Performance-oriented Service Management in Clouds

by

Chao Chen

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

May 2016

Abstract

Cloud computing has provided the convenience for many IT-related and traditional industries to use feature-rich services to process complex requests. Various services are deployed in the cloud and they interact with each other to deliver the required results. How to effectively manage these services, the number of which is ever increasing, within the cloud has unavoidably become a critical issue for both tenants and service providers of the cloud. In this thesis, we develop the novel resource provision frameworks to determine resources provision for interactive services. Next, we propose the algorithms for mapping Virtual Machines (VMs) to Physical Machines (PMs) under different constraints, aiming to achieve the desired Quality-of-Services (QoS) while optimizing the provisions in both computing resources and communication bandwidth. Finally, job scheduling may become a performance bottleneck itself in such a large scale cloud. In order to address this issue, the distributed job scheduling framework has been proposed in the literature. However, such distributed job scheduling may cause resource conflict among distributed job schedulers due to the fact that individual job schedulers make their job scheduling decisions independently. In this thesis, we investigate the methods for reducing resource conflict. We apply the game theoretical methodology to capture the behaviour of the distributed schedulers in the cloud. The frameworks and methods developed in this thesis have been evaluated with a simulated workload, a large-scale workload trace and a real cloud testbed.

To all sleepless nights.

Acknowledgements

I am indebted to many people for the advice, support and friendship they have provided over the past few years. It is my great pleasure to acknowledge them, and their contribution to shaping the contents of this thesis, here.

First and foremost, I would like to express my heartfelt gratitude to my supervisor, Dr. Ligang He. From supporting a foray into research as a master working on my dissertation project, through to the very end of my time at Warwick. In these past years, despite my ignorance and recklessness, Dr. Ligang introduced me to the world of science, and taught me everything I know about how to conduct scientific research. He constantly offered fruitful guidance and strong support during all the time that I was in the Department of Computer Science at the University of Warwick. I benefited greatly from his insightful advice and continuous encouragement. I hope I can inherit some of his talent in the research and many valuable qualities as a person.

The odyssey of my PhD research would much harder without my lab mates and friends in and outside the University of Warwick. Many thanks go to: Dr.Bo Gao, Huanzhou Zhu, Zhuoer Gu, Peng Jiang, Shenyuan Ren, Xufeng Lin, Qiang Zhang, Bo Wang, Ning Jia, Dr.Xin Lu, Dr.Lei Shi, Dr.Wilson Tan, Roberto Fernandez, Dr.Jasmine Desmond, Congrui Ji and Dr.Eric Yi Liu. Many ideas of mine came from our stimulating discussions in the research and all happy memories that we share. In addition, all staff in the Department of Computer Science at the University of Warwick, in particular Dr.Phil Taylor, Dr.Steven Wright, Dr.Roger Packwood, Jackie Pinks, Dr.Christine Leigh, Dr.Adam Chester, Dr.Jane Sinclair, Dr.Mike Joy, Dr.Matthew Leeke, Dr.Arshad Jhumka, Professor.Rob Procter and Professor.Chang-Tsun Li, I sincerely thank them for constructive suggestions and technical support during my work as a teaching assistant, the four years teaching experience during my PhD was a priceless memory for me.

I would like to thank the love of my life, my best friend and my wife Yaoyao Yi, her patience, motivation and unwavering love have promoted me pass through all my difficult times in my life.

Last but not the least, I want to express my eternal gratitude to my parents, Zhendong Chen and Liyun Liu, who always give me unconditional love and support through my life, and remind me that there are other important things in life besides machines and algorithms.

Declarations

This thesis is submitted to the University of Warwick in support of the authors application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree. The work presented was carried out by the author except where acknowledged.

[25] C. Chen, L. He, B. Gao, C. Chang, K. Li, and K. Li. Modelling and optimizing bandwidth provision for interacting cloud services. In *Service-Oriented Computing*, pages 305–315. Springer, 2015

[24] C. Chen, L. He, H. Chen, J. Sun, B. Gao, and S. A. Jarvis. Developing communication-aware service placement frameworks in the cloud economy. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013

[41] B. Gao, L. He, and C. Chen. Modelling the bandwidth allocation problem in mobile service-oriented networks. In *Proceedings of the 18th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 307–311. ACM, 2015

[55] L. He, D. Zou, Z. Zhang, C. Chen, H. Jin, and S. A. Jarvis. Developing resource consolidation frameworks for moldable virtual machines in clouds. *Future Generation Computer Systems*, 2013. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2012.05.015

In addition, the following works are under review:

- GRACES: Game-theoretical Cluster Scheduler with the Awareness of Scheduling Conflict. This paper introduces game-theoretical method for distributed cluster schedulers processing their requests, and the work is presented in this thesis as Chapter 5.

- Developing Communication-aware Service Placement Frameworks in the Cloud Economy. This paper is an extension of [24, 25]. The combination frameworks used in this paper forms the basis of that used in this thesis, and the results of the experiments are presented in Section 4.3 of this thesis.

- A Taxonomy and Survey on Cloud Network Management. This paper generalizes state of the art bandwidth management in the cloud data centres from different aspects. In preparation for submission.

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- Department of Computer Science, The University of Warwick, UK: Postgraduate Research Scholarship

Abbreviations

AA	Application-specific IP address
AWS	Amazon Web Services
AS	Aggregation Switches
BL	Baseline strategy
CAGA	Communication-Aware Genetic Algorithm
CGA	Communication-oriented Genetic Algorithm
CC	Concurrent and Competitive
CIO	Communication Input-Output
CDF	Cumulative Distribution Function
EC2	Elastic Compute Cloud
IT	Information Technology
IO	Input-Output(Analysis Model)
LA	Location-specific IP Address
M-1	Medium-1 strategy
M-2	Medium-2 strategy
M-3	Medium-3 strategy
MB	10242 Bytes
MX	Max strategy
NS	Nash Equilibrium strategy
N.E	Nash Equilibrium
OS	Operating System
PS	Processor Sharing
PM	Physical Machine
QoS	Quality of Service
RC	Resources Container
RN	Random strategy

TOR	Top-of-Rack
VPC	Virtual Private Cloud
VM	Virtual Machine

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

The growing Internet enriches our social activities and modern business in many aspects, and the fundamental core supporting all those Internet-based technologies is defined as "cloud computing" [7]. This emerging technology has evolved the IT industries, because it not only provides the convenience and flexibility for individual developers or startup companies to build their products at a minimal cost, but also extends the commercial successes to the large enterprises by leasing their infrastructure and services. Over the last few years, many non-IT related industries, such as NASADQ and Lamborghini [99], have benefited from the cloud in developing their products without the need of maintaining the expensive IT infrastructures, whilst those IT enterprises which provides cloud services, such Google, Amazon and Microsoft, have profited financially from the cloud marketing at an unprecedented scale of over $20 billions [86].

Despite the rapid development of cloud computing, it rises the complexity and challenges in academia and industry, especially on resources provision, VM placement and scheduling in the cloud. Mastering these aspects with the awareness of the different potential issues becomes one of the most critical research topics in the further advance of cloud computing.

1.1 Resource Provision

In the cloud system, a number of services are often deployed in a collection of VMs. Services interact with each other and the interaction patterns may be dynamic, varying according to the system information at runtime. These impose a challenge in determining the amount of resources required to deliver a desired level of QoS for each service. Inaccurate resources provision incurs either budget waste on unnecessary resources or businesses penalty by breaking the required QoS. The research of resources provision is important as it is directly related to services deployment in the cloud.

Cloud tenants rent resource and services to build their IT platform for processing their daily businesses. For example, British Gas [99] uses Amazon Web Services Elastic Computing Cloud (AWS EC2), AWS Relation Database Service and AWS Simple Storage Service to build their data centre infrastructures to store their daily business data. AWS Auto Scaling automates the scaling of the VM resources as the requests increase. AWS Elastic Load Balancing distributes their daily traffic across VMs. Moreover, the deployed services within the cloud are hosted by a collection of VMs, which are allocated across PMs in the data centres. When the external requests arrive, the deployed services interact with each other, which essentially forms a workflow, to deliver the final results.

Another example of interacting cloud services is NASDAQ QMX, the largest stock exchange company in the world, has been developing their data analysis services on AWS [101]. The data analysis services receive and analyse the companies' financial data submitted by the tenants, and reduce the analysis results such as profit trends, investment risks and so on. The data analysis process may not be completed by a single service, but may involve a collection of interacting services, which are implemented through the standard services provided in the Amazon cloud, such as Amazon simple Storage Service (S3), Amazon Virtual Private Cluster (VPC), Amazon Elastic Compute (EC2), Amazon Direct Connect (DC) and Amazon Elastic MapReduce (EMR), etc.. The invocation relations of these services are illustrated in Figure 1.1, in which a top part of the figure shows a standard workflow of tasks that may be executed during the data analysis process, and the bottom part shows the Amazon cloud and the services deployed in the cloud. The green dash lines show which service a task in the workflow invokes. When a tenant wants to analyze the data, it first invokes the S3 service to upload the financial data to S3, which is task T_1 in the workflow. Then task T_2 invokes Amazon EC2 to start a number of Virtual

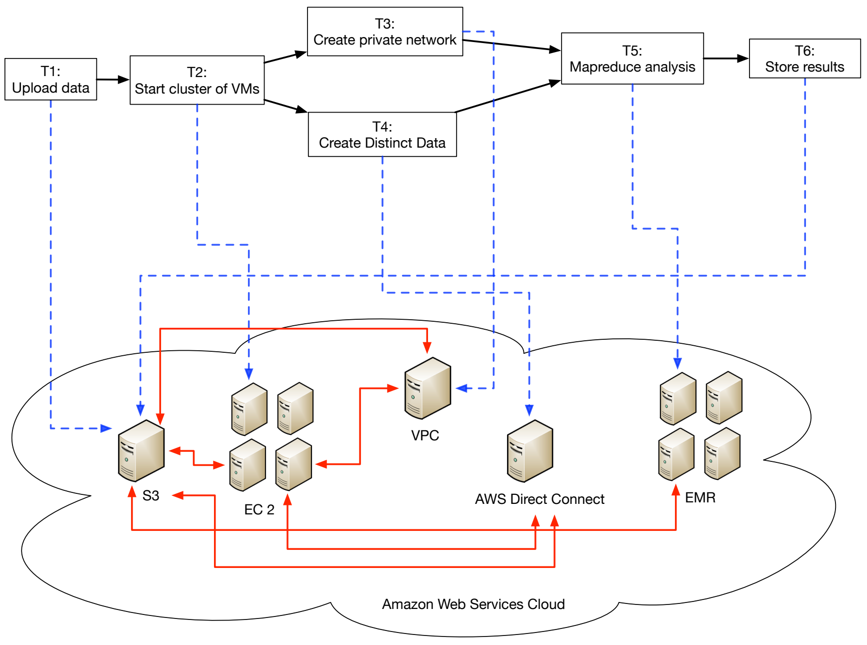


Figure 1.1: The workflow of interactive services

Machines (VMs) to support the services required by the following tasks in the workflow. After T_2 is finished, Task T_3 invokes the VPC service to create an isolated network for the VMs that support a particular service. In the meanwhile, Task T_4 calls the DC service to place the data from the tenant in a distinct storage container (so that the data are separated from other tenants' data) and implement a dedicated network connection from the tenant to the data. After T_3 and T_4 are completed, T_5 invokes the EMR service and uses the mapreduce method to generate the analysis results. Finally, T_6 invokes the S3 service again and stores the analysis results in S3.

Note that we call the above workflow a standard workflow of tasks. This is because some tasks (and services) in the workflow may not be run depending on dynamic system information/state at runtime. For example, T_1 (and service S3) may not be run if the data to be analyzed is already in S3 for the current tenant who is initiating the data analysis process. After T_2 is finished, whether T_3 is run depends on the performance and security need of the current tenant.

Also whether T_4 has to be performed may depends on whether the data contain security-sensitive information or the current tenant conflicts with other tenants in the system. Further, when the services interact with each other, data may be communicated between them. The red line in Figure 1.1 represents the communication relations amongst the services during the data analysis workflow to create the separate networks for VMs. Also the data stored by the S3 service needs to be sent to EMR service to perform the mapreduce operation. As can be seen from the above discussions, we may not always know the actual invocation workflows across multiple services in the cloud. Similar observations have also been made in [16, 70].

In addition to the above data analysis requests spawning a workflow of the service invocations, there may be other requests that are also invoking these standard services in the Amazon cloud, either through a single invocation from an external tenant (external requests) or through the invocations spawned by other services inside the cloud (internal requests). For example, a request is sent to the S3 service to upload the data for other purposes and is irrelevant to the above data analysis process. Due to the dynamic interaction relation among the services, which are demanded by the mix of the external and internal requests, it is a non-trivial task to determine the suitable amount of resources for supporting each service so as to deliver the desired QoS. This situation also exists in many other application domains when their services are deployed in the cloud, such as video transcoding [100], TV program production [27] and so on.

1.2 VM Placement

Virtualization is a core technology in cloud computing, which is the key for providing the flexibility and isolation to cloud tenants in terms of using the machine resources, such as CPU cores, memory, disk storage and bandwidth. Virtualization implements the hypervisor to split the physical resources in a single Physical Machine (PM) into multiple VMs, so that the VMs hosted in

PMs can be leased to and utilized by different tenants to deploy their services without interference. On the contrary, although multiple users and their services can co-exist in the traditional architecture, i.e., a single Operating System (OS) being installed in a PM, they are not running in isolated environments. Their architectures are illustrated in Figure 1.2.

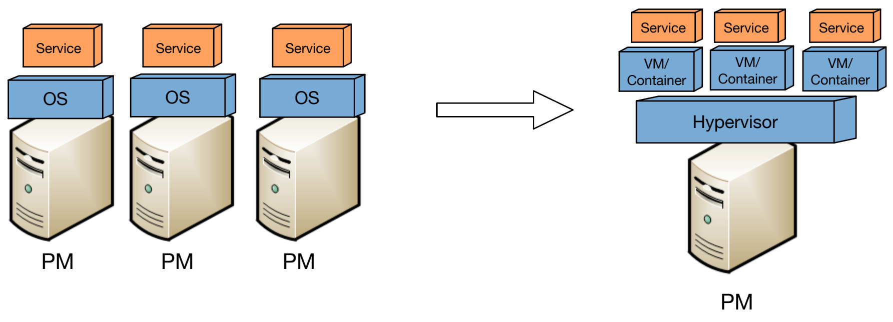


Figure 1.2: The transformation of PM usage

VMs in a PM can scale up their resource capacity and can also be migrated into another PM to achieve resource consolidation. This ability cuts the considerable cost by helping reduce the number of used PMs, and also gives the cloud tenants and cloud providers the flexibility to dynamically adjust their implementations based on various objectives. Based on this idea, different types of virtualization have been developed. For example, QUEM [15], Hyper-V [83] and KVM [72] utilize the full virtualization technique to virtualize the full hardware for each VM, which make it easy to implement the guest OS in a VM. On the other hand, Xen[11], which currently is the mainstream virtualization technique in academia, adopts para-virtualization that virtualizes all hardware resources for a main VM (domain 0), which isolates the CPUs and memory resources for each hosting VM, but shares the bandwidth resources with all VMs rather than virtualizing all hardware resource for each VM. Although para-virtualization needs to slightly modify the VM's OS to utilize the hardware virtualized by domain 0, its performance can be significantly better than full virtualization [11, 114]. Recently, the virtualization technique which is even lighter emerges. [82] adopts the process-virtualization that uses the Linux container process to isolate the predefined resources for the processes of individual users. This implicates that all users have to use the identical OS kernel, but can have separated computing and bandwidth resources from each other. Better performance can be achieved than other existing virtualization techniques. Despite the advantages and disadvantages of different virtualization techniques, they share similar principle aim, i.e., splitting the PM resources across multiple users without interference. We thus regard each isolated resources unit as the VM or the RC (Resource Container) (in chapter 5).

VM-to-PM placement is an important problem since different placements lead to different performance and resource consumption, such as the number of used PMs in total, the amount of communication traffic among services, and so on. The resource capacity in PMs, such as the total number of CPU cores, memory and storage capacity, and the communication patterns among services complicate this performance optimization problem.

1.3 Job Scheduling

Cloud infrastructure is constructed with the support of virtualization in individual PMs. Figure 1.3 shows the overview of a typical cloud infrastructure. Such a cloud infrastructure can be seen in the modern open source cloud platforms [85, 87, 96]. The cloud, actually is formed as one or a few of data centres, each of which often comprising a few thousands of PMs connected with the networking products such as switches. Normally, a central manager or scheduler handles all incoming requests demanding for resources or services from the Internet, and allocates the requests to suitable PMs and commit required VMs and other resources. The PM allocation and resource commitment of VMs can be elastically adjusted according to the administration policies and the tenant requirements. The “pay-as-you-go” model is adopted in cloud that only charges the amount of utilized resources for the length of usage time. Cloud tenants can focus on the product development while the work of maintaining and expanding the supporting IT infrastructures is considered by the cloud providers.

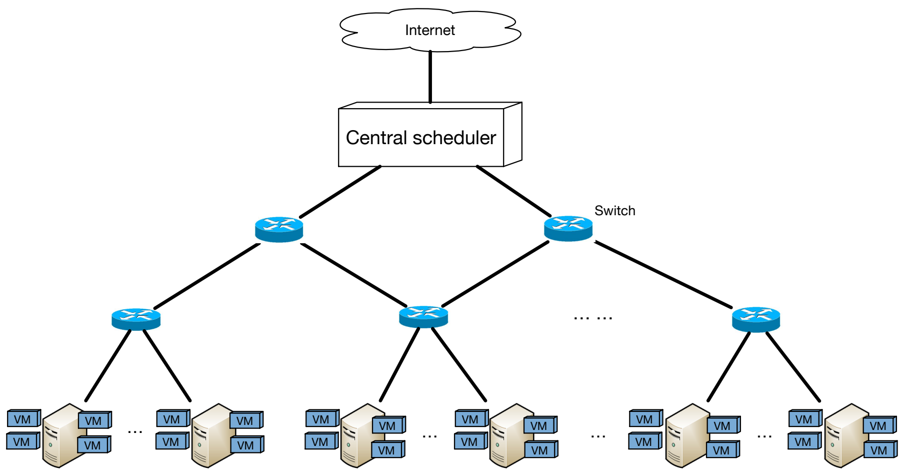


Figure 1.3: The overview of cloud architecture

However, it has exposed limitation to adopt traditional centralized scheduler to handle service requests in large-scale clouds. Distributed job scheduling frameworks have been proposed in the literature to maximize the scheduling throughput and reduce the job pending time [18, 36, 69, 95]. However, distributed schedulers may cause the resource conflicts, i.e., different schedulers attempt to schedule the jobs to the same resource so as to exceed the capacity of the resource. Therefore, in such an emerging architecture, alleviating the resource conflict plays a vital role in maintaining the desired performance.

1.4 Contributions

In order to address the above issues, the following work has been conducted in this thesis.

- We develop a framework to determine the provision of computing resources required for cloud services to deliver the desired level of QoS. The frameworks capture the interaction relations among different services in the cloud.
- We further extend the above framework to model bandwidth provision so

as to not only meet the external communication demand but also the internal communication relations. And we evaluate it with simulations at the scalability of a industrial level. A real world AWS cloud testbed with the framework has been implemented to verify its efficiency and effectiveness.

• We develop two VM-to-PM placement algorithms under the constraints on computing resources and communication cost.

• We propose the game-theoretical methodology for distributed job schedulers to exploit the trade-off between resource conflicts and resource demand. In this thesis, the parallel scheduling behaviour by distributed schedulers is modelled as a non-cooperative game and the Nash Equilibrium point is solved for the game, which represent the best scheduling behaviour of distributed schedulers.

• We conduct the experiments to evaluate the effectiveness of the proposed methods for resource provision, VM placement and distributed scheduling. The experiments are carried out with simulated workload, the workload trace from a production cloud and the a cloud testbed.

1.5 Thesis Overview

The rest of this thesis is organized as follows. **Chapter 2** reviews the modern cloud computing architecture, the existing methods of managing the computing and communication resources in cloud systems and other existing work related to this thesis.

Chapter 3 presents the novel frameworks presented in this thesis for modelling the demands for computing resources. These frameworks borrow the ideas from the Input-Output model in economy and capture the interaction relation among cloud services, which are different from the existing resource provision work that focuses on individual single services. These frameworks can be used by cloud providers or cloud tenants to model and plan their resource offering or resource purchase. This chapter further presents the VM-to-PM placement methods. These methods are communication-aware, which differentiate our work from the existing work in literature. The extensive experiments with simulated workload and real industrial workload trace have been conducted. The real experiments have also been carried out on the AWS cloud platform to verify the effectiveness of the efficiency of these methods.

The services are hosted in VM in clouds. After the provision for computing resources is determined in chapter 3. **Chapter 4** further extends the framework to model provision of communication resources.

Chapter 5 proposes a game-theoretical methodology to regulate the scheduling behaviour of distributed schedulers in cloud systems. The resource conflict due to the independent scheduling decisions made by distributed schedulers is quantitatively analysed. Experiments have been conducted to evaluate the proposed methodology.

Finally, **Chapter 6** summarises the thesis. The conclusions are drawn and some further research directions are discussed.

CHAPTER 2

Related work

As listed in chapter 1, the key aspects that define the difficulties in the cloud computing include: 1) resources allocation under dynamic services interactions; 2) bandwidth provision in different architectures; 3) VM-to-PM allocation with different objectiveness; 4) job scheduling at the emerging architecture of distributed schedulers. Therefore, in this chapter we explore and investigate some of existing techniques, to understand their methodologies and weaknesses in the cloud computing.

2.1 Resource Allocation for VMs

Various methodologies have been proposed to construct the performance model, i.e., to establish the relation between the performance of a VM (e.g., throughput, the time needed to complete a request) and the resource capability allocated to the VM (e.g., CPU, memory and storage) [3, 13, 14, 16, 23, 68, 104]. For instance, the work in [68] used layered queuing network to model the response time of a request in a multi-tiered web service hosted in VM environments, while hardware resources (e.g., CPU and disk) are modelled as processor sharing (PS) queues. In particular, the work in [104] modelled the contention of visible resources (e.g., CPU, memory, I/O) and invisible resources (e.g., shared cache, shared memory bandwidth) as well as the overheads of the VM hypervisor implementation. In the simulation experiments, we use the queuing theory as the exemplar technique to derive the performance model.

