However, the non-deterministic workload and VM availability make the resource provisioning decisions more difficult to make.

Cost-constrained data-intensive applications In chapter 6, this dissertation solves the auto-scaling problem for deadline-constrained data-intensive cloud applications. The next step is to solve the other dimension of the optimization problem - maximizing application performance for cost-constrained applications in the data-intensive context.

Hybrid cloud and federated cloud For security concern and legacy system design reasons, cloud applications may need to keep some in-house computing infrastructure. They route jobs to the cloud when the local resource is not capable to handle the workload to meet their service level agreements. This dissertation assumes that all the resources used to build the cloud application come from the cloud providers. There is no interaction and workload distribution between the local resources and cloud resources. The next step of this work is to extend this research into the hybrid cloud computing environment. Moreover, another extension direction is for a multi-cloud environment, the so called federated cloud.

Hadoop support Hadoop has become a very important computing framework to process data-intensive jobs. It has its own file system called HDFS. Though HDFS can be built on VM local storage and offers high availability, it does not consider the job performance and application cost. Therefore, the resource provisioning mechanisms and the data prefetching strategy presented in chapter 6 is a good complement to enable Hadoop jobs to be deadline- and cost-aware. Essentially, such extension adds one more factor in the optimization problem. It is going to be a trade-off among application cost, job turnaround time and service availability. This can also be considered as the fault-tolerance support in the auto-scaling solution.

Open source the code Chapter 3 introduces the architecture and implementation of an auto-scaling solution. The next step is to make this implementation and all the resource provisioning and allocation algorithms publicly available. The auto-scaling library could help the research community to develop new ideas and help the cloud practitioners to uses cloud resources more efficiently.

Workload prediction While the monitor-control structure of the presented auto-scaling solutions can handle the dynamic workload well, better scaling and scheduling decisions can be made if the workload information is known in advance. Suitable VMs can be chosen to accommodate incoming workload types. Budget can be distributed among individual jobs in a more optimized way. Instances can be acquired in advance and unnecessary VM shutdown can be avoided to alleviate the VM startup delay problem.

Evaluation benchmark The resource provisioning and allocation problem in the cloud is a hot topic. This dissertation has seen quite a few ongoing research projects. While it is interesting to see many innovative ideas and different perspectives, one great difficulty is to compare their performance. It is hard to tell the advantages and disadvantages for each individual algorithm in a quantitative way. In addition to the subtle assumption differences, the evaluation is performed in different environments (e.g. local cluster, real clouds and simulation), with different benchmarks (e.g. workload traces, synthetic workload and random numbers) and measured in different metrics. The research community is missing consistent and agreed benchmarks and evaluation platforms to conduct the research. Therefore a benchmark and evaluation library can be developed to help to compare the algorithm performance and speed up the research process.

Bibliography

[1] P. Mell and T. Grance. The NIST Defnition of Cloud Computing. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[2] F. Gens. IDC Predictions 2012: Competing for 2020. http://www.idc.com/getdoc.jsp?containerId=231720.

[3] J. Gantz, S. Minton, and A. Toncheva. Cloud Computing's Role in Job Creation. http://www.microsoft.com/en-us/news/download/features/2012/IDC_Cloud_jobs_White_Paper.pdf.

[4] F. Gens, M. Adam, M. Ahorlu, D. Bradshaw, L. DuBois, M. Eastwood, T. Grieser, S. Hendrick, V. Kroa, R. Mahowald, S. Matsumoto, C. Morris, R. Villars, R. Villate, N. Wallis, and A. Florean. Worldwide and Regional Public IT Cloud Services 20122016 Forecast. http://www.idc.com/getdoc.jsp?containerId=236552.

[5] Gartner. Gartner's Top Predictions for IT Organizations and Users, 2012 and Beyond: Control Slips Away. http://www.businesswire.com/news/home/20111201005541/en/Gartner-Reveals-Top-Predictions-Organizations-Users-2012.

[6] Amazon EC2. http://aws.amazon.com/ec2.

[7] Amazon Web Service Case Studies. http://aws.amazon.com/solutions/case-studies/.