There is the work addressing resource allocation for a group of VMs with communications among them [16, 63]. For example, [16] models the interacting workflow as a Markov Chain. It detects the changes in the QoS of each service based on a sliding time window, and estimates the resource provision for services based on the QoS requirements. The work in [63] can translate the performance goals of the tasks submitted by tenants to the resource allocation in terms of the combination of the number of VMs and the network bandwidth between the VMs. Since different resource combinations may produce similar performance, the work further proposes a method to select the resource combination that can balance the resource utilization.

The work in [63] bears similarity with the work presented in this work. The difference is that the work in [63] is job-oriented (or client-oriented), i.e., to calculate the resource allocation given the specific tasks submitted by the tenants. However, as the service invocations (i.e., the tasks) may vary according to the dynamic system information, and it may be difficult to know the full picture of the tasks/workflows to be run in the cloud. The work in this thesis is service-oriented, which does not focus on allocating resources for a set of specific tasks or workflows, but aims to allocate resources based on the service interaction patterns. This work does not even have to know the full information of the tasks/workflows to be run.

2.2 Bandwidth Provision in Clouds

The network architecture of modern data centre is described as the network topology, routing/switching devices, the used protocols and commodity-class physical machines [12, 48]. Figure 2.1 shows a classic three-tier topology [56, 73] of a data centre. The network elements are divided into three layers in terms of:(1) access layer; (2) aggregation layer; (3) core layers. In the access layer, the Top-of-Rack (TOR) switch provides the basic connection to each PM mounted on to this data centre. Every Aggregation Switch (AS) in the aggregation layer distributes data traffic from those ToRs to the core layer. Each ToR switch is connected with more than one ASs for the redundancy and high availability. Moreover, the core switches in the core layer are responsible for the connection between the Internet and inside the data centre. Since the three-tiers topology is simple and easy to understand, it is the major choice for most data centres first choice.

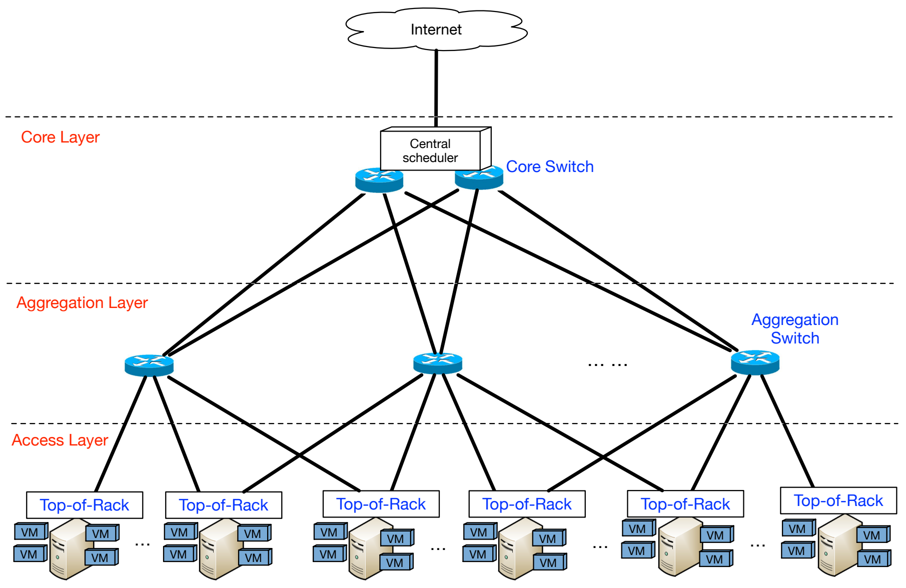


Figure 2.1: The overview of three-tier architecture

Fat-tree topology [77], as we illustrate in the Figure 2.2, is the topology aiming to maximise the end-to-end bisection bandwidth. In the fat-tree topology, although the hierarchical organization still exists in edge (access), aggregate and core layers, the number of switches is much more than three-tiers topology. Switches in ToRs and AS are formed as the the number of k "pods" that each pod consists of $(\frac{k}{2})^2$ PMs, 2 layers of $\frac{k}{2}k$ port switches. Each edge switch connects to $\frac{k}{2}$ PMs and $(\frac{k}{2})$ aggregate switches, each the aggregate switch connects $(\frac{k}{2})$ edge and $(\frac{k}{2})$ core switches, so that there are $(\frac{k}{2})^2$ core switches connect to k pod, which reduces the traffic load at the core layer. Fat-tree topology is better bandwidth connections and highly cost effectively compared to the three-tier counterpart, but the complex IP addressing scheme and multipath routing algorithm are not easy to be implemented.

VL2 [45] is the hierarchical fat-tree based topology. VL2 uses a flat automated addressing scheme that facilitates the placement of PMs anywhere in

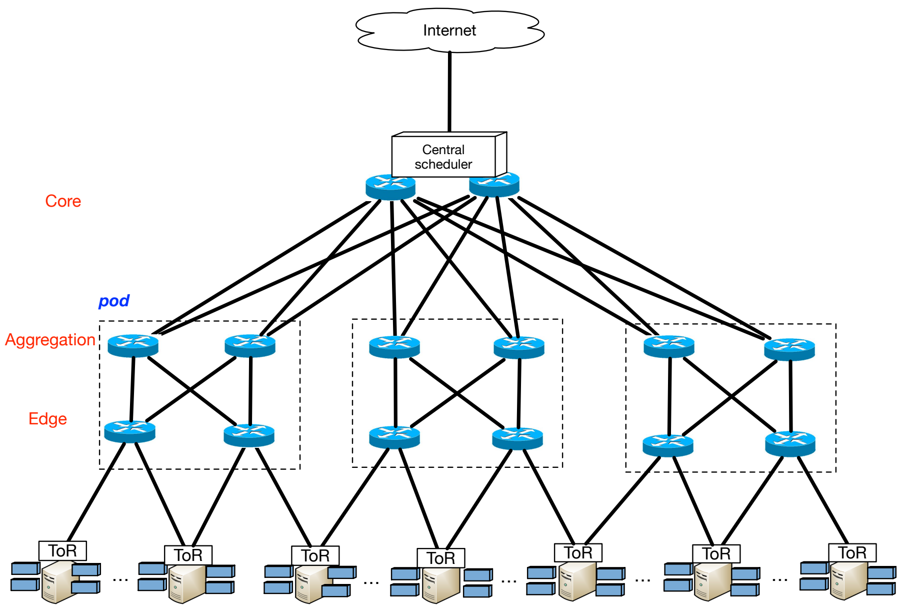


Figure 2.2: The overview of Fat-tree architecture

the data centre by two IP address families: Location-specific IP Address (LA) for the PM within the network, and Application-specific IP address (AA) for the application or VM. The LA is used to forward the packet in the physical network whilst the AA keeps unchanged regardless of movements in a PM's location within the data centre. The sending PM encapsulates the AA information in the LA packet header, but it is trapped and de-encapsulated by the ToR switch associated with the receiver. This topology has provided more flexibility to programmers within the cloud data centre to abstract and operate the network bandwidth and services communication compared to other topologies. However, the virtual overlays and centralized management, such as VL2 Directory services, are even more expensive than others to implement.

There are also many others advanced network topologies for abstracting and managing the data centre, such as [28, 48, 49, 52, 102]. Regardless of various network topologies, the cloud network is virtualized and manager by the central scheduler to assign different services. The work in [9, 10, 91, 92] implements the techniques to enforce the defined bandwidth allocation for each VM that is used to host specific services. Those techniques are principally based on the Hose model [38], which have the analogies with practical switch networks that offers the abstraction to each VM for one service has a dedicated link of guaranteed minimum bandwidth capacity to a non-blocking virtual switch as in Figure 2.3. However, these studies do not consider the policies to determine the appropriate bandwidth capacity for each service and its constituent VMs from a holistic perspective. The work in [9] and [10] designed the centralized controller to assign the bandwidth in order to achieve the required performance. On the other hand, the work in [91, 92] distribute the ability of allocating the bandwidth and develops sophisticated policies in the hypervisor and the switch to compete the bandwidth for ensuring guaranteed communication. Although the above studies developed the elaborate techniques to manage shared networks and enforce the bandwidth allocation, they do not consider the policies to determine the appropriate bandwidth capacity for each service and its constituent VM from a holistic perspective.

The work in [108, 115] adopts the stochastic model to analyse the bandwidth requirements for each service. [111] guides the bandwidth provision dynamically. [51, 67] propose the routing policies within the data centres to exploit the potential bandwidth provision. Their works are solely based on "external" demands, while our work not only satisfies the external demands, but also captures the internal demands due to service interactions.

2.3 VMs-to-PMs Placement

The VM-to-PM placement problem mainly aims to consolidate resources and improve resource utilization [20, 46, 57, 59, 112]. Various methods have been proposed in literature to address the VM-to-PM placement problem, including knapsack modelling [57], mixed integer programming [22, 90, 103], genetic algorithms [55, 103, 112], ant colony optimization [39, 42] and heuristic methods [59, 79, 106]. For example, the work in [59] develops the heuristic squeeze

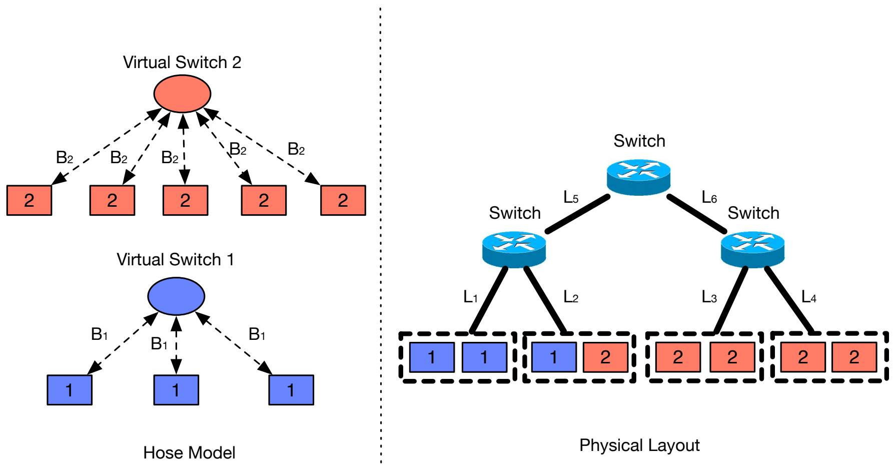


Figure 2.3: An example of the hose model, which guarantees the bandwidth demands of two services (blue for service 1 and red for service 2) with the mapping of their virtual networks to a physical network.

and release measures to dynamically redistribute the workloads in the cluster according to the workload level on each individual node, so as to minimize the usage of physical machines. The work in [57] develops a server consolidation scheme, called Entropy. Entropy strives to find the minimal number of nodes that can host a set of VMs, given the physical configurations of the system and resource requirements of the VMs. The objective is formalized as a multiple knapsack problem, which is then solved using a dynamic programming approach. In the implementation, a one-minute time window is set for the knapsack problem solver to find the solution. The solution obtained at the end of the one-minute time space is the new state (i.e., the new VM-to-PM placement). The work in [55] designed a genetic algorithm to find a VM-to-PM placement that uses the minimal number of PMs. However, the above work is used to tackle the placement of independent VMs (i.e., there are no communications among VMs), aiming to minimize the usage of physical machines. In this thesis, we investigate the placement of the interacting VMs and focus on finding the VM placement that can minimize the communication cost.

There is also some works tackling the placement problem for the VMs with inter-VM communications, aiming to minimize the communication costs [50, 81]. The work in [50] models such a VM-to-PM placement as a min-cost network flow problem and then uses the Breadth First Search to find the optimal placement solution. The work in [81] uses the classical min-cut graph algorithm to obtain the optimized placement solution. In order to model the placement problem as the min-cost network flow problem or the min-cut graph problem, they need to know the specific communication pattern between each pair of VMs. As we have discussed, in some cases, we may not know the full picture of the submitted workload, and therefore can not accurately determine the communication pattern between each individual VM. In this thesis, we model the interactions among the services, and treat a service (and the set of VMs supporting the service) as a whole without the need to know the specific communication pattern between each pair of VMs.

2.4 Analysis of Invocation Pattern among Services in Cloud

There are a number of existing techniques to obtain the invocation patterns of the services [8, 26, 110]. The work in [8] implements a multi-level probabilistic model to infer the probability of a service calling another service. The fundamental idea is to monitor the packets sent and received by a service, and then compute the dependency probability between the services by leveraging the observation that if accessing service B depends on service A, then packets exchanged with A and B are likely to co-occur. The work in [26] then uses the k-means clustering technique in data mining area to analyze the service trace and calculate the correlation probability between services.

2.5 Job Scheduling in Clouds

Data centres nowadays have to process a large scale of jobs on a daily basis. On one hand, the resource demand, along with the commercial success of cloud computing, becomes the major driving force for the cloud providers to increase the size of their data centres. On the other hand, in order to handle the jobs efficiently, cloud giants, such as Google, Microsoft and Amazon, have developed various cluster management frameworks in their production clusters [107]. Among them, one conventional approach is to develop a centralized scheduler in the cluster, which manages diverse types of job submitted to the cluster. However, because of the large number of the jobs and the complexity of making scheduling decisions for some types of job, the centralized schedulers become the performance bottleneck for delivering resources and processing jobs timely. A recent trend thus is to deploy multiple, independent schedulers in a cluster. Different schedulers make scheduling decisions simultaneously for different types of job, aiming to improve the throughput and cluster utilization. These independently working schedulers in a cluster are termed *distributed schedulers* in the literature [18, 88, 95].

The scheduling procedure in the cloud with distributed schedulers is illustrated in Figure 2.4. When a request is submitted, it is compiled as a workflow with resource requirements. The workflow is modelled by a directed acyclic graph (DAG). In the graph, nodes represent jobs such as map, reduce or join jobs, and edges represent dataflow. Each job consists of one or more identical parallel tasks. Those jobs are allocated to individual pending queues corresponding to their dependent services based on the job type, resource preference or job size. Additionally, a distributed scheduler asynchronously scans [1] the jobs in its pending queue, and finds its best matching machines and claim the resources in these machines to launch the tasks. Different schedulers choose the machines based on their preferences in various aspects. In summary, the role of

[1]The order of scanning can be implemented using different policies. Here we set it as FIFO (First-In-First-Out).

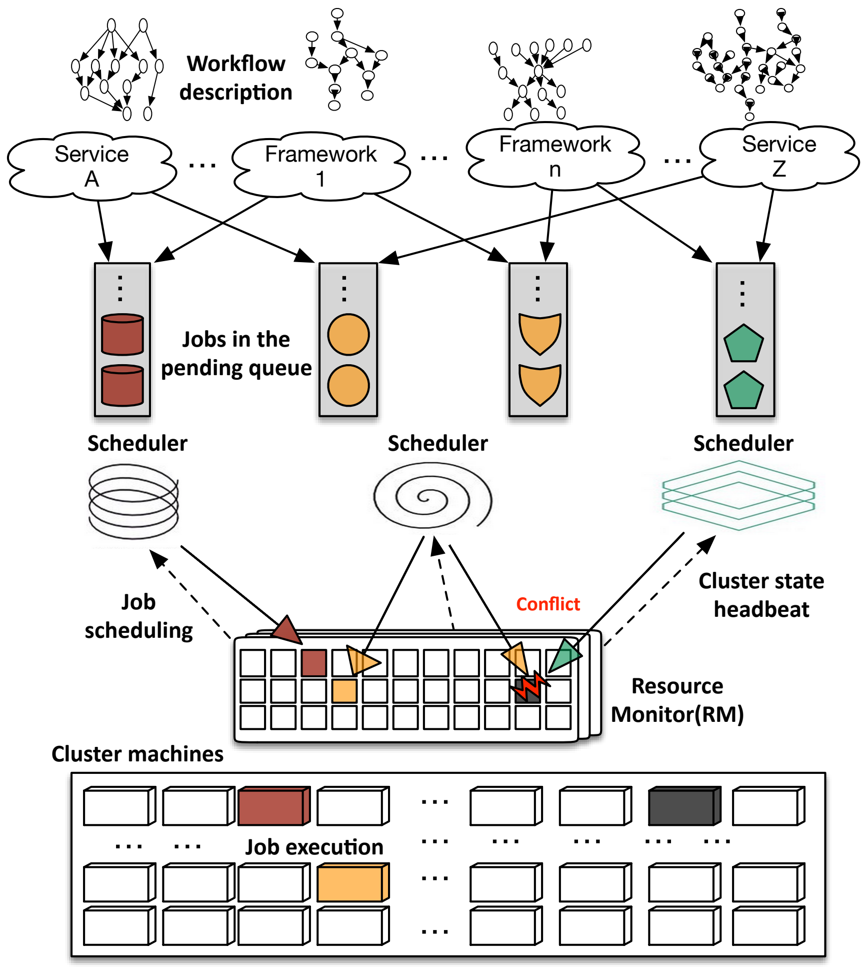


Figure 2.4: The overview of a shared cloud

a scheduler covers two parts: *Scheduling* and *Servicing*.

In *Scheduling*, the scheduler scores resources (machines) to decide the best ones for its jobs. The resulting scores could be very different for different schedulers. For example, the scheduler for Hadoop[113] or Spark[116] jobs chooses its target machines based on the machine load and data locality, whilst the scheduler for scheduling web server applications needs to perform some complex optimization algorithms to consider the heterogeneity and performance interference. In addition, there is a separate Resource Monitor (RM) that periodically collects from all machines the reports about their resource load and machine state, and then aggregates the reports as the global cluster state for all schedulers. Each scheduler thus keeps a copy of global cluster state to make its own scheduling decision.

In *Servicing*, the scheduler claims the resources in the target machines to initialize, start or cancel its jobs, such as fetch needed data and install the required software packages. The time cost in initialization is not negligible in most productive clusters [18, 107]. The median initialization cost is around 25 seconds [107]. Therefore, unlike the monolithic scheduler, such as Mesos [58] and Fuxi [118], distributed schedulers do not have a central coordinator and their scheduling attempts and job initialization might conflict with each other. Once the conflict occurs, only one of the scheduling attempts is granted while others are deemed failed and consequently these schedulers have to reschedule the jobs in question.

We investigate the scheduling events in the production clusters at Google and Microsoft, which have developed their distributed cluster schedulers, Omega and Apollo, respectively. However, there exists a significant ratio of rescheduling in their daily scheduling events. Especially in Google, the ratio of task reschedule is up to 64% of total scheduling events [94].

Distributed schedulers for the large-scale cloud

The emerging trend of distributed schedulers firstly starts with Omega [95]. Since then, many similar products have been developed [18, 33, 36, 69]. Although these schedulers are aware of the scheduling conflicts, their focuses are on resolving the conflicts after they occur rather than trying to reduce the chance of conflict in the first place. In this thesis, we balance the size of requested resources in this shared cloud environment by carefully analyzing the scheduling cost, performance behaviour of the jobs and the level of competition among distributed schedulers.

Game-theoretical scheduling

Game-theoretical scheduling is a classic scheduling problem in the cluster management [19, 31, 37, 43, 44, 71, 75, 89]. These approaches are based either on the central scheduler architectures to achieve system-level load balance [89] or resource allocation [19] between jobs and cluster machines, or on sidestepping the non-cooperative game and making each participating computer to cooperate together and achieve the global optimal performance [47, 75]. These approaches cannot be applied in our scenario as distributed schedulers are selfish and strive to maximize their own scheduling performance.

We thus model the scheduling scenario as a non-cooperative game and develop a theoretically proven solution to reduce the conflict possibility.

Performance metrics in Clouds

It is not straightforward to determine the suitable performance metrics to evaluate the scheduling performance in Clouds since a variety of performance issues may be examined. Recent research has focused on reducing the makespan specifically on the off-line jobs [33, 69], allocation interferences [35] and the quality of performance [34] for on-line jobs. Their solutions primarily target a particular type of jobs. The work [62, 117] proposes the performance concepts of Application Normalized Performance (ANP) and System Normalized Performance (SNP). The performance cost adopted in our work shares the similarity with these two metrics. But the difference is that based on them, we tailor the utility function for distributed schedulers so that that it can be applied on both on-line and off-line jobs.

2.6 Background of the IO model

In the IO model, the economy is divided into sectors. Each sector produces goods except for the open sector, which only consumes goods. When a sector produces goods, it needs to consume the goods produced by other sectors. The consumption matrix C captures the consumption relations among sectors. An element c_{ij} in C represents the amount of goods produced by sector i that have to be consumed by sector j in order for sector j to produce one unit (e.g., in terms of US dollars) of goods. The consumption matrix C represents the internal demands. Assume the column vector D contains the goods demand from the open sector, which represents the external demand. The element d_i in D represents the amount of goods from sector i required by the open sector. Let the column vector X be the equilibrium levels of production output that can satisfy both internal and external demands in the economy. The element x_i in X represents the equilibrium level of output by sector i. X must satisfy Eq. 2.1, which may be solved for X by transforming it to Eq. 2.2.

$$X = C \cdot X + D \tag{2.1}$$

$$X = (I - C)^{-1} \cdot D \tag{2.2}$$

2.7 Summary

This chapter has reviewed other related techniques adopted in the cloud computing. We discussed difficulties and shortages of current approaches, especially in a dynamic, complex and large scale of cloud environment. In particular, the computing and bandwidth resource provision (Section 2.1 and Section 2.2) are advanced in chapter 3 and chapter 4, respectively, which we implement in a large scale of cloud services and a real cloud testbed with dynamic and complex interactions. Moreover, two sophisticated VM-to-PM placement algorithms have been designed and implemented in these two chapters, two of them are designed for reducing the communication cost amongst services under different constraints. In Section 2.5, we proposed a game-theoretical mechanism for the emerging new architecture of distributed schedulers in chapter 5.

CHAPTER 3
Computing Provision for Cloud Services

Numerous cloud services are deployed in the cloud. These services are often not isolated. After a cloud tenant invokes a service, the service may request further actions from other services during or after its execution. The interactions among the services are not static and may vary according to dynamic system information at runtime. To determine the amount of resources required for each of these services that becomes critical in order to deliver a desired level of QoS.

The services in a cloud system are typically hosted in VMs. Therefore, determining the suitable resource quantity for the services comes down to determine the resource capacities allocated to the VMs that host the services. There are some existing works building the performance model for the processing capacity of a VM. For instance, establishing the relation between a VM's processing capability and the amount of the resource capacities (such as CPU percentage, memory size, network bandwidth) allocated to the VM [68, 104]. For example, Amazon EC2 offers small, medium, large and extra large VMs [97]. The performance model established in the existing work can calculate the processing capability of these different types of VMs for a type of tasks.

The work in this chapter makes use of the performance model of a VM established in the literature. Assuming that the processing capability of one VM is known, this chapter presents a method to determine the sufficient number of VMs for interacting services in a cloud system. The proposed method borrows the ideas from the Leontief Input-Output Model in economy [78] (called the IO model in this chapter). The IO model conducts the input-output analysis for different industry sectors in an economy. It is able to capture the consumption relations among different sectors and calculate the equilibrium level of production for each sector, so as to satisfy both external demands from the open sector (e.g., people) and internal demands due to the consumptions relations among individual sectors in an economy. Moreover, the IO model is able to analyze the impact of the increase in the external demand for a particular sector on the production of all sectors in the economy.

The behaviours of the interacting services in a cloud system bear the similarity with the behaviours of different industry sectors in an economy. A service supplies resources, which are consumed by tenants and also by other services due to service interactions. To the best of our knowledge, this chapter is the first one in literature that applies the IO model in economy to formalize and solve the resource demand problem in clouds.

Further, when the services interact with each other, data may be communicated between them. If the VMs that host the services with frequent communications among themselves can be placed to the same PM, the communication cost could be significantly reduced.

There is the existing work in literature investigating the VM-to-PM placement problem [39, 50, 55, 57, 81]. However, the existing work either focuses on consolidating the independent VMs (i.e., there are no interaction between VMs) into resources, i.e., finding a VM-to-PM placement that can minimize the number of PMs used to host the VMs, or requires to know the specific communication patterns between individual VMs. Different methods have been developed to model such a VM-to-PM placement problem. For example, the placement problem has also been modelled as a knapsack problem [57], an ant colony optimization problem [39], a mixed integer programming problem [90] and a genetic algorithm [55]. Then the existing solvers and the bespoke methods were developed to solve the objective functions for the optimized placement solutions. Heuristic algorithms have also been developed to find the placement solutions [59].

This chapter develops a communication-aware strategy to place the VMs that host the interacting services on physical machines, aiming to minimize the communication costs incurred by the service interactions. A genetic algorithm is then developed to find a VM-to-PM placement with significantly reduced communication costs. Briefly, the main differences between our work and the work in literature are that 1) this work aims to find a placement solution to minimize the communication costs among services and 2) the approach adopted in this work does not need to know the specific communication pattern between individual VMs.

We have also conducted experiments to compare the framework proposed in this chapter with two existing placement methods: one striving to use the minimal number of PMs to host VMs, and the other applying the heuristic approach to placing VMs. Our experimental results show that the proposed communication-aware framework significantly outperforms the heuristic approach in terms of both communication cost and the number of used PMs, and that comparing with the method aiming to achieve the minimal number of used PMs, our communication-aware approach is able to significantly reduce the communication overhead in the cloud with only a tiny fraction of increase in resource usage.

The remainder of this chapter is organized as follows. Section 3.1 presents the workload and system models. Section 3.2 proposes the method of modelling resource demands of services in a cloud economy. The communication-aware VM placement framework is presented in Section 3.3. Section 3.4 and Section 3.5 evaluate the effectiveness of the proposed framework. Finally, we concluded in Section 3.6.

3.1 Workload and System Models

Let $\mathbb{S} = \{s_1, \ldots, s_M\}$ denote a set of M services deployed in a cloud. λ_i denotes the arrival rate of the requests directly from the tenants for service s_i. p_{ij} denotes the probability that after service s_i is invoked and executed, service s_i will further call service s_j. A service is hosted in a set of VMs (i.e., a virtual cluster). Assume each VM that hosts the same service (i.e., each VM in a virtual cluster) is allocated with the same resource capacity (e.g, the proportion of CPU, memory size, etc..). This assumption is reasonable because this is the normal practice when using a virtual cluster to host a service [9]. VM^i denotes a VM that hosts service s_i. There may be multiple VMs on a PM. We assume that PMs and network links are homogeneous, i.e, the PMs and the network links connecting any two PMs in the cloud has the same performance. This assumption is reasonable since homogeneous machines and communication networks are typically used to construct a cloud system.

Given the arrival rate of the requests for service s_i and given VM^i's resource capacity, there are a number of existing techniques in literature [68, 104] to calculate the adequate number of VM^is that can satisfy the desired QoS in terms of a particular performance metric (e.g., average waiting time of the requests, throughput).

Table. 3.1 lists the notations used in the chapter.

Table 3.1: Notations for VM provision

notations	Explanation
C	consumption matrix
c_{ij}	the amount of goods produced by sector i that have to be consumed by sector j in order for sector j to produce one unit of goods
S	the number of services
s_i	service i
e_{ij}	the amount of data that are sent when service s_i invokes s_j
p_{ij}	the probability that one invocation of s_i causes the invocation of s_j
n_i	physical machine i
VM^i	a virtual machine hosting service s_i
v_{ik}	the number of VM^is in n_i
p_{ji}	the possibility that executing s_j causes a further invocation of s_i

3.2 Modelling Resource Demands of Cloud Services

This section applies the IO model to formalize and calculate the equilibrium level of resource capacity demanded by the external tenants and the interacting services in a cloud economy. The constructed model is called the cloud-IO model in this chapter. In order to apply the IO model to formalize a cloud economy, we have to use the entities in the IO model (i.e., sector and goods) to represent the entities in cloud environments, such as service, request, VM, resource, etc..

In this chapter, a service in the cloud economy is regarded as a sector in the IO model while the external tenants are regarded as the open sector, which is straightforward. However, the challenge is to identify the entity in the cloud economy that is suitable to be regarded as goods, and also determine the consumption relations among services. We first attempted a straightforward option and use the requests sent by the tenants or the services to represent goods. This option seems to be intuitive, because a service processes (consumes) requests from tenants and other services, and also generates (produces) requests to invoke other services. Then the problem comes down to how to determine the resource capacity for services so that the requests can be processed in a way that the desired QoS can be met. However, we later realize that it is not appropriate to treat the requests as goods. This is because the requests generated by services are not going to be consumed by the tenants while the goods produced in the IO model are consumed by the open sector. In this chapter, a group of VMs hosting a service are regarded as goods produced by the service.

Now we present how to determine the consumption relations among services, i.e, obtain the consumption matrix. Note that c_{ij} in the consumption matrix C represent the amount of goods produced by sector i that have to be consumed by sector j in order for sector j to produce one unit of goods (e.g., in terms of US dollars). Consider one VM (a unit of good) of service s_j. ψ_j denotes the arrival rate of the requests that one VM of service s_j (i.e., one VM^j) can handle to deliver the specified QoS. As discussed in Section 3.1, there are existing techniques to calculate ψ_j, given the resource capacity allocated to the VM. We use a function f to represent such a technique, i.e., Eq. 3.1, where the first parameter represents service index (i.e., s_j), the second parameter R_j represents the resource capacity allocated to each VM of s_j (we assume every VM in the same service has the same resource capacity), and the third parameter represent the number of VMs of the service.

$$\psi_j = f(j, R_j, 1) \tag{3.1}$$

Every time service s_j is invoked, there is the possibility of p_{ji} that s_j will send a request to further invoke s_i. Therefore, in a time unit one VM^j sends $\psi_j \times p_{ji}$ requests to s_i. The number of VMs that need to be produced by s_i to handle the requests with the arrival rate of $\psi_j \times p_{ji}$ is then equivalent to the goods produced by service s_i that have to be consumed by service s_j in order for s_j to produce one unit of goods (i.e., one VM), which is actually c_{ij} in the IO model. Again, the existing techniques in literature can calculate c_{ij} based on the arrival rate of $\psi_j \times p_{ji}$ and the given resource capacity allocated to each VM^i. We use a function g to represent such a technique, i.e., Eq. 3.2, where the first and second parameters have the same meanings as those in Eq. 3.1, and the third parameter represents the arrival rate of the requests.

$$c_{ij} = g(i, R_i, \psi_j \times p_{ji}) \tag{3.2}$$

In doing so, we have established the consumption matrix in the cloud-IO model. Let λ_i be the rate at which the tenants (open sector) send the requests to service s_i. Then we can use the g function in Eq. 3.2 to calculate the number of VM^i that have to be produced by s_i to process the requests with the arrival rate of λ_i, which is d_i in the column vector D in the IO model. Namely, d_i can be obtained using Eq. 3.3.

$$d_i = g(i, R, \lambda_i) \tag{3.3}$$

By doing so, the external demand vector D is obtained, $X = [x_1, ..., x_i, ..., x_M]^T$ denotes the column vector that represents the number of VMs required for each of M services in the cloud economy. X can be calculated by Eq. 2.2.

3.3 The Communication-aware VM Placement

Section 3.2 calculates the number of VMs required for each service in the cloud. This section investigate the issues of mapping all the VMs obtained in Section 3.2 to PMs. The VM-to-PM mapping in literature often focuses on minimizing the number of PMs used to accommodate the VMs, so as to minimize the resource and/or energy consumption. However, in this chapter, there is the possibility that after a service is run, it may send a request to another service for further actions. Some data may be sent along with the request. If the VMs that host the different services with frequent communications can be placed in the same PM, then the communication cost could be reduced. This section develops a framework to find the VM-to-PM mapping that minimizes the communication cost in the cloud.

According to the cloud-IO model, c_{ij} represents the number of VMs that need to be produced by s_i to handle the requests send by one VM in s_j. Therefore, if the ratio of the number of VM^is to the number of VM^j in PM n_k, denoted as α_{ijk}, is no less than to c_{ij}, then the requests (along with the data) sent by the VM^js can be handled by the VM^is in the same PM without breaching the QoS of s_i, and therefore eliminates the necessity to send the requests and data to the VM^i in a different PM. On the contrary, if α_{ijk} is less than c_{ij}, then a proportion of the requests sent by the VM^js in n_k have to be processed by VM^is in a different PM. The greater the difference between c_{ij} and α_{ijk} is, a larger proportion of the requests and data sent by VM^js in n_k have to be sent out of n_k and therefore a higher communication cost in the cloud. The communication-aware service placement framework developed in this chapter is based on this insight and aims to find a VM-to-PM mapping with the minimal communication cost in the cloud.

3.3.1 Formalizing the Problem

This section models the total communication cost incurred by an arbitrary VM-to-PM mapping in the cloud. As discussed above, when α_{ijk} is less than c_{ij}, the communication will occur between n_k and another PM where there are VM^i. v_{ik} denotes the number of VM^is in n_k, given a VM-to-PM mapping $\mathcal{M}$. The communication cost incurred by the mapping $\mathcal{M}$, denoted as $\mathcal{C}(\mathcal{M})$, can be calculated by Eq. 3.4 and Eq. 3.5. In Eq. 3.5, the term $(f(j, R_j, v_{jk}) \times p_{ji} - f(i, R_i, v_{ik}))$ calculates that the amount of requests that are sent from s_j in PM n_k to s_i in a time unit, but cannot be handled by VM^is in n_k (if $\alpha_{ijk} < c_{ij}$) in order to maintain the QoS. Therefore, these requests have to be sent to be processed by VM^is in a different PM. The number of these requests times e_ji is then the total amount of data that have to be communicated in the cloud caused by the inadequate resource capacity of s_i in PM n_k comparing with that of s_j in the same PM. Since we assume that the communication network in the cloud is homogeneous, we do not have to consider which PM these data will be sent to. The communication cost is then the sum of all these data that have to be sent out of the local PM by any service in the cloud, which is Eq. 3.4.

$$\mathcal{C}(\mathcal{M}) = \sum_{k=1}^{N} \sum_{j=1}^{M} \sum_{i=1}^{M} \beta_{ijk} \tag{3.4}$$

$$\beta_{ijk} = \begin{cases} e_{ji} \times (f(j, R_j, v_{jk}) \times p_{ji} - f(i, R_i, v_{ik})) \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{if } \alpha_{ijk} < c_{ij} \\ 0 \qquad\qquad\qquad\qquad\qquad\qquad \texttt{otherwise} \end{cases} \tag{3.5}$$

The objective is to find a VM-to-PM mapping such that $\mathcal{C}(\mathcal{M})$ is minimized, subject to certain constraints. This can be formalized as Eq. 3.6, where x_i is the number of VM^is obtained in Section 3.2.

$$\begin{aligned} \texttt{miminize} \quad & \mathcal{C}(\mathcal{M}), \\ \texttt{subject to:} \quad & \forall i : 1 \leq i \leq M, \sum_{k=1}^{N} v_{ik} = x_i \\ & v_{ik} \geq 0 \end{aligned} \tag{3.6}$$

3.3.2 Designing the Genetic Algorithm

A Genetic Algorithm, called CAGA (Communication-Aware Genetic Algorithm), is developed in this chapter. CAGA tries to find the optimal mapping with the least communication cost. In a typical Genetic Algorithm (GA), a solution is encoded and then the crossover and mutation operations are applied to evolve the solutions. Moreover, a fitness function is used to judge the quality of the solutions and guide their evolution direction so that better solutions can be gradually generated over generations. In the GA developed in this chapter, the communication cost defined in Eq. 3.4 is used as the fitness function. This section mainly presents the encoding of the solution, the crossover and the mutation operations designed in our GA.

Encoding the Solution and Fitness Function

In CAGA, a solution is a VM-to-PM mapping. It is encoded as an one-dimensional array, denoted as A. An element a_i in A holds the index of a VM. B_r denotes the capacity of the r-th type of resources in a PM. Given an encoded solution, the PM that a VM is mapped to is determined in the following way. Starting from the first element in the solution, the VMs are placed into PM_1 in the order of their positions in A, until the total capacity of the VMs starts to exceed the capacity of PM_1. The VMs are then placed into the next PM. Formally, if the first k PMs have been fully occupied and the VM in a_i (i.e., VM_{a_i}) is the first VM that cannot be placed into PM_k any more, the VMs in the positions from a_i to a_{j-1} should be placed into PM_{k+1}. j can be determined using Eq. 3.7, in which $b_r(a_u)$ is the capacity of the r-th type of resource allocated to the VM with the index of a_u. For each of R types of resource in consideration, Eq. 3.7 obtain the least j_r such that the total capacity of that resource of the VMs from a_i to a_{j_r} begins to exceed B_r. Then j is the minimum number among j_r $(1 \leq r \leq R)$. The procedure repeats until all VMs have been placed into PMs. By doing so, CAGA knows which PM a VM is placed into.

$$\begin{gathered} j = min\{j_r | 1 \leq r \leq R\} \\ \texttt{subject to:} \sum_{u=i}^{j_r} b_r(a_u) > B_r \end{gathered} \tag{3.7}$$

In the encoding, CAGA starts to place a VM to a new PM only when the current PM does not have enough remaining capacity to host the VM. Therefore, the method used by CAGA to encode and calculate the VM-to-PM mapping will not generate excessive spare capacity in PMs, and therefore reduce the number of PMs used to host VMs. Indeed, our experiments show that the number of PMs used by CAGA is very close to that obtained by the VM-to-PM mapping method aiming to use the minimal number of PMs to host VMs.

CAGA aims to find a VM-to-PM mapping with minimal communication cost. Therefore Eq. 3.4 that calculates the communication cost of a mapping is used as the fitness function of a solution.

Selecting Solutions

In GA, the solutions need to be selected from the current generation of solutions to perform the crossover and the mutation operations. CAGA applies the tournament method [84] to select the solutions used to generate next generation of solutions. The tournament method is as follows. Assume there are h solutions in one generation. Each time, CAGA randomly selects k solution (k is called tournament size) from the current generation. Then CAGA takes the one with the lowest communication cost among these k solutions and uses it as one parent solution in the crossover operation. The same way is used to obtain the other parent solution. Then the crossover operation, which is presented in subsection 3.3.2.C, is performed over the two parent solutions to generate two child solutions. The procedure repeats until there are h solutions in the next generation.

Crossover and Mutation

The two-point crossover is used in CAGA. In the crossover, two points are randomly selected for two parent solutions to divide each parent into three portions. All VMs in the middle portion are swapped between the parent solutions. The resulting two solutions are children solutions in the new generation. But such a swap may cause repetitive VMs in a child solution, i.e., there may be two VMs with the same index in one solution. In order to eliminate such repetitive VMs, the swapping action is performed in the following way in CAGA. At position i in the middle portion of both parents, $a1_i$ and $a2_i$ are the indexes of VMs in parent 1 and parent 2, respectively. In parent 1, the crossover operation finds the VM with the index of $a2_i$ and swap $a1_i$ and $a2_i$. In parent 2, similarly, the crossover operation finds the VM with the index of $a1_i$ and swap $a2_i$ and $a1_i$. Such swapping is performed at every position in the middle portion of two parents. By doing so, we effectively swap the middle portions between parents, and the resulting children solutions will not have the repetitive VMs.

After crossover, the mutation operation is performed on the two newly generated child solutions. A mutation probability δ is set. For each VM in a child solution, there is the probability of δ that the VM will swap the positions with another randomly selected VM in the child solution. The mutated child solutions become the solutions in the new generation.

3.3.3 Designing the CGA

Since the CAGA uses VM index to encode the VM-to-PM placement, its convergence is limited by the number of VMs. We redesign CAGA as Communication-oriented Genetic Algorithm(CGA) based on the services-oriented encoding. The selecting solution and the fitness function are still the same as CAGA, and we have redesigned the encoding, crossover and mutation operations that are presented in the following two sections.

Encoding the Solution

In CGA, a solution is encoded as an one-dimensional array, denoted as A, An element a_i in A holds a pair of values (s_i, h_i), where s_i is the service index and h_i is the number of VMs hosting service s_i in the pair. To some extend, an element in the one-dimensional array is like an element in the two-dimensional array in CSA, but does not contain the information about which PM the VMs in an element are located in, and it has significantly saved the searching space than the CAGA.

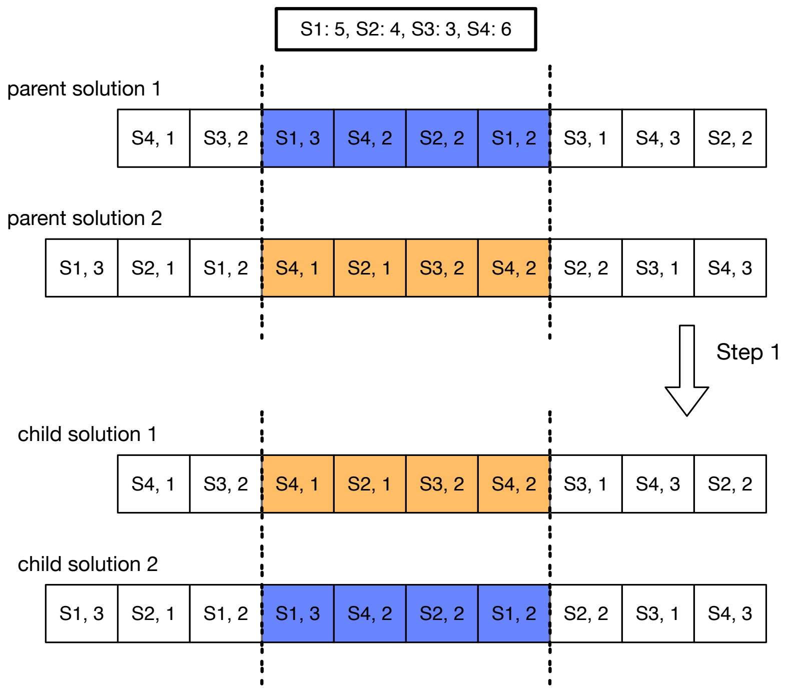
Given a encoded solution, we start to decide VMs are mapped to the PM from the first element in the solution, VMs in the element (i.e., (s_i, h_i)) are placed into PM_1 in the order of their position in A, until the total capacity of the VMs starts to exceed of PM_1. The VMs are then placed into the next PM.

Crossover and Mutation

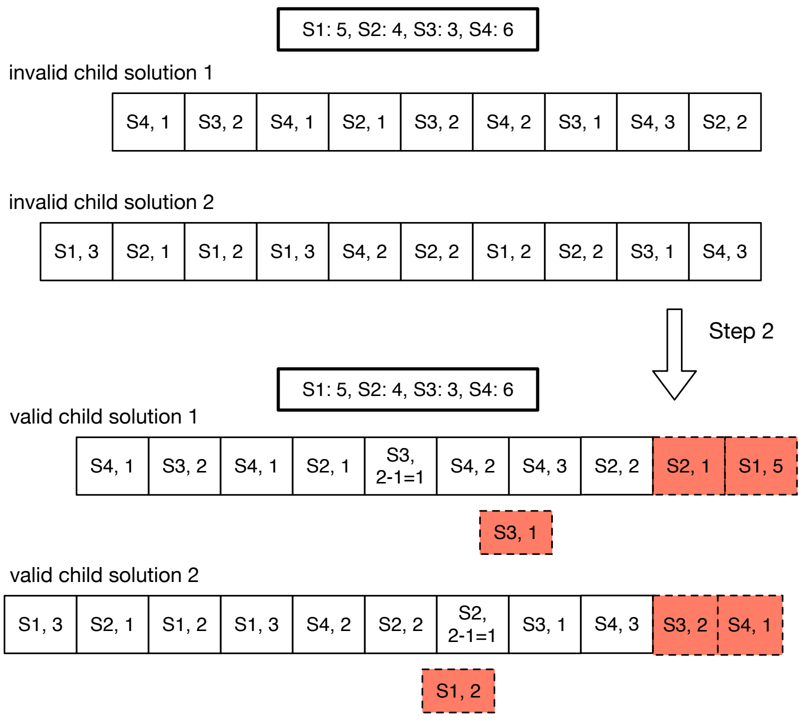
The original two-point crossover operation will generate repetitive or missed VMs for services to the original VM-to-PM placement as we discussed in section 3.3 at chapter 3. The CAGA adopts moving and swapping the selected VM indexes on the original parents solutions to complete the crossover and mutation, so that it does not actually move or swap the parent solution, which avoids the duplicated or missed VMs for services. This approach cannot apply here because the selected element on one of parent solutions cannot guarantee it also can be found in the another parent solution as the way of our encoding. For example, one of parent solution has $(S_2, 3), (S_2, 1)$, whilst another parent solution has $(S_2, 2), (S_2, 2)$, and if the crossover is involved to S_2, it will be impossible to find the swapped element in the original solution. Therefore, we use the following method to adjust the invalid child solution back to valid solution.