[8] The future of cloud computing: 9 trends for 2012. http://www.zdnet.com/blog/btl/the-future-of-cloud-computing-9-trends-for-2012/80511.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[10] D. F. Parkhill. *The challenge of the computer utility*. Addison-Wesley Professional, USA, 1966.

[11] Rackspace. http://www.rackspace.com.

[12] Google Compute Engine. https://cloud.google.com/products/compute-engine.

[13] Windows Azure. http://www.windowsazure.com.

[14] Google App Engine. https://cloud.google.com/appengine.

[15] Scaleforce. http://www.salesforce.com.

[16] Google Apps. http://www.google.com/intl/en/enterprise/apps/.

[17] Microsoft Office 365. http://www.microsoft.com/en-us/office365.

[18] Cloud computing, X-as-a-Service. http://en.wikipedia.org/wiki/Cloud_computing.

[19] G. Juve and E. Deelman. Resource provisioning options for large-scale scientific workflows. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 608 –613, dec. 2008.

[20] M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 41 –48, oct. 2010.

[21] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis. Schedule optimization for data processing flows on the cloud. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 289–300, New York, NY, USA, 2011. ACM.

[22] J. Li, M. Humphrey, D. Agarwal, K. Jackson, C. van Ingen, and Y. Ryu. escience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –10, april 2010.

[23] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995. 10.1007/BF01277643.

[24] J. Jacob, D. Katz, B. Berriman, J. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M. Su, T. Prince, and R. Williams. Montage a grid portal and software toolkit for science grade astronomical image mosaicking. *Int. J. Comput. Sci. Eng.*, 4(2):73–87, July 2009.

[25] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1 –10, nov. 2008.

[26] Netflix. http://www.netflix.com.

[27] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM JOURNAL ON APPLIED MATHEMATICS*, 17(2):416–429, 1969.

[28] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1:117–129, 1976.

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

[30] M. Mao and M. Humphrey. Scaling and scheduling to maximize application performance with budget constraints in cloud workflows. submitted.

[31] M. Mao and M. Humphrey. Resource provisioning and intermediate data managemetn for workflows on IaaS cloud. in preparation.

[32] I. Foster. What is the Grid? A Three Point Checklist. http://dlib.cs.odu.edu/WhatIsTheGrid.pdf.

[33] P. Plaszczak and R. Wellner. *Grid Computing*. Elsevier/Morgan Kaufmann.

[34] IBM White Paper. IBM Solutions Grid for Business Partners: Helping IBM Business Partners to Grid-enable applications for the next phase of e-business on demand. http://joung.im.ntu.edu.tw/teaching/distributed_systems/documents/IBM_grid_wp.pdf.

[35] R. Buyya and S. Venugopal. A Gentle Introduction to Grid Computing and Technologies. http://www.buyya.com/papers/GridIntro-CSI2005.pdf, 2002.

[36] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, nov. 2008.

[37] K. Aberer. P-grid: A self-organizing access structure for P2P information systems. In *Proceedings of the 9th International Conference on Cooperative Information Systems*, CooplS '01, pages 179–194, London, UK, UK, 2001. Springer-Verlag.

[38] R.W. Moore, M. Wan, and A. Rajasekar. Storage resource broker; generic software infrastructure for managing globally distributed data. In *Local to Global Data Interoperability - Challenges and Technologies, 2005*, pages 65 – 69, june 2005.

[39] M. Mao and M. Humphrey. A performance study on the VM startup time in the cloud. *2012 IEEE Fifth International Conference on Cloud Computing*, 0:423–430, 2012.

[40] M. Wieczorek, A. Hoheisel, and R. Prodan. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Gener. Comput. Syst.*, 25(3):237–256, March 2009.

[41] H. Zhao and R. Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 14 pp., april 2006.

[42] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Grid resource management. chapter Workflow management in GriPhyN, pages 99–116. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[43] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings of the 9th Heterogeneous Computing Workshop*, HCW '00, pages 349–, Washington, DC, USA, 2000. IEEE Computer Society.

[44] R. Sakellariou and H. Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111, april 2004.