An example is given in Figure 3.1 to show the workings of the crossover operation. In Figure 3.1(a), the two-point crossover is performed. The top of the figure gives the number of each service calculated by the cloud-IO model. After the crossover, two child solutions are generated. However, the two child solutions are invalid. Child solution 1 contains 5 VMs of s_3, which is more than the calculated number of VMs (3) for s_3, 3 VMs of s_2, which is less than the calculated number 4, and 0 VM of s_1, which is less than the calculated number 5. For similar reasons, child solution 2 is also invalid. Figure 3.1(b) shows how to modify invalid child solutions into valid ones. In invalid child solution 1, since s_2 and s_1 have the numbers of VMs less than the specified numbers by 1 and 5, respectively, we append two elements to the invalid child solution 1, $(s_2, 1)$ and $(s_1, 5)$ to make the solution valid in terms of services s_2 and s_1. Further, since the number of VMs for s_3 is more than the specified number by 2 in the invalid child solution 1, we start from the end of the solution and remove the VMs of the element that contains s_3 until 2 VMs of s_3 have been removed. The first element containing s_3 is $(s_3, 1)$. Since all VMs in the element have been removed, the element is deleted. The next element containing s_3 is $(s_3, 2)$. We only need to remove 1 VM from the element. The similar modifications can be performed to make invalid child solution 2 to become valid.

The mutation operation will be performed on two newly generated valid child solution after the crossover operation is finished. A mutation probability δ is set as CAGA, which is usually below 0.2. Then, for each child solution, we randomly select $y \times \delta$ elements, where y is the number of elements in the child solution. For each selected element, the mutation operation further randomly selects another element and swaps these two elements. Because the mutation does not need to swap with another solution, the duplicated or missed VMs issues on the crossover is not existed, the validation procedure would not be necessary in the mutation operation.


S1: 5, S2: 4, S3: 3, S4: 6
parent solution 1
S4, 1 | S3, 2 | S1, 3 | S4, 2 | S2, 2 | S1, 2 | S3, 1 | S4, 3 | S2, 2
parent solution 2
S1, 3 | S2, 1 | S1, 2 | S4, 1 | S2, 1 | S3, 2 | S4, 2 | S2, 2 | S3, 1 | S4, 3
Step 1
child solution 1
S4, 1 | S3, 2 | S4, 1 | S2, 1 | S3, 2 | S4, 2 | S3, 1 | S4, 3 | S2, 2
child solution 2
S1, 3 | S2, 1 | S1, 2 | S1, 3 | S4, 2 | S2, 2 | S1, 2 | S2, 2 | S3, 1 | S4, 3

(a)


S1: 5, S2: 4, S3: 3, S4: 6
invalid child solution 1
S4, 1 | S3, 2 | S4, 1 | S2, 1 | S3, 2 | S4, 2 | S3, 1 | S4, 3 | S2, 2
invalid child solution 2
S1, 3 | S2, 1 | S1, 2 | S1, 3 | S4, 2 | S2, 2 | S1, 2 | S2, 2 | S3, 1 | S4, 3
Step 2
S1: 5, S2: 4, S3: 3, S4: 6
valid child solution 1
S4, 1 | S3, 2 | S4, 1 | S2, 1 | S3, 2-1=1 | S4, 2 | S4, 3 | S2, 2 | S2, 1 | S1, 5
S3, 1
valid child solution 2
S1, 3 | S2, 1 | S1, 2 | S1, 3 | S4, 2 | S2, 2 | S2, 2-1=1 | S3, 1 | S4, 3 | S3, 2 | S4, 1
S1, 2

(b)

Figure 3.1: CGA Crossover

3.4 Performance Evaluation on the Synthetic Workflow

We have conducted simulation experiments with the synthetic workflow to evaluate the performance of the proposed communication-aware framework. A pool of S cloud services are assumed in a cloud. In the simulation experiments of this work, the workflows are generated to simulate the interactions among services. In real systems, we typically do not know the entire invocation workflows across multiple services in the cloud. In this case, the service interaction patterns, i.e., p_{ji} in Table 3.1, can be obtained by analyzing the invocation trace of each individual service in the cloud, or analyze the source code of a service and its execution flow.

With the information of the generated workflows, p_{ji} can be calculated as follows. A workflow has h nodes with the random topology. A node in a workflow represents the invocation of a service randomly selected from the service pool. Therefore, a service may appear multiple times in a workflow. A link from service (node) s_i to s_j represents that after s_i is run, s_i sends a request to further invoke s_j. The weight of a link represents the amount of data that needs to be sent from s_i to s_j when s_i invokes s_j. A workflow has a entry service (the first service that has to been invoked in the workflow). External requests arrive to invoke the entry service, which is regarded as the external demand. The arrival rate of the external requests to workflow w_i is denoted as λ_i. The invocations among services inside the workflow is regarded as internal demand. With the topology of w_i and λ_i, we can easily calculate the following variables for w_i: 1) the rate at which s_j is invoked (denoted as $\lambda_i(s_j)$); 2) the rate at which s_j invokes s_k (denoted as $\lambda_i(s_j, s_k)$); 3) the the amount of data sent from s_j to s_k in a time unit (denoted as $e_i(s_j, s_k)$). In w_i, the probability of s_j invoking s_k (denoted as $p_i(s_j, s_k)$) can be calculated as $\frac{\lambda_i(s_j,s_k)}{\lambda_i(s_j)}$. If the number of different workflows generated in the simulation is W, then the probability of s_j invoking s_k (i.e., p_{jk} in Table 3.1) can be calculated as Eq. 3.8. The total amount of data sent from s_j to s_k (denoted as E_{jk}) in a time unit can be calculated as Eq. 3.9, while the total arrival rate of the requests to s_j, denoted as λ^j, can be computed using Eq. 3.10. In the experiments, three types of workflows are generated in the experiments: communication-intensive, computation-intensive and general workflow. In the communication-intensive, computation-intensive, and general workflow, e_{ij} is randomly obtained from the range of [*min_comme*, *max_comme*], [*min_compe*, *max_compe*] and [*min_gene, max_gene*], respectively. The computation time of a node in all workflows is randomly selected from the range of [*min_comp*, *max_comp*] with the average value of *avg_comp*.

$$p_{jk} = \sum_{i=1}^{W}(\frac{\lambda_i}{\sum_{i=1}^{W}\lambda_i} \times p_i(s_k, s_j)) \tag{3.8}$$

$$E_{jk} = \sum_{i=1}^{W}(\lambda_i \times e_i(s_k, s_j)) \tag{3.9}$$

$$\lambda^j = \sum_{i=1}^{W}(\lambda_i \times \lambda_i(s_j)) \tag{3.10}$$

There are the existing techniques [104] to obtain the function f in Eq. 3.1. The value of the function f is the processing rate of a VM. In the experiments, we apply the queuing theory [68] to obtain the g function. Assume that the external requests arrive following the Poisson process, and the computation time of a service and the communication time of sending data between services follow the exponential process. According to the queuing theory, the average response time of service si, denoted as T_i, can be calculated by Eq. 3.11, where $|s_i|$ is the number of VMs that is used to host s_i, μ_i is the mean process rate of a VM hosting s_i (which is the inverse of mean computation time of an invocation in the VM of s_i and is actually the value of the f function) and P_n is the probability that the number of requests being processed in the virtual cluster is no less than n. Assume the QoS of service s_i is that the average response time of an invocation of the service is no more than q_i. q_i is normally set as

Table 3.2: Experimental parameters

Parameters	Value
S	40
B	50
[*min_comme*, *max_comme*]	[20, 30]
[*min_gene, max_gene*]	[10, 20]
avg_comp	15
h (the number of tasks in a workflow)	40
W	3
[*b_min*], [*b_max*]	[5, 15]
[*min_compe*, *max_compe*]	[2, 8]
[*min_comp*, *max_comp*]	[10, 20]
slack	20%
δ (mutation probability)	0.2

avg_comp $\times$ (1 + *slack*). Given λ^i and q_i, we can calculate from Eq. 3.11 the minimum $|s_i|$ that satisfies the QoS, which is the g function in Eq. 3.2 and Eq. 3.3. p_jk has been calculated in Eq. 3.8. Therefore, c_kj in the consumption matrix can be calculated using Eq. 3.2. With the arrival rate of the external requests, we can apply the queueing theory to calculate the number of VMs required to serve the external requests, which is D in Eq. 2.1. Finally, the number of VMs allocated to each service can be calculated using Eq. 2.2.

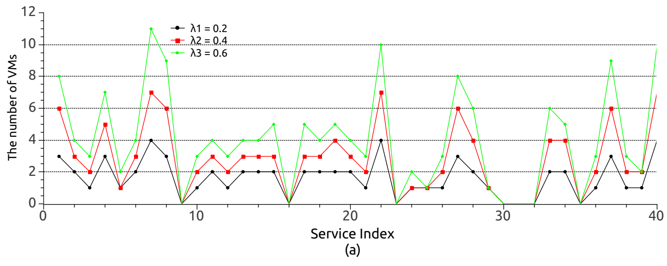
$$T_i = \frac{1}{\mu_i} + P_n \frac{1}{|s_i| \times \mu_i - \lambda^i} \tag{3.11}$$

The capacity of a physical machine is set to be B. The resource capacity allocated to a VM in s_i is set to be b_i, which is randomly selected from the range of [*b_min, b_max*]. Unless otherwise stated, the value of the experimental parameters are set as in Table. 3.2 as we adopted from the work in [55].

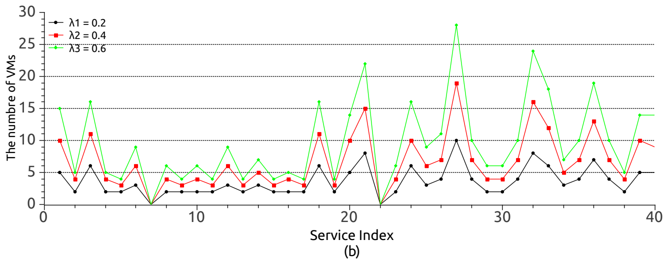
The existing work on placing VMs to PMs mainly focuses on achieving the minimum number of PMs used to host the VMs [57, 59] (which is called the *Min-nodes* algorithm in this chapter), assuming that the VMs are independent with each other. The CAGA framework developed in this chapter takes the service (VM) interactions into account. The Min-nodes method presented in [57] models the VM-to-PM placement as the bin-packing problem and then uses the existing solver to solve the problem for the VM-to-PM placement that minimizes the usage of PMs. In the experiments, we compared CAGA with the Min-nodes algorithm in terms of communication cost and the number of used PMs. Moreover, we compared CAGA with a heuristic VM-to-PM placement algorithm. In the heuristic, the VMs from different services are placed in a PM in a round-robin fashion [54]. Starting from s_0,the heuristic algorithm places a VM in s_i to the PM, then places a VM in $s_{(i+1)\%S}$ to the PM, until the PM cannot accommodate more VMs. Then the VMs are placed to a new PM in the same fashion, except for starting from the VM that cannot be placed to the previous PM.

3.4.1 Impact of the Increase in External Demands

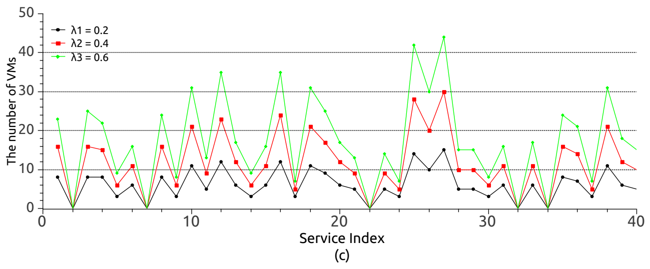
The experiments presented in this subsection investigates the impact of service interactions on resource capacity allocated to each service. Figure 3.2(a, b and c) show the number of VMs allocated to each service under different arrival rates of external requests for communication-intensive, computation-intensive and general workflows, respectively. Figure 3.3 shows the the number of VMs allocated to each service for the three workflows combined. The number of VMs is obtained using the cloud-IO model. As can be seen from Figure 3.2(a, b and c), when the arrival rate of external requests increases, not only the number of VMs allocated to the entry service of the workflow increases (s_1 in the figures), but that allocated to other services in the workflow also increases. The level of increment in some services is even much greater than that in the entry service. With the cloud-IO model, we can quantitatively obtain the impact of the increase in external demands on the resource requirements on each service in the cloud. For example, in Figure 3.2(b), when the arrival rate of the external requests increase from 0.2 to 0.6, it imposes the biggest resource burden on service s_{27}, whose VM quantity increases from 10 to 28.



(a) Computation



(b) General



(c) Communication

Figure 3.2: Impact of the increase in external demands; a)computation-intensive workflow; b) general workflow; c) communication-intensive workflow.

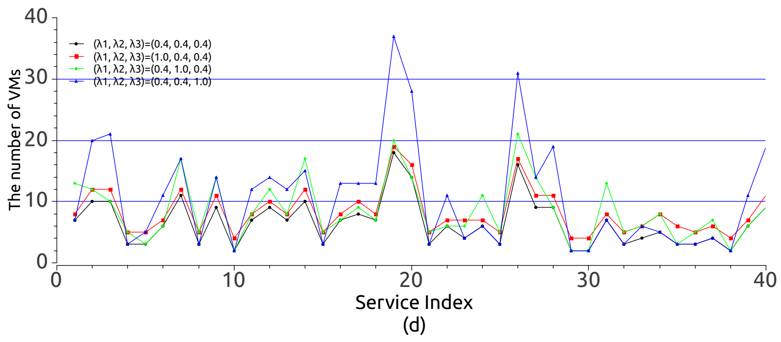
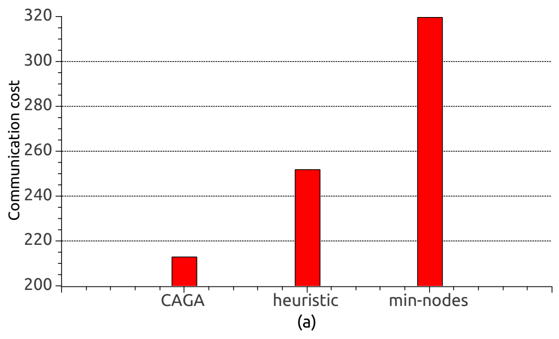


Figure 3.3: Impact of the increase in external demands by three types of workflow combined

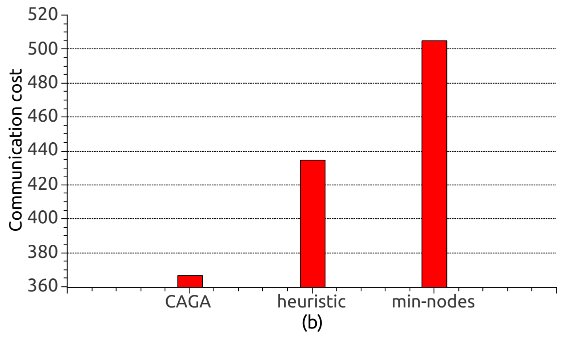
3.4.2 Comparing CAGA with the Existing Placement Methods

This subsection compares CAGA with two existing VM-to-PM placement methods: Min-nodes [57] and the round-robin heuristic [54]. Figure 3.4(a, b and c) present the results for computation-intensive, general and communication-intensive workflows, respectively. It can be seen from these figures that in all cases, CAGA significantly reduces the communication cost compared with other two methods, which suggests the effectiveness of the proposed framework.

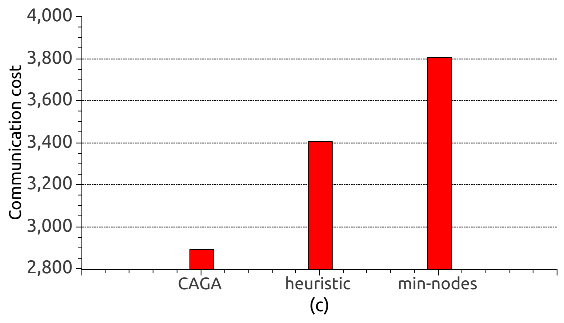
Figure 3.5 compares CAGA with Min-nodes and the round-robin heuristic under different types of workflow in terms of the number of PMs used to host the VMs. It can be seen that although Min-nodes can achieve the least number of PMs, CAGA only uses one more PMs than Min-nodes in all cases. As it has been shown in Figure 3.5, CAGA can significantly reduce the communication cost. These results indicate that CAGA is able to greatly reduce communication overhead in the cloud with only a tiny fraction of increase in resource usage. This is because CAGA takes the communication cost into account when designing



(a) Computation



(b) General



(c) Communication

Figure 3.4: Comparing CAGA with Min-nodes and the round-robin heuristic in terms of communication cost; a)computation-intensive workflow; b) general workflow; c) communication-intensive workflow.

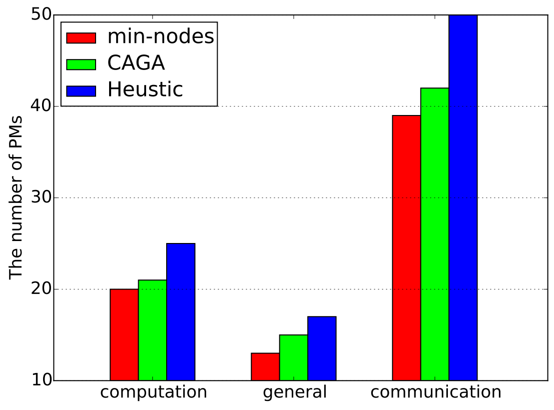


Figure 3.5: Comparing CAGA with Min-nodes and the round-robin heuristic in terms of the number of used PMs

the framework. Moreover the way used by CAGA to encode and calculate the VM-to-PM mapping ensures that there will not be the excessive spare capacity in PMs, and therefore effectively reduces the number of PMs used to host VMs.

3.4.3 Convergence of CAGA

Figure 3.6(a, b and c) show the convergence of the CAGA algorithm over time under computation-intensive, general and communication-intensive workflows, respectively. In theory, one major factor that influence the convergence speed is the number of VMs to be placed into the PMs. This is because the size of the encoded solution equals to the number of VMs to placed. The size of the solution in turn determines the complexity of the crossover and mutation operation. Another major influential factor is the number of services in the cloud, because when calculating communication cost, CAGA needs to calculate α_{ijk} for each pair of services. More services, more calculations are involved. The number of services in the experiments are 40 and the number of VMs to be placed is about 150 VMs. It can be seen from Figure 3.6 that the CAGA can reach the stable result for about 60 seconds in all three cases, and the longest time (65 seconds) is spent by the communication-intensive workflows in which the number of VMs to be placed is 167. The results suggest that CAGA can find a VM-to-PM placement with low communication cost fairly efficiently.

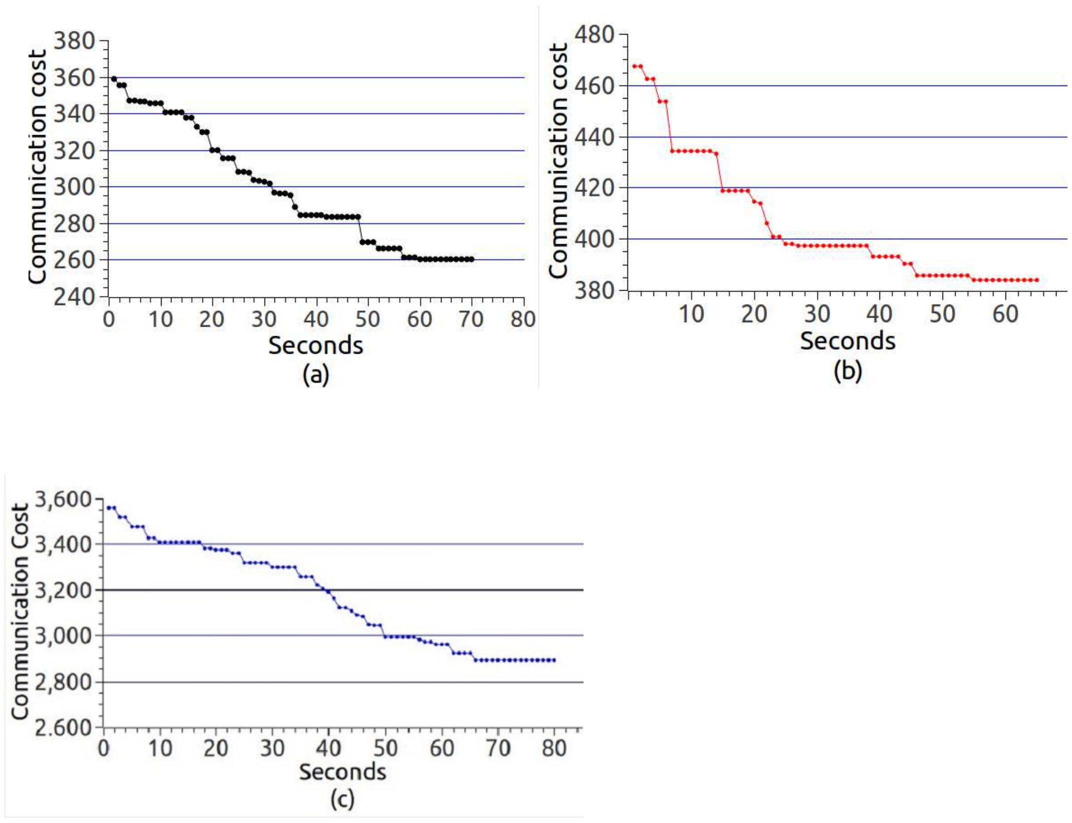


Figure 3.6: Convergence speed of CAGA; a) computation-intensive workflow, b) general workflow, c) communication-intensive workflow

3.5 Performance Evaluation on the Bing Workflow

This section we further evaluate our cloud-IO model, CAGA and CGA algorithm on the workflow from the Microsoft Bing's datacenter trace (called Bing workflow for short) [17, 40].

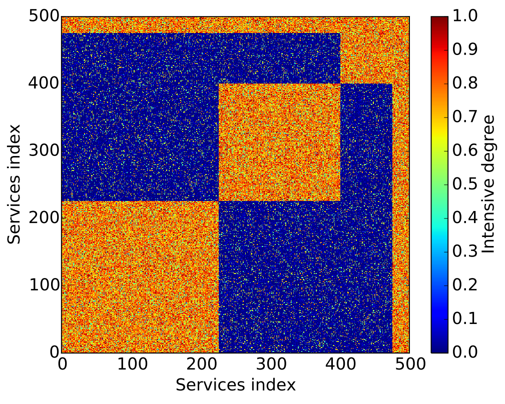
The Bing workflow is described as follows. The work in [40] describes seven types of recurring productive jobs in the Microsoft business datacenter for processing daily commercial activities for both internal developers and external customers (e.g., business partners or tenants), which has the similar characteristics as the tenant requests. In the experiments, hence, we simulate a stream of tenant requests (workflows) using these seven typical recurring job structure.

Moreover, reference [17] surveyed the communication pattern among the services in Microsoft Bing's datacenter. We use it to generate the communication pattern (i.e., p_{ij}) among the services in our simulations. In the experiments, 500 services are generated. In the Bing workflow, the communication pattern can be illustrated using the heat maps shown in Figure 3.7. In Figure 3.7, both x- and y-axis represent the list of deployed services in the datacenter and the colours of the cells represent the communication intensity in the scale of 0 to 1 between the corresponding pairs of services. In the sparse pattern of Figure 3.7 (Figure 3.7a), the deployed services can be divided into service groups according to the communication pattern. A service mainly communicates with other services in the same group. The discrepancy between inter-group and intra-group communications is reduced in the median and the intensive pattern.

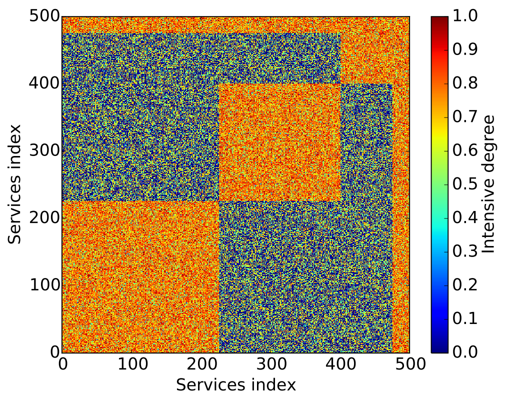
About 5% of services (the services with biggest indices in Figure 3.7) always have intensive communications with all other services, regardless of the interaction pattern. These services are monitor or scheduler services in the datacenter, which have to frequently communicate with other services. The probability that Service i invokes Service j (i.e., p_{ij} in the proposed bandwidth IO model) is set as the communication intensity between Service i and j divided by the sum of the communication intensity between Service i and any other service.

3.5.1 Impact of the Increase in External Demands

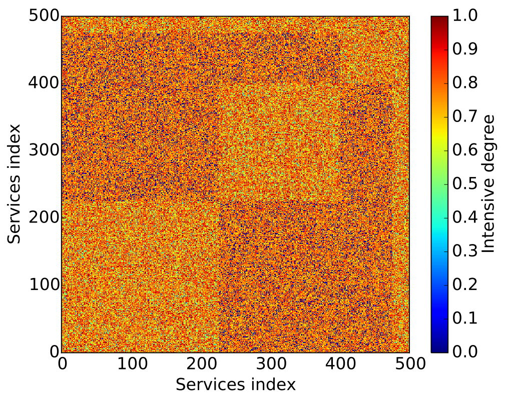
The experiment presented in this subsection investigates the impact of service interactions on resource capacity allocation with 500 services. The workflow pattern in the experiments is based on the Bing trace with the increasing arrival rate of workflows, and we set it as four type of workflows, which are sparse,



(a) Sparse



(b) Median



(c) Intensive

Figure 3.7: Communication patterns among services using on the Bing workflow

median, intensive, and mixed one, where the mixed one we have mixed those three workflows equally together. Note that for the number of VMs for 500 services we are using *box plots* (e.g., see Figure 3.8, 3.9, 3.10 and 3.11), in which the lower part of the main box represents 25-percentile number of VMs for 500 services have, the upper part indicates the 75-percentile, and the red line is the median. The lower whisker is the 5-percentile, the upper one represents the 95-percentile, and the green bullet is the mean number of VMs for 500 services have. As can be seen from those 4 figures, when the arrival rate of external requests increases, the number of VMs for all services has increased. Especially for the intensive workflow, it remarkably increase the number of VMs for all services from the total number of VMs as 5138 to 57610 with the increasing arrival rate. Moreover, it is also noticeable that the mixed workflow is closed to median workflow in terms of VMs amount for services, which is expectable as it has mixed with all workflows with equal probabilities and the sparse workflow "neutralize" the intensive workflow as like median workflows, in terms of VMs amount for services. Therefore, we can see the IO-model can quantitatively obtain the impact of increasing demands on VMs for all services with the larger scale.

3.5.2 Comparing CAGA with the Existing Placement Methods

Figure 3.12 shows that the comparison of CAGA on bandwidth provision with the existing placement methods for the above four types of workflows at arrival rate as 0.5. It indicates the CAGA has achieved the reduction in bandwidth provision around 38% and 15% comparing with the min-nodes and Heuistic placement, respectively, in all workflows, which indicates CAGA can achieve the minimum communication cost in different workflows.

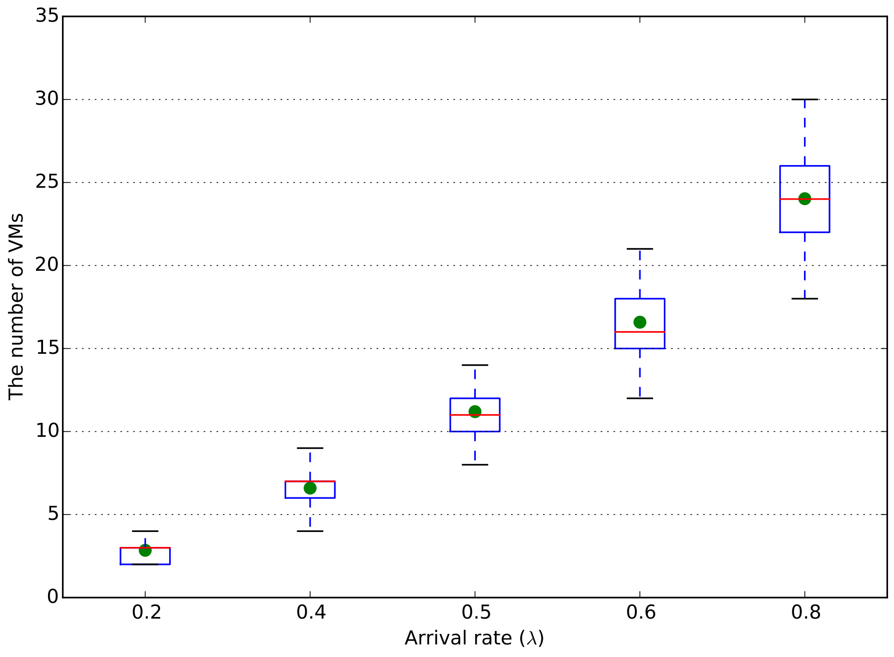


Figure 3.8: Spare workflow

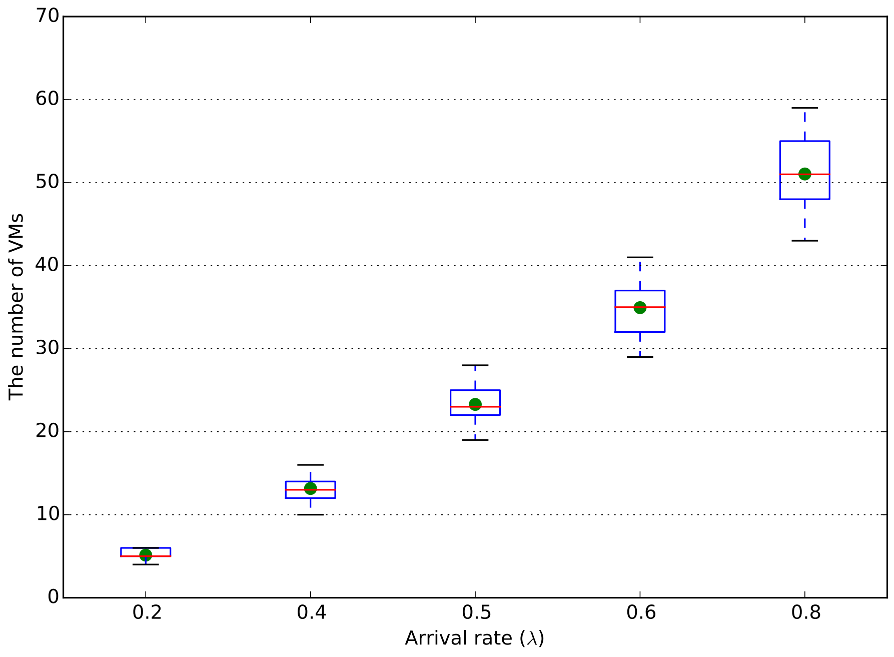


Figure 3.9: Median workflow

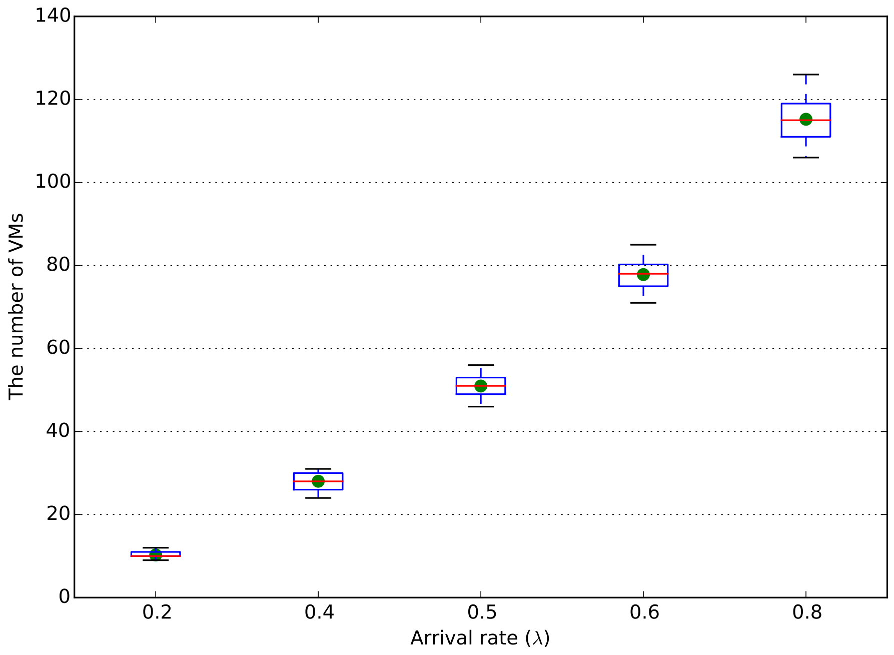


Figure 3.10: Intensive workflow

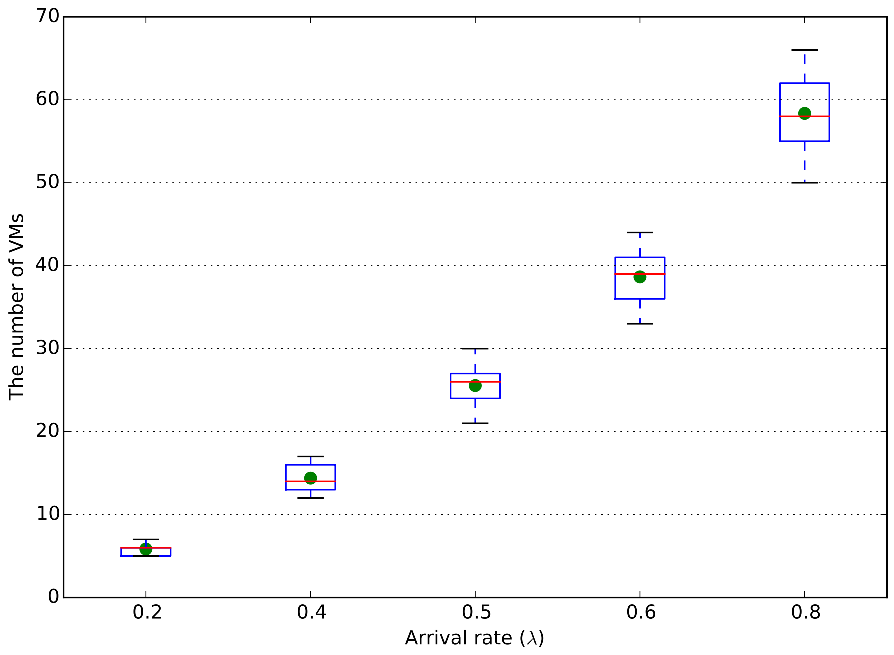


Figure 3.11: Mixed workflow

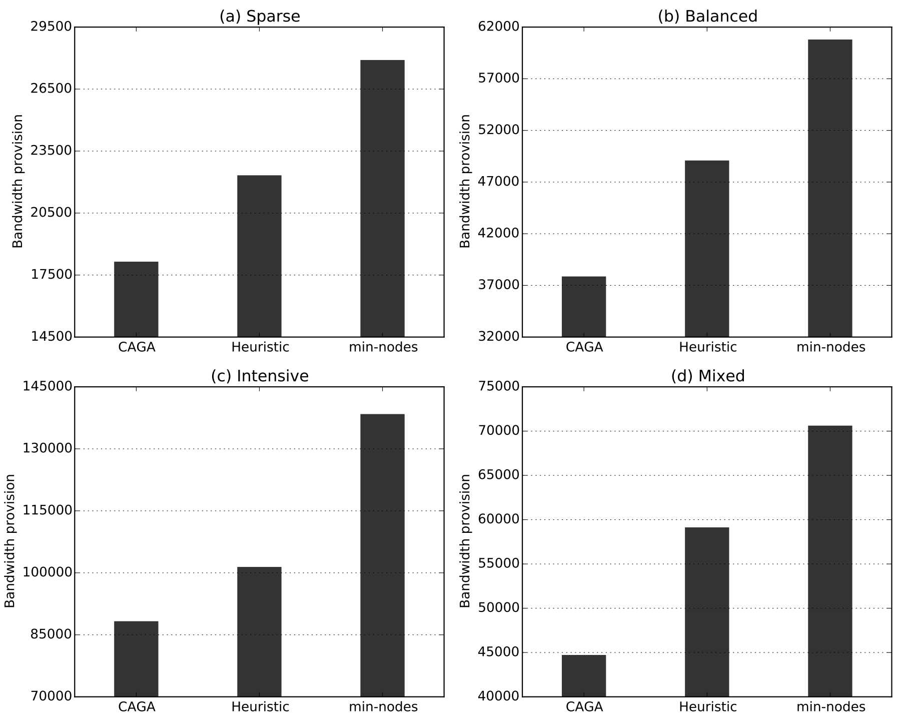


Figure 3.12: Comparing CAGA with different placement methods

3.5.3 Convergence of CAGA

Figure 3.13 presents the convergence of CAGA with the scalability of 500 services by the above four different workflows at arrival rate 0.5. With the increase of number of VMs, we can see the time cost is increasing with the VMs amount, significantly. All of workflows have taken over 10 minutes to be convergent. CAGA performs poorly at the larger scale as it encodes the VM-to-PM solution based on the VM index.

3.5.4 Convergence of CGA

Figure 3.14(a, b, c and d) show the convergence of the CGA algorithm over time under sparse, median, intensive and mixed workflows, respectively. Since we have changed the encoded solution of CGA, the convergence speeds at different scales have been improved significantly compared to CAGA. It can be seen from Figure 3.14 that CGA can reach the stable result for about 100 seconds

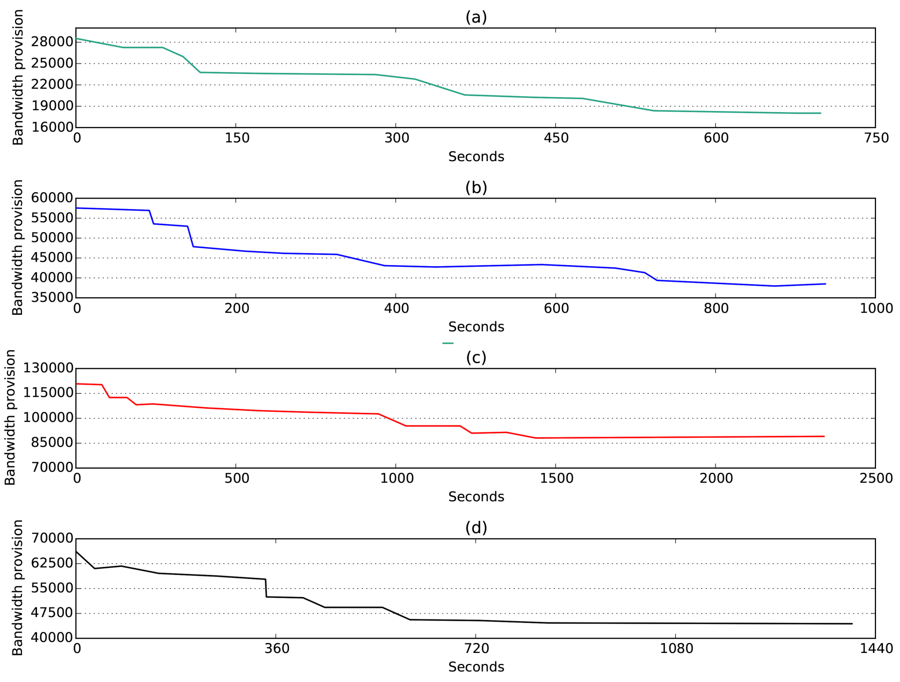


Figure 3.13: Convergence speed of CAGA; a) sparse workflow, b) median workflow, c) intensive workflow d) mixed workflow

for all workflows except the intensive one(Figure 3.14(c)), because the intensive workflow generates more than 30% VMs than others workflows, and it is the longest time is spent to convergence the stable result.

3.6 Summary

This chapter employed the input-output model in economy to model the computing resource demand for interacting services in a cloud. Based on the modelling, this chapter further developed two communication-aware VM-to-PM placement frameworks, one is based on the VM index encoding, and the other one is based on the service index encoding. Both frameworks take into account the interaction costs among services, and aim to find a VM-to-PM placement so that the communication overhead can be minimized. Two frameworks design the genetic algorithm to search for a placement that can optimize com-

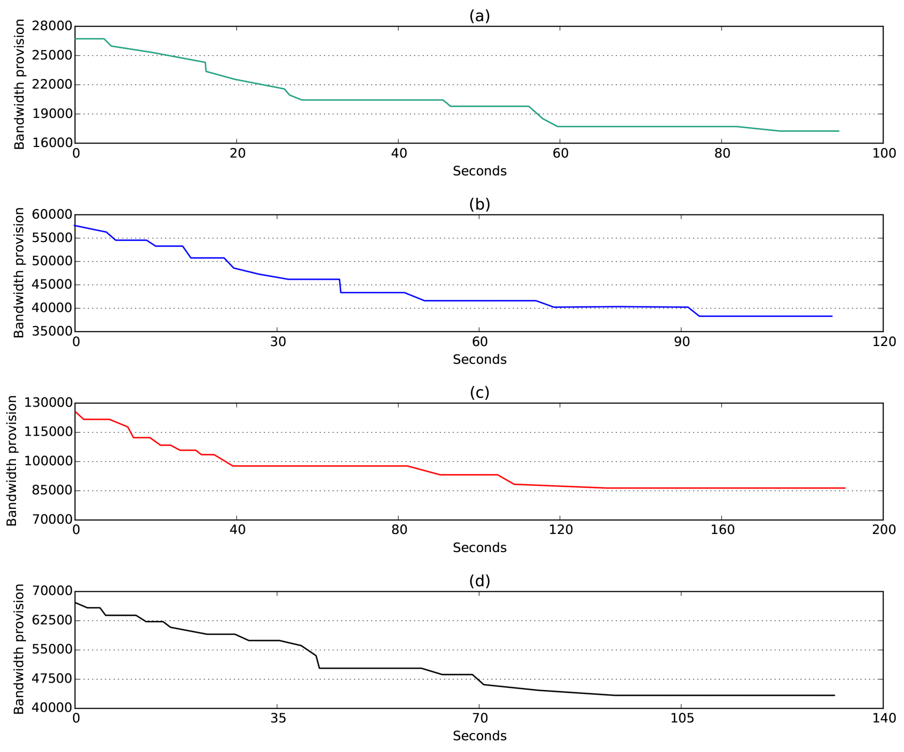


Figure 3.14: Convergence speed of CGA; a) Sparse workflow, b) Median workflow, c) Intensive workflow d) Mixed workflow

munication overhead in the cloud. Simulation results show that the proposed communication-aware framework is able to significantly reduce the communicate cost in the cloud with little increase in a number of used PMs.

This resource provision framework was only modeled for the computing resource amongst cloud services, and two proposed VM-to-PM placement algorithms were focused on reducing communication cost with a increasing number of PMs. As cloud services in most real cases also are affected by its bandwidth provision and the number of used PMs is one of major considerations for cloud providers and tenants, attempts need to be made to address these issues. This will be discussed in detail in the next chapter.

CHAPTER 4
Bandwidth Provision for Cloud Services

Whereas computing resource is the priority for services, bandwidth provision is of an equal importance when the interactive and dynamic data communication within the workflows has concerned. This brings the challenge to determine the bandwidth provision for these services and more specifically for the VMs that host the services. Solving the problem of VM bandwidth provision can help the tenants equip the VMs with proper communication capacity. In EC2, different types of VM instances have different communication capacity and consequently different price rates. Moreover, the data transfer between VMs is also charged in AWS. An exemplar application of this work is that when an enterprise tenant purchases the VMs in EC2 to build a business cloud platform, offering to its users a rich set of interacting services, this work can help the enterprise decide which type of VM instance is most appropriate for each service, so that the VMs are able to fulfil the communication requirement inherent in the business cloud while the enterprise does not pay unnecessary extra bills for VMs with higher bandwidth.

This chapter aims to address this challenge by developing a Communication Input-Output (CIO) model for data communication among services. It borrows the idea from Leontief's Input-Output model in Economy and captures the interaction relation and impact among services. The data communication performed by each service can be calculated from the model. Knowing data communication performed by a service does not necessarily mean that the solution is apparent to the problem of bandwidth provision for the service's VMs. This is because if two VMs of two communicating services are consolidated into the same PM, the data transmission between these two VMs does not consume their bandwidth. Generally, even if the bandwidth provision for the services is determined, the bandwidth provision for each individual VM still depends on the specific VM-to-PM mapping. A lot of existing work has investigated the methods to find the VM-to-PM mapping with the minimal number of PMs. However, previous work does not take into account the non-deterministic nature of service interaction when they design their consolidation strategies. Our studies found that even if the VM-to-PM mapping has the minimal number of PMs, there is still room to further reduce the communication cost in the mapping while maintaining the minimal number of PMs. This chapter designs and implements a Communication-oriented Simulated-Annealing (CSA) algorithm to reduce the total bandwidth provision of all VMs in a set of interacting services. The CSA algorithm takes as input the VM-to-PM mapping with the minimal number of PMs that is generated by the existing strategies. The CSA gradually adjusts the initial VM mapping to generate new mappings with reduced bandwidth provision. The adjustment of VM mappings is designed in the way that it does not increase the number of used PMs.

4.1 Modelling Bandwidth Provision

The proposed framework models the data communication not only in the uplink bandwidth for the sending data amongst interactive services, but also in downlink bandwidth for the receiving data within services. Moreover, the data consumption within the VM-to-PM placement has been analysed by our proposed equations. The following subsections introduce the detailed formulation of each aspect within services communication.