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

[46] I. Brandic, S. Benkner, G. Engelbrecht, and R. Schmidt. QoS support for time-critical grid workflow applications. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8 pp. –115, july 2005.

[47] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005.

[48] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos. Scheduling workflows with budget constraints. In *in Integrated Research in Grid Computing, S. Gorlatch and M. Danelutto, Eds.: CoreGrid series*. Springer-Verlag, 2007.

[49] J. Yu, R. Buyya, and K. T. Chen. Cost-based scheduling of scientific workflow applications on utility grids. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8 pp. –147, july 2005.

[50] D.A. Menasce and E. Casalicchio. A framework for resource allocation in grid computing. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 259 – 267, oct. 2004.

[51] S. Abrishami, M. Naghibzadeh, and D. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, 29(1):158 – 169, 2013. Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures.

[52] P. Marshall, K. Keahey, and T. Freeman. Elastic site: Using clouds to elastically extend site resources. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 43 –52, may 2010.

[53] Nimbus. http://www.nimbusproject.org/.

[54] M. Xu, L. Cui, H. Wang, and Y. Bi. A multiple QoS constrained scheduling strategy of multiple workflows for cloud computing. In *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pages 629 –634, aug. 2009.

[55] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. In *Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12, New York, NY, USA, 2012. ACM.

[56] L. Wu, S.K. Garg, and R. Buyya. Sla-based resource allocation for Software as a Service Provider (SaaS) in cloud computing environments. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 195 –204, may 2011.

[57] D. Villegas, A. Antoniou, S.M. Sadjadi, and A. Iosup. An analysis of provisioning and allocation policies for Infrastructure-as-a-Service clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 612 –619, may 2012.

[58] J. Chen, C. Wang, B. Zhou, L. Sun, Y. C. Lee, and A. Y. Zomaya. Tradeoffs between profit and customer satisfaction for service provisioning in the cloud. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 229–238, New York, NY, USA, 2011. ACM.

[59] Y. C. Lee, C. Wang, A. Y. Zomaya, and B. Zhou. Profit-driven service request scheduling in clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 15 –24, may 2010.

[60] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove. Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 228–235, july 2010.

[61] S. Pandey, A. Barker, K.K. Gupta, and R. Buyya. Minimizing execution costs when using globally distributed cloud services. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 222 –229, april 2010.

[62] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 304–307, New York, NY, USA, 2010. ACM.

[63] T. Dornemann, E. Juhnke, and B. Freisleben. On-demand resource provisioning for BPEL workflows using Amazon's elastic compute cloud. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 140–147, Washington, DC, USA, 2009. IEEE Computer Society.

[64] T. Do andrnemann, E. Juhnke, T. Noll, D. Seiler, and B. Freisleben. Data flow driven scheduling of BPEL workflows using cloud resources. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 196 –203, july 2010.

[65] T. Bicer, D. Chiu, and G. Agrawal. Time and cost sensitive data-intensive computing on hybrid clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 636 –643, may 2012.

[66] M.D. De Assuncao, A. di Costanzo, and R. Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 141–150, New York, NY, USA, 2009. ACM.

[67] R.N. Calheiros and R. Buyya. Cost-effective provisioning and scheduling of deadline-constrained applications in hybrid clouds. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 320–327, Washington, DC, USA, 2011. IEEE Computer Society.

[68] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[69] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin. Improving mapreduce performance through data placement in heterogeneous Hadoop clusters. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –9, april 2010.

[70] D. Yuan, Y. Yang, X. Liu, and J. Chen. A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[71] S. Shankar and D.J. DeWitt. Data driven workflow planning in cluster management systems. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC '07, pages 127–136, New York, NY, USA, 2007. ACM.

[72] RightScale. http://www.rightscale.com.

[73] enStratus. http://www.enstratus.com.

[74] Scalr. http://www.scalr.net.

[75] Microsoft Solver Foundation. http://archive.msdn.microsoft.com/solverfoundation.

[76] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of Windows Azure. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 367–376, New York, NY, USA, 2010. ACM.