4.1.1 The Communication Input-Output Model

We consider a cloud system as an economy, and each service hosted in the cloud as a sector of this economy. Instead of producing goods, cloud services (sectors) produce and exchange/communicate data over the network infrastructure of the cloud. Whereas goods in a real economy are measured by a common currency that is recognised across different sectors, data produced by services is measured in units of bandwidth across the network infrastructure. Similar to the production of goods in an economy as described by Leontief's model, the cause for the production of data by services is also classifiable as internal and external demands.

Internal demand is the data produced by a service as a consequence of a call from another service. Given two services s_i and s_j from service economy S, we define a *consumption coefficient* c_{ij} as Eq. 4.1, where d_i and d_j denote the average data size produced by s_i and s_j respectively, and p_{ij} denotes the probability that one invocation of s_j causes one invocation of s_i. To understand Eq. 4.1, suppose s_j is able to produce one unit of data per unit of time (e.g. it is allocated with one unit of bandwidth). Since an invocation of s_j produces d_j amount of data on average, s_j can be invoked $1/d_j$ times in a unit of time, so that the allocated bandwidth (one unit) of s_j is able to transfer the amount of data produced by s_j. As a consequence, the number of invocations to s_i is then given by $(1/d_j)p_{ij}$. Therefore, the total amount of data produced by s_i can be obtained by Eq. 4.1. As defined by Eq. 4.1, c_{ij} represents the amount of data produced by service (sector) s_i for each unit of data produced by s_j in a time unit. This is in line with the definition of c_{ij} used in Leontief's model.

$$c_{ij} = \frac{1}{d_j} p_{ji} d_i \tag{4.1}$$

In contrast, external demand in a cloud economy is the data produced by a service due to the invocation requested by external tenants. When a service s_i is at the head of a service workflow (e.g., a login service at the start of a workflow), then the number of times s_i is invoked by the tenants in a time unit (which we call the arrival rate of external requests for service s_i and is denoted by λ_i), together with the average amount of data that an invocation of s_i produces (i.e., d_i), determines the amount of data that will be produced by s_i in a time unit due to the external demand. Therefore, the external data demand for s_i, denoted by a_i, can be calculated by

$$a_i = \lambda_i d_i. \tag{4.2}$$

This definition is also in line with the definition of external demand as defined by Leontief's model. The end tenants of the cloud system who trigger service workflows can be regarded as the open sector of the cloud economy which demands data production from the services.

From these derivations, we can see that a cloud economy shares many similar properties to that of a real economy. By Eq. 4.1 and Eq. 4.2, we are able to apply the philosophy of Leontief's IO model to a cloud setting as follows.

We denote x_i^{out} as size of data produced by s_i in a time unit in order to meet both internal and external demand (we use "out" to indicate that these are the data that need to be sent out from s_i). We can establish the relation shown in Eq. 4.3, where X^{out} and A are vectors of dimension $|S|$ holding the data production (x_i^{out}) and external data demands (a_i in Eq. 4.2) of the cloud economy, respectively, and C is the matrix of c_{ij}. Eq. 4.3 establishes the interdependencies within the cloud economy in terms of data production. x_i^{out} represents the amount of data that may be transmitted over the *uplink* network interface of the PMs that service s_i is hosted in. Note that if s_i and the destination service of some data sent by s_i are located in the same PM, no uplink bandwidth of the PM needs to be consumed for transferring this part of data. In Subsection 4.1.2, we will present how to handle this situation and determine the bandwidth allocation for individual VMs that collectively host service s_i.

$$X^{out} = CX^{out} + A \tag{4.3}$$

In addition to the economy described by Leontief's model, which only considers the amount of goods produced by each sector, we need to calculate the amount of data received by each service in our data demand IO model. This is because Leontief's model does not consider the additional cost associated with a service receiving the data through its host PM's *downlink* network interface.

Among x_i^{out} of data sent by s_i, the amount of $x_i^{out}p_{ij}$ will be sent to s_{ij}. Let c'_{ij} denote the probability that a unit of data produced by s_i is to be received by s_j. Then c'_{ij} can be calculated as $\frac{x_i^{out}p_{ij}}{x_i^{out}} = p_{ij}$. We denote x_{ij}^{out} as the size of data transmitted from s_i to s_j in a time unit and x_{ji}^{in} as the size of data received by s_j from s_i in a time unit, then we have

$$x_{ji}^{in} = x_{ij}^{out} = c'_{ij}x_i^{out}. \tag{4.4}$$

Additionally, we denote x_i^{in} as the size of data consumed by s_i (i.e., received from all services) in a time unit. x_i^{in} can then be calculated by Eq. 4.5, where X^{in} is the vector of x_i^{in} and C' is the matrix of c'_{ij}. Eq. 4.5 establishes the relationship between data production (out) and consumption (in).

$$X^{in} = C'X^{out} \tag{4.5}$$

We summarize the notations used in this section in the first half of Table 4.1, and another half of table we will describe in the following section.

4.1.2 Bandwidth Provision for VMs

From the CIO model, we can derive the amount of data that are communicated by each service. In this section, our objective is to translate this quantity into actual bandwidth provision for individual VMs hosting a service. In a cloud system, each service is hosted by a collection of VMs. We assume that the service is the only service hosted in each of the VMs. This assumption is reasonable since it is a typical setting in clouds to host different cloud services in different VMs so as to provide the isolated service environments.

When two VMs of a pair of services are located on the same PM, data may

Table 4.1: Notations for bandwidth provision

Notation	Definition
The Communication IO Model	
s_i, s_j, S	Services indexed i and j from service universe S
d_i	Average size of data produced by an invocation of s_i
p_{ij}	The probability that one invocation of s_i causes one invocation of s_j
c_{ij}	The amount of VMs produced by service i that have to be consumed by service j in order for service j to produce one VM
x_i^{out}	Size of data produced by s_i in a unit of time
x_i^{in}	Size of data consumed by s_i in a unit of time
x_{ij}	Size of data transmitted from s_i to s_j
Bandwidth Provision for VMs	
VM^i	A virtual machine hosting s_i
PM_k	A physical machine indexed k in the cloud
$\mathcal{M}$	VM to PM allocation scheme
V_i	The number of VM^i in the cloud
v_{ik}	The number of VM^i in PM_k

be transmitted locally and thus does not consume the VMs' physical bandwidth. However, in order to take advantage of this local data transmission channel, the local ratio between the numbers of VMs of two service needs to match their global ratio. This is explained in detail below.

Given a pair of services s_i and s_j from S, V_i and V_j denote the total number of VMs in the cloud for hosting these two services, respectively. Since most clouds, such as AWS, are equipped with Elastic Load Balancing [98] or the fairness scheduler [113] to distribute the load traffic for services, we can assume that the workload of s_i is evenly distributed across all VMs that host s_i. Consequently, the amount of data sent from a VM^i (VM^i denotes a VM that hosts service s_i) to service j can be calculated by $\frac{x_{ij}^{out}}{V_i}$, where x_{ij}^{out} is the data sent by service i to j in a time unit, which is calculated by Eq. 4.3. Given a PM PM_k, v_{ik} and v_{jk} denote the number of VM^i and VM^j in PM_k, respectively. Then in PM_k, the amount of data that are communicated by VM^is to service j is $v_{ik}\frac{x_{ij}}{V_i}$. If $\frac{v_{ik}}{v_{jk}}$ (i.e., the local ratio of the number of VM^i to the number of VM^j

in PM_k) is no greater than $\frac{V_i}{V_j}$ (i.e., the global ratio of the number of VM^i to the number of VM^j in the cloud), all data sent by VM^is in PM_k (the VMs that host service i in PM_k) to service j can be handled by VM^js in PM_k. Therefore, there is no need to consume the bandwidth of VM^i (or VM^j) for sending (or receiving) these data. For example, assume V_i and V_j are 20 and 50, respectively. If in PM_k, v_{ik} is 2 and v_{jk} is 6, then there are more than fair share of VM^j (which is 5) in PM_k to handle the data sent by VM^i in the same machine (since $2/6 < 20/50$).

On the contrary, if the local ratio is greater than the global ratio, which means that there are not adequate VM^j in PM_k to handle the data sent by VM^i in PM_k. The portion of data that cannot be handled by VM^j in PM_k, denoted by y_{ijk}, have to be sent by $VM^i$to VM^j in another PM, PM_l, and therefore consume the uplink bandwidth of VM^i and the downlink bandwidth of VM^j. y_{ijk} can be calculated by Eq. 4.6. Eq. 4.6 essentially compares whether the local ratio is no greater than the global ratio. If so, y_{ijk} is 0. Otherwise, Eq. 4.6 calculates the data that s_i has to send out after deducting the portion of data that can be handled by VM^j in the same machine.

Since y_{ijk} is the data communicated in a time unit, y_{ijk} is essentially the bandwidth that has to be allocated to the VM^is in PM_k for sending data to service s_j. Therefore, $\frac{y_{ijk}}{v_{ik}}$ is the uplink bandwidth that has to be allocated to each VM^i in PM_k for sending the data to s_j, while $\frac{y_{ijk}}{v_{jl}}$ is the downlink bandwidth allocated to each VM^j in PM_l for receiving y_{ijk}. The total uplink bandwidth that needs to be allocated to VM^i in PM_k can be calculated by $\sum_{s_j \in PM_k} y_{ijk}$.

$$y_{ijk} = \max\{v_{ik}\frac{x_{ij}}{V_i}(1 - \frac{v_{jk}(V_i/V_j)}{v_{ik}}), 0\} \tag{4.6}$$

Given a VM-to-PM mapping, denoted by $\mathcal{M}$, the total uplink communication bandwidth generated by $\mathcal{M}$ can be calculated by Eq. 4.7, where y_{ijk} is the amount of data that are sent from VM^i (hosting service i) in PM_k (consuming the uplink bandwidth of PM_k) to VM^j (hosting service j) in other PMs. The total downlink bandwidth generated by a VM-to-PM mapping can be calculated in a similar way.

$$\mathcal{C}(\mathcal{M}) = \sum_k \sum_j \sum_i y_{ijk} \tag{4.7}$$

We summarise some notations used in this section in the second half of Table 4.1.

4.2 The Communication-oriented Simulated Annealing Algorithm

In the classical SA approach, an initial solution is first generated (a solution is encoded) and the neighbourhood searching routine is then applied to generate new suitable candidate solutions. A cost function and the metropolis criterion [105], which models the transition of a thermodynamic system, are used to determine the quality of the solutions and guide the searching direction so that better solutions can be gradually generated until the stopping criterion is met.

In this section, we design a Communication-oriented SA (CSA) algorithm that aims to find the VM-to-PM mapping with the minimal bandwidth provision for all VMs. In the CSA algorithm, the initial solution is set as the VM-to-PM mapping that is generated by the MinPM algorithm [57] (i.e., the algorithm that produce the VM-to-PM mapping that uses the minimal number of PMs to host VMs). The amount of bandwidth provision calculated in Eq. 4.7 is used as the cost function for the CSA algorithm. The CSA algorithm adjusts the VM-to-PM mapping, aiming to reduce the bandwidth provision without increasing the number of PMs. This section presents the encoding of the solution, the neighbourhood searching routine and the entire flow of the CSA algorithm designed in this chapter.

Encoding the solution

In the SA algorithm, a solution is encoded as a two-dimensional array, A, in which an element $a[i][j]$ represents how many VMs of Service s_j there are in PM_i. Note that this encoding method does not differentiate the VMs for the same service. This way, the number of VMs does not affect the complexity of the algorithm. Consequently, the proposed SA algorithm can find the good VM-to-PM mappings efficiently.

Neighbourhood searching

In SA, the design of neighbourhood searching routine is critical for generating good solutions with good efficiency. This subsection presents the method to conduct the neighbourhood searching.

Two probabilities, p_p and p_s, are set to represent the possibility that the VM mapping of a service in a PM is adjusted. An intuitive way to determine the PMs and the services whose VM mapping is adjusted is as follows. Given a current solution (i.e., an encoded VM-to-PM mapping), the routine loops over every PM. In each iteration of the loop, a random number between 0 and 1 is generated. If the random number is greater than p_p, the VM mapping in the PM remains unchanged and the loop moves to the next PM. Otherwise, the VM mapping in the PM is adjusted and the routine enters into a second-layer loop. In this loop, the routine iterates over every service in the PM. In each iteration, a random number between 0 and 1 is generated. If the number is greater than p_s, the VM mapping for this service remains unchanged. Otherwise, the VM mapping of this service is adjusted.

Although the above method is intuitive, the routine has to run a two-layer loop for generating every new solution. In order to improve the efficiency, the following design is adopted for the neighbourhood searching. The neighbourhood searching routine randomly selects $N \times p_p$ PMs (N is the total number of PMs) to adjust the VM mappings of some services in these PMs. For a selected PM, the routine further randomly selects $M \times p_s$ services (assume M is the number of services in the PM) and the VM mappings of these services will be adjusted.

For service s_i in PM_j, its VM mapping is adjusted in the following way. First, the neighbourhood searching routine randomly selects another PM, PM_k, and then randomly selects a service, s_l ($l \neq i$), in PM_k. The routine then tries to swap the VMs between s_i and s_l. In order to render a valid swap, the routine calculates the maximum number of VMs that can be swapped between the two services, which can be calculated using Algorithm 1, where f_k and f_l are the spare resource capacity in PM_k and PM_l, respectively, v_{ik} is the number of VM^i in PM_k, $swap_{ik}$ is the maximum number of VM^i that can be swapped in PM_k. A valid swap is one after which the total capacity of every type of resource (the resource types of CPU utilization, memory and bandwidth are considered in this work) allocated to the VMs in either PM does not exceed the total physical resource capacity of the PM. This validity rule guarantees that the number of required PMs does not increase. The pseudo-codes of the neighbourhood searching routine is presented in Algorithm 2.

As discussed above, the neighbourhood searching routine randomly selects $N \times p_p$ PMs (N is the total number of PMs) and in each selected PM, the routine further selects $M \times p_s$ services to adjust their VM mappings. Therefore, the time complexity of Algorithm 2 is $O(p_p \times N \times p_s \times M)$.

Algorithm 1 Calculating the maximum number of VMs that can be swapped

```
1: if $VM_k^i \times v_{ik} < VM_l^j \times v_{jl}$ then
2:     $Swap_{ik} = v_{ik}$
3:     $Swap_{jl} = [\frac{VM_k^i \times v_{ik} + f_k}{VM_l^j}]$
4: else if $VM_k^i \times v_{ik} > VM_l^j \times v_{jl}$ then
5:     $Swap_{jl} = v_{jl}$
6:     $Swap_{jk} = [\frac{VM_l^j \times v_{jl} + f_l}{VM_k^i}]$
7: else
8:     $Swap_{ik} = v_{ik}$
9:     $Swap_{jl} = v_{jl}$
10: end if
```

Algorithm 2 Neighbourhood searching

```
1: Randomly select $\lfloor p_p \times N \rfloor$ PMs
2: for each of these PM do
3:     Randomly select $p_s \times S \mid_k$ services in $PM_k$
4:     for each of the services do
5:         Randomly select a PM, $PM_l (l \neq k)$ and a service $j$ $(j \neq i)$ in $PM_i$
6:         Call Algorithm 1 to calculate the maximum number of VMs in $VM^i$ and $VM^j$ that can form a valid swap
7:         Swap the calculated number of VMs between $s_i$ in $PM_k$ and $s_j$ in $PM_c$
8:     end for
9: end for
10: Return the new VM-to-PM mapping, $\mathcal{M}'$
```

Simulated Annealing

Algorithm 3 outlines the entire SA process aiming to find the optimal VM-to-PM allocation. In the algorithm, T is the initial temperature of the SA process, which is typically set as 1000 [105], and $factor$ is the cool-down factor of the SA process, which is typically set as 0.85[105]. In each iteration, $\mathcal{M}$ is the current VM-to-PM mapping. Algorithm 2 is called to generate a new candidate VM-to-PM mapping, $\mathcal{M}'$ (line 4). Eq. 4.7 is then applied to calculate the communication cost ($\mathcal{C}'(\mathcal{M}')$) of the new mapping $\mathcal{M}'$ (line 5). If $\mathcal{C}'(\mathcal{M}')$ is better(smaller) than that of the current mapping, the algorithm accepts the new mapping and the new mapping becomes the current mapping (line 6-8). Otherwise, the metropolis criterion, calculated by $\exp(\frac{-\Delta\mathcal{C}(\mathcal{M})}{T})$, is used to decide whether this new but worse VM mapping should be accepted. If the calculated metropolis criterion is greater than a float number randomly generated between 0 and 1 (line 7), $\mathcal{M}'$ is accepted. Otherwise, the current mapping remains intact. The iteration repeats until the current mapping stays unchanged for a certain number of consecutive iterations (counted by j) or the number of iterations (counted by i) reaches a pre-set number, k_{max1} and k_{max2} in the algorithm (line 2).

There are at most k_{max2} iterations in the “while” loop in Algorithm 3. In each iteration, calling Algorithm 2 dominates the time spent in an iteration.

Therefore, the time complexity of Algorithm 3 is $O(k_{max2}p_pNp_sM)$. In our experiments, we found that the solutions generated by the CSA algorithm have stabilised when the "while" loop iterates for 500 times (i.e., k_{max2} is set as 500) for the system scale of 500 services and around 1000 PMs .

Algorithm 3 The Communication-oriented Simulated Annealing Algorithm

```
Require: $\mathcal{M}$
1: $i = 0, j = 0$
2: while $j \leq k_{max1}$ or $i \leq k_{max2}$ do
3:     $T \leftarrow T \times factor$
4:     $\mathcal{M}' \leftarrow$ Call Algorithm 2
5:     $C'(\mathcal{M}') \leftarrow$ Call $Eq.$4.7
6:     $\Delta C(\mathcal{M}) \leftarrow C'(\mathcal{M}') - C(\mathcal{M})$
7:     if $\Delta\mathcal{C}(\mathcal{M}) < 0$ or $\exp(\frac{-\Delta\mathcal{C}(\mathcal{M})}{T}) > \boldsymbol{R}(0,1)$ then
8:         $\mathcal{M} \leftarrow \mathcal{M}'$
9:         $j = 0$
10:    else
11:        $j = j + 1$
12:    end if
13:    $i = i + 1$
14: end while
```

4.3 Performance Evaluation

We have conducted the simulation experiments to evaluate the effectiveness of the CIO model and the CSA algorithm developed in this work. Both trace from the Bing trace as we have used in Chapter 3.5, and synthetic trace are used in the simulations.

In the experiments, 500 services are generated. Reference [17] surveyed the number of VMs that a service has in Bing's datacenter. The results are summarized in Table 4.2, which shows the percentage of services that has a certain number of VMs. For example, 25% of services have 1-2 VMs each. It can be seen that most services are "small" and 80% of services have no greater than 10 VMs. This distribution of the number of VMs in a service is used to generate the VMs for services in our simulations.

The synthetic trace is generated in the following way. A set of 500 services

Table 4.2: Percentage of Services comprising different numbers of VMs

% Services	25%	25%	20%	10%	18%	2%
Number of VMs	[1, 2]	[3, 5]	[6, 7]	[8, 10]	[11, 99]	[100, 10^4]

are generated. A service is defined as the start service, from which all workflows in the trace start. Another service is defined as the end service, which means that when the workflow reaches to this service, it stops and will not invoke further services. The degree of parallelism (denoted by DP) is set, which is 3 by default, when generating the workflow instances for the synthetic trace. For all services except the end service, after a service (e.g., s_i) invoked by a task is completed, it further randomly invokes DP (e.g., 3) services. The roulette wheel method is used to randomly determine which DP services are selected based on p_{ij}. In the synthetic trace, the value of p_{ij} is randomly set from the range of [0.001, 0.003] with the average of 0.002 (i.e., 1/500, where 500 is the number of services generated in the trace). The workflow instance stops growing when all branches in the workflow reach the end service. The technique presented in [24] is used to calculate the number of VMs for each service. In both synthetic trace and Bing trace, the strategy presented in [57] is used to generate the initial VM-to-PM mapping with the minimal number of PMs.

Our evaluation covers the following main aspects: 1) evaluating the effectiveness of the developed CIO Model in calculating the appropriate bandwidth allocation for the services deployed in the cloud, 2) evaluating the effectiveness of the proposed CSA algorithm, 3) evaluating the efficiency of the CSA algorithm, and 4) evaluating the effectiveness of applying the developed IO Model on the AWS cloud platform.

4.3.1 Accuracy of the CIO Model

The experiments presented in this subsection investigate the impact of communication pattern on the bandwidth allocation and the accuracy of the proposed IO model.

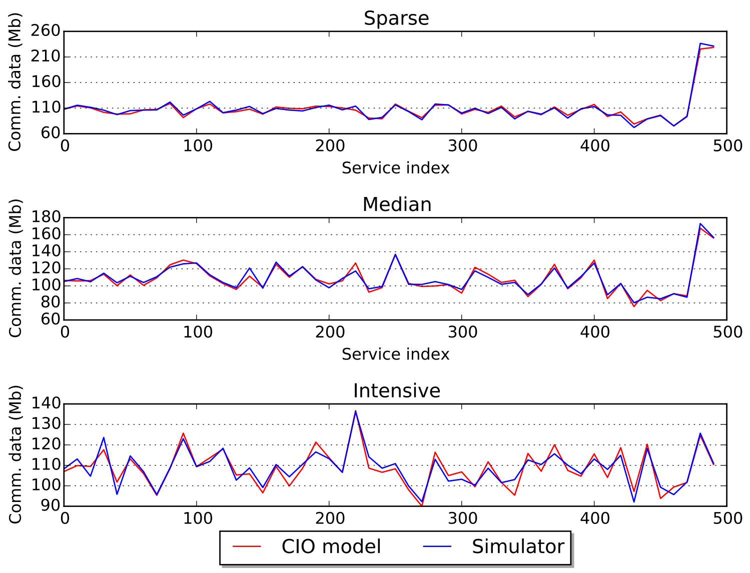


Figure 4.1: Accuracy of the CIO model using the Bing trace

In the Bing trace, the probability that Service i invokes Service j (i.e., p_{ij} in the proposed bandwidth IO model) is set as the communication intensity between Service i and j divided by the sum of the communication intensity between Service i and any other service. On the another hand, for the synthetic trace, it is straightforward to determine p_{ij} since a service randomly invokes another service in the service set. With p_{ij}, we apply the bandwidth IO model to calculate the bandwidth allocated for each service.

In the simulator developed in this work, we allocate the calculated bandwidth to the services and then run the simulation experiments. We record the amount of data that are communicated by each service. If the proposed bandwidth IO model is effective, then the amount of data that are communicated by each service in a time unit in the simulation experiment should equal to the bandwidth allocated to each service.

The experimental results for the Bing trace are shown in Figure 4.1. It can be seen from this figure that our bandwidth IO model is fairly accurate. The experimental records show that the average percentages of discrepancy between the results obtained by the CIO model and those by the simulation experiments

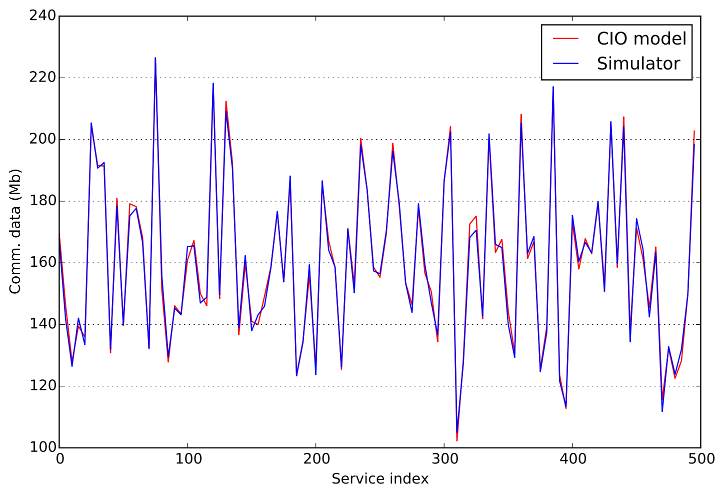


Figure 4.2: Accuracy of the CIO model using synthetic traces

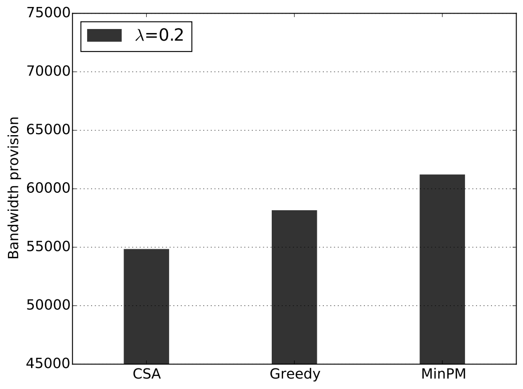
are 0.8%, 1.3% and 3.4% for Sparse, Median and Intensive communication patterns, respectively. The reason why the accuracy of the CIO model decreases as the communication pattern intensifies may be because as the communication pattern changes from Sparse to Intensive, it becomes increasingly more indeterminstic that which service a service invokes each time. Such increasing dynamic makes the CIO model less accurate. The results for the synthetic trace is shown in Figure 4.2. The mean percentage of discrepancy between the CIO model and simulation experiments is 1.3%, which once again suggest that the CIO model is able to capture the bandwidth demands accurately.

4.3.2 The Effectiveness of CSA

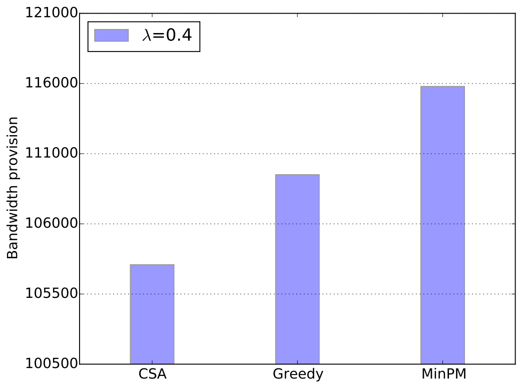
The experiments in this subsection investigate the effectiveness of the proposed Simulated Annealing (SA) algorithm. In the experiments, we first used the methods proposed in [57], which we call the MinPM algorithm in this Chapter, to obtain the VM-to-PM mapping that uses the minimal number of PMs to host the VMs. We then apply the proposed SA algorithm to further adjust the VM-to-PM mapping in order to reduce the communication cost without increasing the number of PMs. We also used the greedy method presented in [54] to perform the VM-to-PM mapping and compared the results against those generated by the proposed SA. In the greedy algorithm, all services are ranked in the decreasing order of their communication intensity (i.e., the data that have to be communicated by a service in this Chapter). The greedy algorithm first place the VMs of the first service (i.e., the one with most communication intensity) on PMs, with each PM having the same number of VMs or having at most ± 1 difference if it can not be evenly divided). Then the greedy algorithm selects the next service, s_2, and tries to place its VMs to PMs so that the local ratio of the number of VMs of s_1 to that of s_2 in a PM equal (or is the closest) to the global ratio of the total number of VMs of s_1 to that of s_2. The procedure repeats until all VMs are mapped.

The experiment results for the Bing trace are presented in Figure 4.3(d). It can be seen from this figure that in all cases, SA significantly reduces the communication cost compared with other two methods. These results indicate the effectiveness of the proposed SA algorithm. Further observation shows that the advantages of CSA over other algorithms decrease when the communication intensity increases. For example, when the communication pattern is sparse, the communication cost obtained by CSA is less than that by MinPM by about 44%, while the cost is reduced by only 17% when the communication pattern is intensive. This result can be explained as follows. When the communication pattern is intensive, a service has almost equal possibility to communicate with any of other services. Therefore, the VM-to-PM mapping methods are less important in terms of reducing the communication cost. On the contrary, when the communication pattern is sparse, overall communication cost is more sensitive to the mapping methods.

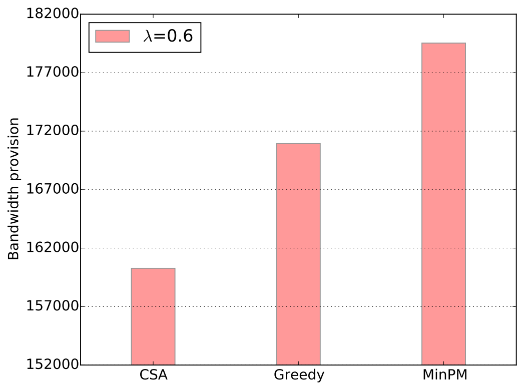
For the synthetic trace, we increase the arrival rate of the generated workflows and use the technique presented in [24] to calculate the number of VMs for each service under different arrival rates. The experimental results are shown in Figure 4.3(a, b and c). It can be seen that CSA still outperforms other two algorithms. However, the advantage is not very prominent. Comparing with



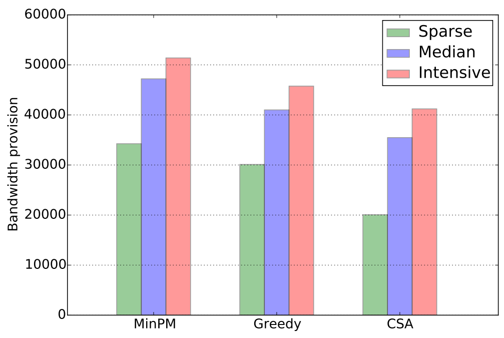
(a) Synthetic workflow's arrival rate is 0.2



(b) Synthetic workflow's arrival rate is 0.4



(c) Synthetic workflow's arrival rate is 0.6



(d) Bing trace

Figure 4.3: Comparing CSA with other existing algorithms using synthetic and Bing trace

MinPM, CSA reduces the communication cost by about 10%. This trend is consistent with that observed in Figure 4.3(d). This is because in the synthetic trace, a service randomly invokes (i.e., has equal opportunity to invoke) another service. According to the analysis of Figure 4.3(d), it is reasonable that CSA has the diminished advantage over other two algorithms under the synthetic trace.

4.3.3 The Efficiency of CSA

Figure4.4 shows the convergence of the proposed CSA algorithm over time under different communication patterns of the Bing trace. It can be observed from these figures that the bandwidth provision decreases dramatically in the first few seconds in all cases (about 4.5 seconds) and then the trend gradually tails off, which shows that the proposed CSA algorithm is efficient in reducing the bandwidth provision. This is because a number of optimization measures are designed in searching neighbourhood solutions as well as in calculating the bandwidth provision of a VM-to-PM mapping.

Further observations show that the convergence time remains almost the same as the communication pattern changes from "sparse" to "intensive", which suggests that the communication pattern does not have much impact on the convergence time. This result is reasonable since the communication pattern is not a main parameter that affects the number of operations in the SA algorithm.

Figure 4.5(a) and 4.5(b) aim to investigate the convergence time spent by CSA over the number of services and the number of PMs, respectively. The synthetic trace is used in these experiments. The y-axis is the time spent by CSA to establish the stable sub-optimal bandwidth provision, i.e., convergence time. Note that in the experiments, the number of PMs is determined by the workflow arrival rate. We set the size of VM so that the average number of VMs in a PM is fixed (in Figure 4.5(b), the average number of VMs in a PM is 8). Then the number of PMs increases as the workflow arrival rate increases. This is why the x-axis of Figure 4.5(b) is the arrival rate. In Figure 4.5(a), the

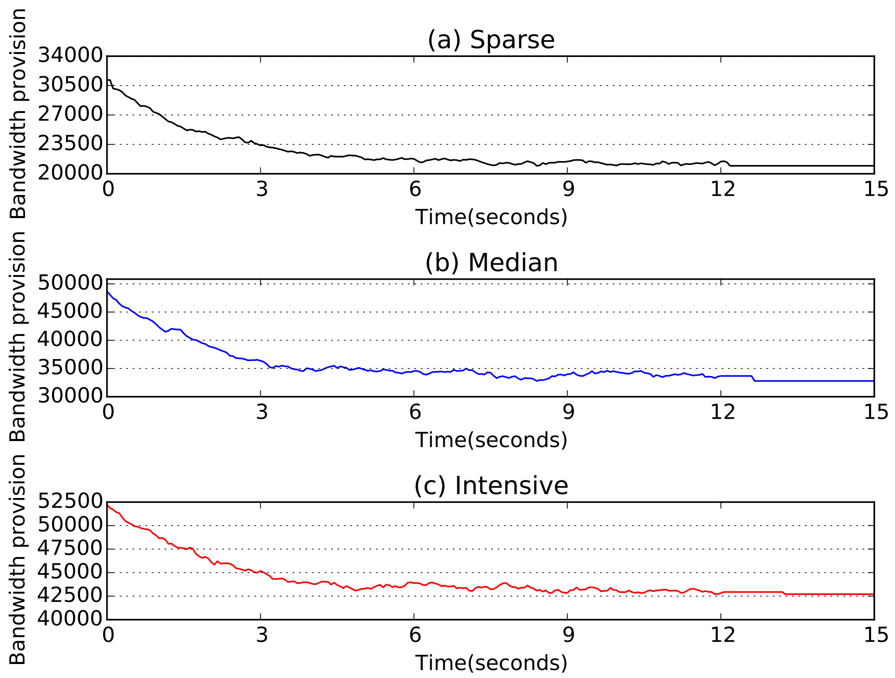


Figure 4.4: Convergence of CSA using the Bing trace

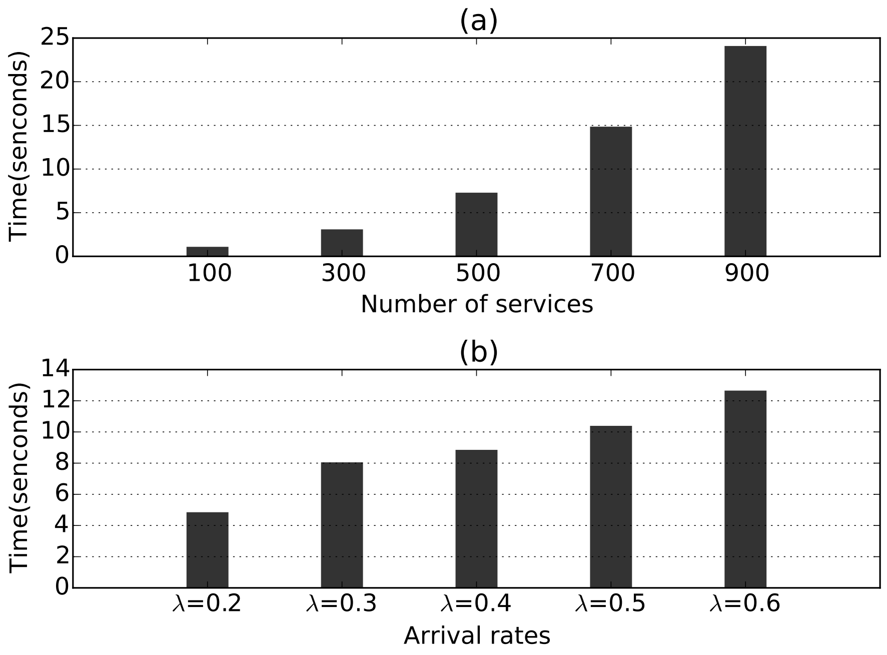


Figure 4.5: Convergence time of CSA over a) the number of services and b) workflow arrival rate. The synethic trace is used

average number of VMs in a PM is set to be 8 and the arrival rate is set to be 0.2.

As seen in Figure 4.5(a), the convergence time increases as the number of services increases from 100 to 900, as to be expected. This is because as the number of service increases, CSA loops over more services to calculate the local ratios for each service pair.

In Figure 4.5(b), the numbers of PMs are 476, 686, 896, 1105 and 1314 as the arrival rate increases from 0.2 to 0.6 with increment of 0.1. As can be seen from this figure, the convergence time increases as the number of PMs increases, which is also to be expected. This is because in each iteration, CSA has to calculate the local ratios of service pairs in each PM.

4.3.4 CIO Model on a AWS Testbed

We deployed a testbed in AWS cloud system to verify the effectiveness and show the applicability of the proposed CIO model. The AWS testbed comprises six AWS EC2 instances with each representing a different service. Note the purpose of this experiment is to verify the effectiveness of the CIO model, not the CAS algorithm since we cannot control the VM-to-PM mapping in AWS. This is why we only use one EC2 instance to host a service. In order to ensure these EC2 instances are not allocated on the same physical node, we deploy each instance on each of three availability zones in two regions in Europe (Ireland and Frankfurt).

The interaction pattern among services are generated in the following way. The six services are put into three service groups. Services 1 and 2 are in group 1. Services 3 and 4 are in group 2. Services 5 and 6 are in group 3. In the experiments, it is set that a service has 70% probability to invoke the other service in the same group and 30% probability to invoke any of other services in a different group.

Service invocations are generated in the following way. A tenant (a computer in our lab) initiates a stream of service requests to the services in the deployed

Table 4.3: Amazon EC2 instance configurations-1

EC2 configuration	m3.large	m3.4xlarge	c4.4xlarge	c4.8xlarge
Max bandwidth (MB/s)	62.5	125	250	500

AWS testbed following the Poison process with average arrival rate of λ. When a request is sent to the AWS testbed, the data of size d_{in} is also sent with the request, which represents the data communication of the cloud system with the open sector in the IO model. A request randomly invokes a service (run in an EC2 instance), which then kicks off a workflow of service invocations following the service interaction pattern specified above. When a service invokes another service, the data of size d_{intra} is also sent from the former service to the latter. When n service invocations are performed, n-th service will not invoke further services, which represents that the workflow of service invocations that serves the initial request sent by the tenant is completed and that the workflow contains n tasks. After a workflow is completed, the last service sends the data of size d_{out} back to the tenant, which represents the cloud system returns the results back to the tenant. In the experimental results presented in this subsection, d_{in}, d_{intra} and d_{out} are set as 200MB. λ and n are set to be 1 and 10, respectively. Since we focus on investigating bandwidth provision in this experiment, we let each service have enough capacity to process the computation tasks and set the computing time of a service invocation to be one second.

We apply the CIO model to compute the bandwidth provision of each service (i.e., each EC2 instance), which is shown in Table 4.4. AWS provides a list of EC2 instances with different bandwidth configurations, some of which are listed in Table 4.3. The tenants can select the EC2 instances from them. For a service, we select the EC2 instance with the bandwidth configuration bigger but closest to the bandwidth provision computed by the CIO model. The selected EC2 instances for each service and their corresponding bandwidth configurations are also shown in Table 4.4.

In the experiments, we record the average time spent by the testbed to

Table 4.4: IO Model bandwidth configuration

Service index	1	2	3
IO Model (Mb/s)	431.6	200	248
Instance type	c4.8xlarge	c4.4xlarge	c4.4xlarge
Bandwidth (Mb/s)	500	250	250
Pricing ($/hour)	2.112	1.056	1.056
Service index	4	5	6
IO Model (Mb/s)	366	532	465
Instant type	c4.8xlarge	c4.8xlarge	c4.8xlarge
Bandwidth (Mb/s)	500	500	500
Pricing ($/hour)	2.112	2.112	2.112

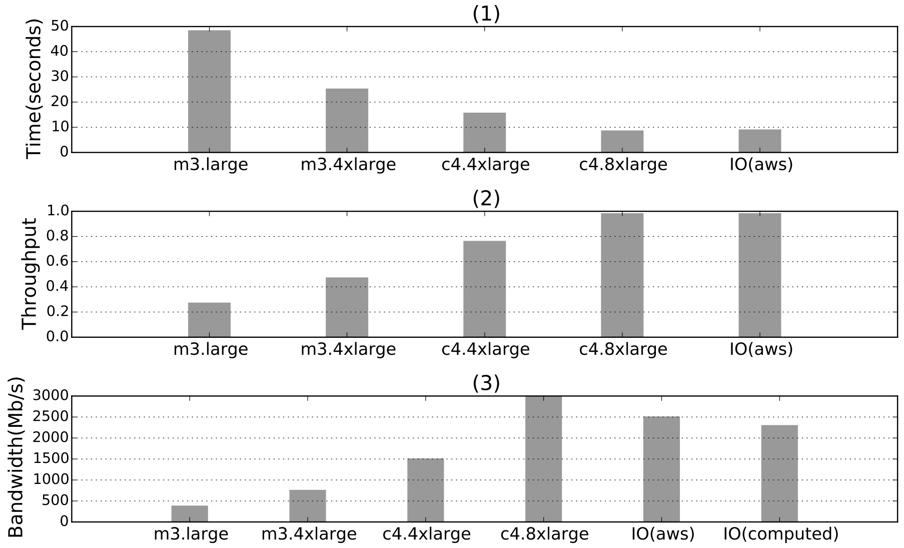


Figure 4.6: Performance comparison with different configurations of EC2 instances

finish a workflow, i.e, the duration between the time when the tenant sends the initial request and the time when the tenant receives the results sent by the testbed, which is shown in Figure 4.6.1 (the column labelled by IO(aws)). To investigate the effectiveness of the CIO model, we also ran the experiments with other EC2 instance configurations for the services. For example, the column labelled by m3.large corresponds to the performance obtained by running the experiment with each service being run in a m3.large instance (this is reasonable because tenants tend to select the same configuration for all EC2 instances in their system). It can be seen that only when the experiment is run with the c4.8xlarge instances (most bandwidth and therefore most expensive as shown in Table 4.4), the same performance is obtained as that by the configurations computed by our IO model.

We also record the throughput of the testbed in terms of the number of service invocations processed by a testbed in one second, which is shown in Figure 4.6.2 (the column "IO(aws)"). Note that a service invocation measured in this experiment includes receiving the data from the previous service or the tenant, performing the computation, and sending the data to the next service or the tenant. Similar as in Figure 4.6.1, we also present the throughput obtained by using other EC2 instance configurations. Once again, the performance obtained by the CIO model is same as that by the configurations in which all services are run by most expensive instances, c4.8xlarge.

Figure 4.6.3 shows total bandwidth allocated to all services when running the experiment with different configurations. The column "IO(computed)" is the actual total bandwidth computed by the IO model while The column "IO(aws)" is the total bandwidth of the EC2 instances selected according to the CIO model. It can be seen that by applying the CIO model, the testbed needs less bandwidth (therefore less money as shown in Table 4.4), but still deliver the same performance (as shown in Figure 4.6.1 and 4.6.2).

4.3.5 Cloud-IO Model in AWS

We build our testbed cloud system by several deployed services [66] to complete a workflow that encodes the original 1080p video files to produce different video formats from 720p to 144p. There are four video services and one front service in the testbed, and the front service plays the role as a central scheduler receiving the demand requests with the original video file from the tenant and dispatches it to those video services to encode the required formats as in Figure 4.7. For each encoding service, after it received the original video file, it will split the original video as several segments and send them to another service to encode, separately. Thus, each service will need to cooperate with other services together to complete the original video encoding. The service has the encoding video segments that needs to run the computation tasks to complete the encoding. When all segments of video have encoded the required format, the first video service will gather and join them as the result data sending it to the front service so that the tenant's demand request is completed. Moreover, four video services are divided into 2 groups. Group 1 contains service 1 and service 2, service 3 and service four are allocated in the group 2. In the experiment, a service has been set with 70% probability to send its video segments to another service in the same group, and 30% probability to invoke any of other services in a different group.

We represent each service as one AWS VPC, and each VPC could be deployed in two different regions in two availability zones within Europe (Ireland and Frankfurt). For one service represented by a VPC, there is a collection of VM instances to do its computation tasks, and one extra VM instance is responsible for the load balance and communication traffic amongst its internal VMs instances and others VPC (services). In addition, as the front service only transfers the original video file to other video services without doing computation tasks, we record the performance results of an average time window within an one-day experiment to investigate the effectiveness of cloud-IO model by four video services on processing 40 requests.

For this experiment, we initiate a stream of tenant demand requests that each request contains the video files to all services and the related encoding tasks as one minute time interval generated from the machine in our lab. The size of video file and the number of encoding tasks are two random variables based on the Gaussian distribution, 100 Mb, 20 encoding tasks are the mean values, and 30 Mb, 5 encoding tasks are the standard deviation values for the video file and

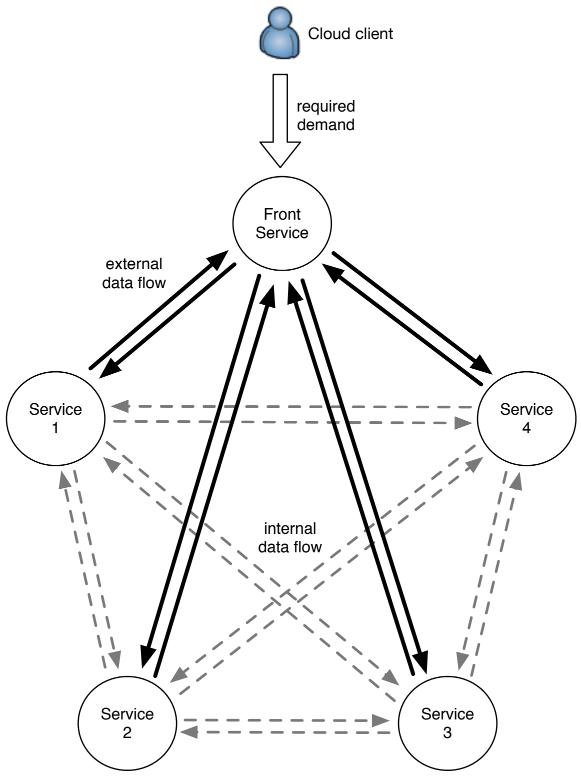


Figure 4.7: The illustration of AWS testbed workflow

Table 4.5: Amazon EC2 instance configuration-2

EC2 configuration	m3.large	m3.4xlarge	c4.4xlarge	c4.8xlarge
Bandwidth (Mb/s)	62.5	125	250	500
vCPUs	2	8	16	32
Processing encoding task (per minute)	1	4	8	16
Pricing cost ($ per hour)	0.146	0.616	1.056	2.112

encoding tasks, respectively. AWS has provided a list of EC2 instances with different configurations and pricing cost, which we listed in the Table 4.5, thus we compare the testbed cloud systems with different configurations chosen from those configurations and the configuration computed by our cloud-IO Model.