[77] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace–a toolset for the performance prediction of parallel and distributed systems. *Int. J. High Perform. Comput. Appl.*, 14(3):228–251, August 2000.

[78] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, Jignesh M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan. New grid scheduling and rescheduling methods in the GrADS project. *Int. J. Parallel Program.*, 33(2):209–229, June 2005.

[79] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '98, pages 122–142, London, UK, UK, 1998. Springer-Verlag.

[80] B. Chun and D. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02, pages 30–, Washington, DC, USA, 2002. IEEE Computer Society.

[81] B. Chun and D. Culler. Market-based proportional resource sharing for clusters. Technical report, Berkeley, CA, USA, 2000.

[82] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3):169–182, August 2005.

[83] F. Popovici and J. Wilkes. Profitable services in an uncertain world. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 36–, Washington, DC, USA, 2005. IEEE Computer Society.

[84] J. Sherwani, N. Ali, N. Lotia, Z. Hayat, and R. Buyya. Libra: a computational economy-based job scheduling system for clusters. *Softw. Pract. Exper.*, 34(6):573–590, May 2004.

[85] Earliest deadline first scheduling. http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling.

[86] Cloud Workload Patterns. http://www.microsoftpdc.com/2009/SVC54.

[87] A. Iosup and D. Epema. Grid computing workloads. *Internet Computing, IEEE*, 15(2):19 –26, march-april 2011.

[88] S. Ostermann, R. Prodan, T. Fahringer, R. Iosup, and D. Epema. On the characteristics of grid workflows.

[89] Windows Azure Diagnostics, Logging and Monitoring. http://archive.msdn.microsoft.com/WADiagnostics.

[90] Amazon CloudWatch. http://aws.amazon.com/cloudwatch/.

[91] S. Frischbier and I. Petrov. From active data management to event-based systems and more. chapter Aspects of data-intensive cloud computing, pages 57–77. Springer-Verlag, Berlin, Heidelberg, 2010.

[92] E. Deelman, S. Callaghan, E. Field, H. Francoeur, R. Graves, N. Gupta, V. Gupta, T.H. Jordan, C. Kesselman, P. Maechling, J. Mehringer, G. Mehta, D. Okaya, K. Vahi, and L. Zhao. Managing large-scale workflow execution from resource provisioning to provenance tracking: The CyberShake example. In *e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference on*, page 14, dec. 2006.

[93] Grid Physics Network in ATLAS. http://www.usatlas.bnl.gov/computing/grid/griphyn/.

[94] Sloan Digital Sky Survey. http://www.sdss.org/.

[95] Biomedical Informatics Research Network. http://www.birncommunity.org/.

[96] Basic Local Alignment Search Tool. http://blast.ncbi.nlm.nih.gov/.

[97] S. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 181–192, New York, NY, USA, 2010. ACM.

[98] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

[99] S. Bharathi and A. Chervenak. Data staging strategies and their impact on the execution of scientific workflows. In *Proceedings of the second international workshop on Data-aware distributed computing*, DADC '09, New York, NY, USA, 2009. ACM.

[100] S. Ko, R. Morales, and I. Gupta. New worker-centric scheduling strategies for data-intensive grid applications. In *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, MIDDLEWARE2007, pages 121–142, Berlin, Heidelberg, 2007. Springer-Verlag.

[101] Amazon EC2 Instance Types. http://aws.amazon.com/ec2/instance-types/.

[102] Google Cloud Storage. http://cloud.google.com/storage.

[103] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 401 –409, may 2007.

[104] R. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1):23–50, January 2011.

[105] G. Wang and E. Ng. The impact of virtualization on network performance of amazon EC2 data center. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 1163–1171, Piscataway, NJ, USA, 2010. IEEE Press.

[106] M. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon S3 for science grids: a viable solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, DADC '08, pages 55–64, New York, NY, USA, 2008. ACM.

[107] D. Brown, P. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb. A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 39–59. Springer London, 2007.

[108] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pages 225–, Washington, DC, USA, 2002. IEEE Computer Society.

[109] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: The Montage example. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1 –12, nov. 2008.