In the beginning of experiment, we assign those four AWS configurations with VMs capacity that can only meet the external computational demands. To compare the performance and resource cost with different configurations, we indicate them as four black bars in Figure 4.8. In addition, we gradually increase 2 additional VMs for each service at each configuration to see the impact of performance, which we note them as in the figure with different colour bars, respectively. We can see the throughput and completion time have not been improved too much even its configuration has the increasing number of VMs. Especially, for m3.large, it barely has no improvement on the throughput when the VMs are increasing. It is because most of data-flows including sending and receiving amongst four video services are around 400 Mb, and the performance bottleneck for those configurations is the bandwidth. On the contrary, the configuration of c4.8xlarge has the performance bottleneck on its processing ability. This configuration has reached the peak performance after increasing 2 VMs for each service, but its performance keeps in constant even with increasing VMs number. However, we can see the EC2 configuration computed by cloud-IO Model can reach the best performance by without increasing the number of VMs.

Furthermore, if we look deeper into those configurations in the Figure 4.9, which we noted the resources details and pricing cost for those testbeds, we can see more differences among those configurations. It is no surprise that the increasing VMs amount incurs higher financial cost for the configuration, except for the configuration of c4.8xlarge, due to the shorter time for completing requests and it used less money than its initial one. However, the throughput cannot be improved when more VMs are added up with the climbing pricing cost. And even it has the similar performance result with the configuration

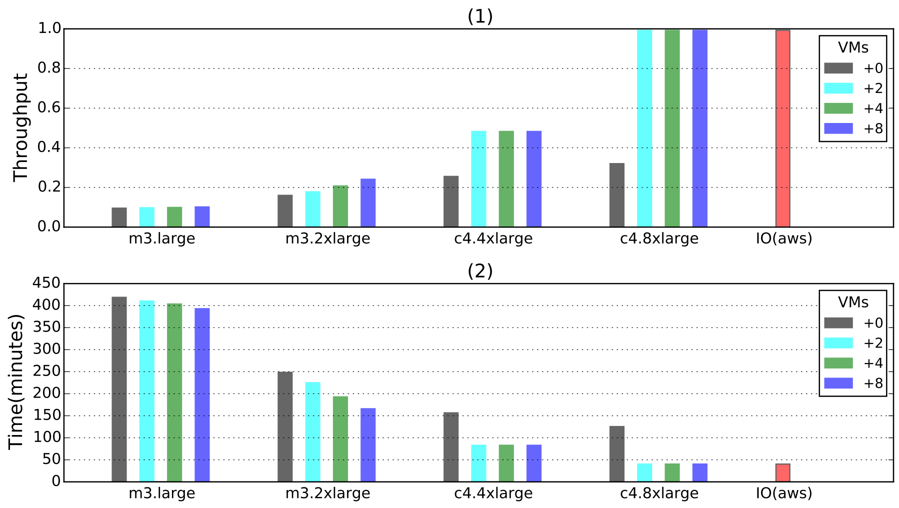


Figure 4.8: The comparisons of AWS Configurations in Performance

computed by cloud-IO model, it costs more money for using too many powerful VMs resource. On the another hand, the configuration computed by cloud-IO model still can obtain the maximum performance with less and cheaper VMs configuration and bandwidth provision. Moreover, for Figure 4.9(3), the last two red bars represent the configuration we deployed in AWS and actual bandwidth provision computed by cloud-IO model, we can see that a smaller bandwidth provision could be adopted if we can have more suitable EC2 instance configuration.

4.3.6 AWS VM-to-PM Placement

Since we are unable to control the actual VM-to-PM placement in the AWS, to directly verify the impact on performance of VM-to-PM placement for our AWS testbed is intractable. However, AWS restricts the number of VPCs as up to five in the same region for each cloud tenant, and the same bandwidth configuration within one region can communicate via AWS private internal IP rather than the public Internet IP, which significantly surpasses the communication between two different regions with the public IP as we shown in Table 4.6. Furthermore, in AWS, there is no extra pricing cost for communication via the internal IP, but

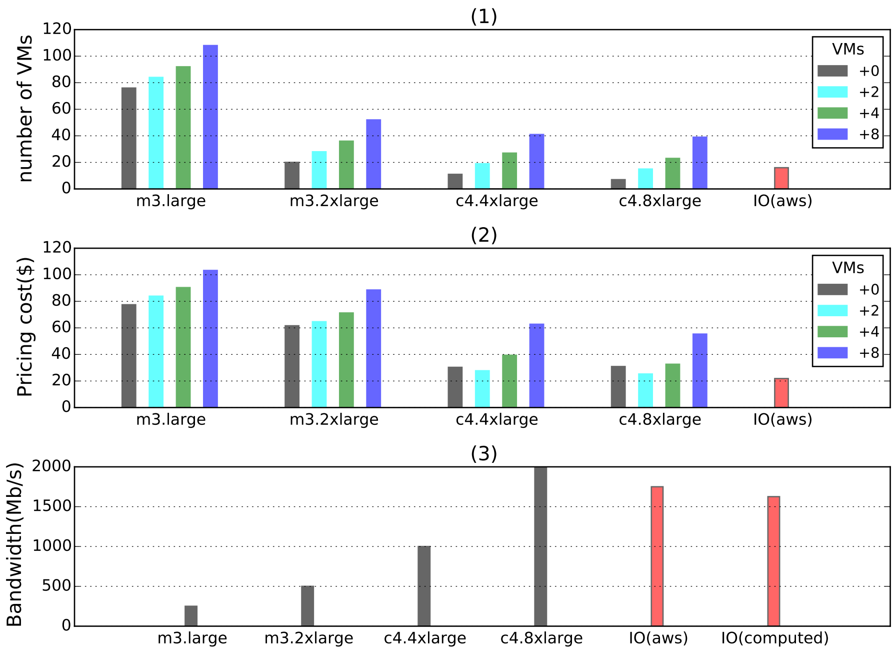


Figure 4.9: The comparisons of AWS Configurations in Resource Cost

AWS charges in the sending and receiving for $0.04 per GB data per hour. Thus, these key elements have shared the analogies in our scenario of VM-to-PM placement, we adopt the VPC-to-Region placement to verify the impact on the performance of different VM-to-PM placements, and we choose CAGA as our placement algorithm, because the AWS region treats all VPCs equally so that there is no differences to allocate the all VPCs by different placement algorithms in terms of the number of regions.

In this experiment, we still use the testbed with four services for video encoding. Because this aim of VM-to-PM placement is the bandwidth provision rather than the number of VMs, we set each service with three VMs to do the encoding tasks for the simplicity, and making it is enough to complete our computation tasks. To match the VM-PM placement with AWS VPC-to-region, we allocate each VM into one VPC. With the limitations of AWS VPC, there are at least 3 different regions that to allocate 4 services (VPCs) with 12 VMs in total. Hence, this evaluation focuses on the different placements of 12 VMs-to-3

Table 4.6: AWS EC2 bandwidth in public/private IP

EC2	m.3.large		m3.4xlarge	
configuration	public IP	private IP	public IP	private IP
Bandwidth (Mb/s)	62.5	$\boxed{400}$	125	$\boxed{600}$

EC2	c4.4xlarge		c4.8xlarge	
configuration	public IP	private IP	public IP	private IP
Bandwidth (Mb/s)	250	$\boxed{1000}$	500	$\boxed{1500}$

regions.

For the tenant demand request, we adopt the same way as the last one by generating a stream of demand requests for four services with one minute as the time interval to record an average time window of completing 40 requests.

We firstly compare the testbeds with 100Mb demand requests in different placements to see the differences on the throughput and the pricing cost based on the configurations that ranges from Non-IO Model and Non-CAGA to cloud-IO Model and CAGA as we display it in the Figure 4.10. For those configuration polities either Non-IO model or Non-CAGA, we choose m3.4xlarge EC2 instance type (125 Mb/s) as our default EC2 instance, and a random placement method, respectively. It is obvious that the configurations with either cloud-IO Model or CAGA have enhanced the throughput of testbed as the maximum, and the time cost is reduced at the minimum. But these configurations incur the significant difference at the pricing costs as in the Figure 4.10(3), for the policy of CAGA with the optimal VMs-to-Region placement uses the communication data-flow among services via EC2 internal IP, it gains the bandwidth and reduces the unnecessary bandwidth provision cost comparing to the policy of cloud-IO model without CAGA.

However, only relying on CAGA that cannot guarantee the "best" performance whilst we increase the demand request at 200 Mb as we represent in the Figure 4.11. Although the cloud-IO model still can keep its maximum performance with the increasing demand requests, the pricing cost of cloud-IO model

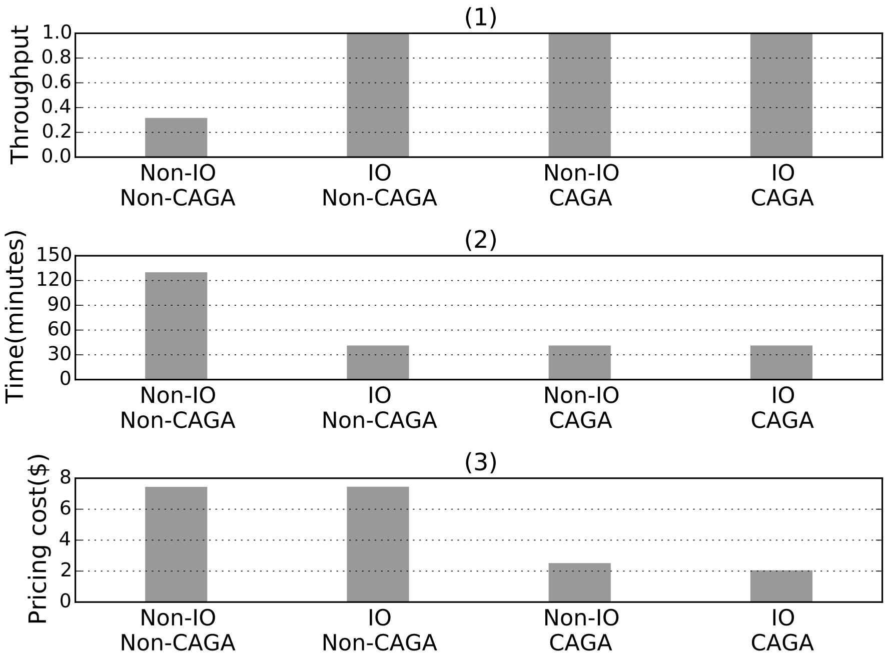


Figure 4.10: VMs-to-Regions placements with requests (100 MB/s)

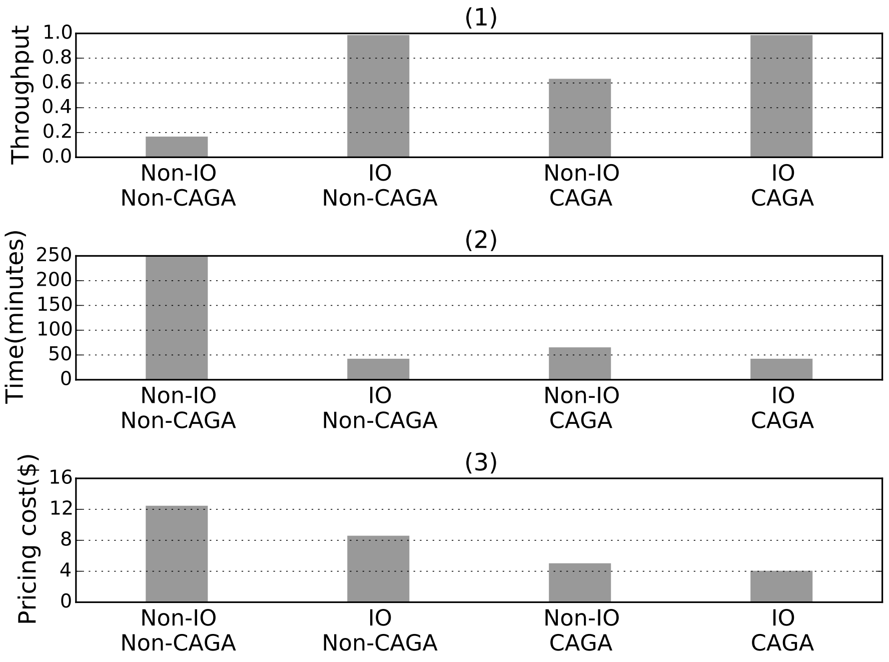


Figure 4.11: VMs-to-Regions placements with increasing requests (200 MB/s)

without CAGA($8.488) is higher than the one with CAGA($4) over double times, as CAGA has successively taken the advantages of better VM-to-Region placement and it chooses the cheapest VMs.

4.4 Summary

This chapter presented a Communication Input-Output (CIO) model, which extends the economical Input-Output model to model data communication among cloud services. Based on the CIO model, the method of determining the bandwidth provision for VMs is developed. Further, a Communication-oriented Simulated Annealing (CSA) algorithm is developed. The CSA algorithm takes the VM-to-PM placement with the minimal number of PMs as the initial mapping and adjusts this mapping iteration by iteration, aiming to obtain the VM-to-PM placement with the minimal bandwidth provision without increasing the number of PMs used. The simulation and real experiments were conducted to verify the effectiveness of the proposed CIO model and the proposed VM-to-PM algorithms. The last but not the least, we have combined two IO models on computing and bandwidth resources as a complete cloud-IO model, and the cloud-IO model has been implemented in the real AWS cloud platform with the VM-to-PM placement algorithm to verify its correctness and effectiveness.

By the proposed cloud-IO model and VM-to-PM algorithms, the cloud tenants and providers can model and decide the resource provision and allocation within a cloud. Owing to the increasing scale of job requests in the cloud, the emerging architecture of distributed schedulers has been implemented to replace the traditional central cluster scheduler. But this nascent architecture has also incurred new challenges for cloud computing, in terms of scheduling the above resources provision and allocation. Scheduling issues within the cloud will be discussed in the next chapter.

CHAPTER 5
Job Scheduling in the Cloud

Data centres nowadays have to process the massive scale of jobs on a daily basis. On one hand, the resource demand, along with the commercial success of cloud computing, becomes the major driving force for the cloud providers to increase the size of their clusters. On the other hand, in order to handle the jobs efficiently, cloud giants, such as Google, Microsoft and Amazon, have developed various cluster management frameworks in their production clusters [107]. Among them, one conventional approach is to develop a centralized scheduler in the cluster, which manages all and diverse types of job submitted to the cluster. However, because of the massive number of the jobs and the complexity of making scheduling decisions for some types of job, the centralized schedulers become the performance bottleneck for delivering resources and processing jobs timely. A recent trend thus is to deploy multiple, independent schedulers in a cluster. Different schedulers make scheduling decisions simultaneously for different types of job, aiming to improve the throughput and cluster utilization. These independently working schedulers in a cluster are termed *distributed schedulers* in literature [18, 88, 95].

In data centres, a job typically runs in a resource container, the examples of which are Linux Docker and VM [58]. A resource container may consume a certain amount of various types of resources such as CPU, memory, storage and bandwidth. Scheduling decisions involve determining which physical machine a resource container should run in. Since distributed schedulers make the scheduling decisions independently, it is likely that different schedulers decide to place their resource containers in the same physical machine and that the total resource capacity of these resource containers exceed the resource capacity of the physical machine. This situation is called the scheduling conflict between distributed schedulers. It has been shown that the scheduling conflict is a crucial part of performance penalty for distributed schedulers, since the distributed schedulers may spend a long period of time in rescheduling jobs due to scheduling conflict rather than put them into execution.

This scheduling conflict problem has been recognized and the measures have been taken in literature to resolve the conflict. For example, Omega [95] accepts a job even if the scheduling conflict occurs for some of the job's tasks. However, the scheduler will keep rescheduling the tasks that conflict with the scheduling decisions made by other schedulers. Apollo [18] implemented a waiting queue on each machine so that the conflicting tasks are not rejected and return immediately back to the scheduler. If a task stays in the waiting queue for too long, the scheduler will try to schedule some duplicated tasks to other machines to speed up the processing of this task.

Essentially, the existing measures focus on resolving the conflict after it happens, not on preventing scheduling conflicts. Both strategies cause the straggler tasks, which increase the makespan of the whole job since the makespan of a job depends on the slowest constituent task in the job. Also, pending in the machines and running duplicated tasks cost more unnecessary resources. Furthermore, because the scheduler will still hold the resources until all tasks in a job have been completed, the early completed tasks does not release the resources in the machines although the resources are idle.

In this work, we investigate the performance penalty incurred by the scheduling conflicts and the relation among the conflict and the number of resources requested by the schedulers. We then propose a game-theoretical solution for distributed schedulers to improve the job performance. Finally, we conduct the simulation experiments and real experiments on AWS platform to evaluate the effectiveness of the models and the methods proposed in this work.

5.1 Motivating Case for the Strategy

In this section, we review distributed scheduling in clouds and investigate the possible tradeoff between resource demand and conflict overhead in the cloud with distribute schedulers. We also present the opportunities and challenges of improving job performance in such a system.

5.1.1 Workload Character

The exemplar applications running in clouds include web indexing and searching, database services, data batch processing frameworks and such so on. To meet their performance targets, each job is allocated with a certain amount of resources and runs in resource containers, which can consume up to a predefined amount of resources on the target machine [107]. The resource containers can be represented as a vector of different types of resource, such as CPU, memory, storage and bandwidth. For example, if a job request for a resource container is represented as $[2, 5, 2, 300]$, it means that this job demands a target machine that has at least 2 CPU cores, 5GB memory, 2GB storage and 300Mb/s bandwidth so that the requested resource container can be created to run the job.

Different resource allocation polices

In a broad sense, resource allocation policies can be divided into two major types. First, *lenient allocation* is often used for the jobs running in the offline mode, i.e., the jobs can be allocated to run on any machine as long as the allocation does not exceed the machine's resource capacity. For example, most data analysis frameworks, such as Mapreduce [32], Dryad [61] or Spark [116], do not care the type of CPU cores or type of machines for their execution as long as they are allocated with enough resources. Second, some online services, which contain the legacy web applications or latency-intensive jobs, have to run on a particular type of resources or machines [94, 109]. This type of resource allocation is called *strict allocation*, which needs to specify a particular type of

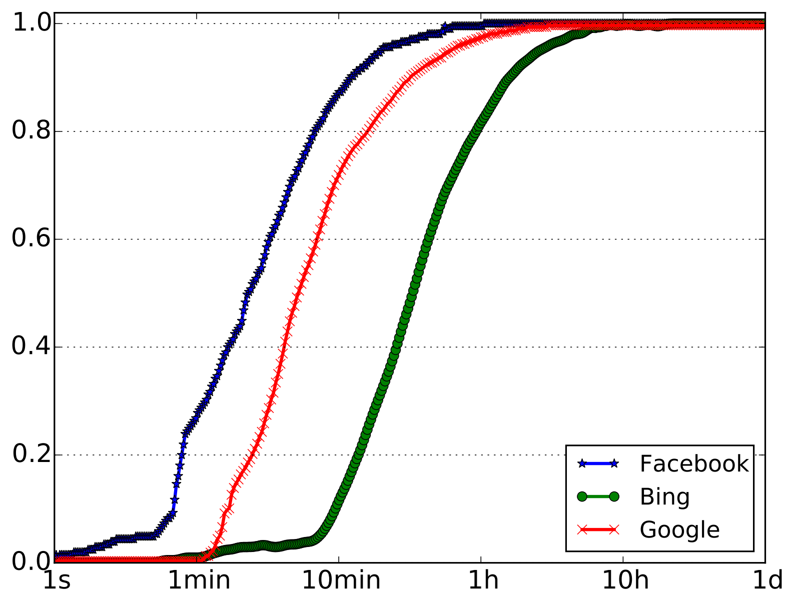


Figure 5.1: Cumulative Distribution Function(CDF) of jobs duration for Bing, Facebook and Google traces

resource containers(i.e., CPU cores or kernel version). If any specified resource container in the strict allocation is occupied by other jobs, the scheduling conflict occurs.

Predictable Job Distribution

In most clusters, there are a significant number of jobs which have repeated profile and follow similar distributions [1, 40, 53, 65]. Multiple instances of a job typically have similar resource requirements and pattern. Figure 5.1 shows the cumulative distribution function of the execution time of the jobs from the production cluster traces in Microsoft Bing, Facebook and Google [5, 60, 62, 93]. As can be seen from the figure, most jobs are short, especially for Facebook and Google trace, with around 80% of jobs taking less than 10 minutes. In all traces, there are a few requests that are very long, spending up to 1 day. Table 5.1 represents the job size distribution for those traces. Nearly 90% of jobs across

Table 5.1: Job size distributions for Bing, Facebook and Google traces

	% of Jobs		
Number of tasks	Facebook	Bing	Google
1-10	85%	43%	60%
11-50	4%	8%	16%
51-150	8%	24%	20%
151-500	2%	23%	2%
>500	1%	2%	2%

three traces contain less than 150 parallel tasks in a job. Only a few percentage of jobs are quite large and each has more than 500 tasks. Therefore, most running jobs are predictable by observing the historical trace and analyzing its probability distribution.

Tolerable Reduction of Resource Demands

There are two main reasons why the schedulers can slightly reduce the jobs' resource demands without major performance degradation. First, in the most production environments, users often deliberately request more than enough resources to account for the occasional load spikes and machine failures [107]. The work in [35, 93] reports that the actual resource usage is lower than the requested resource usage by about $40-50\%$ in most of time. Thus, it is acceptable that the cluster schedulers reduce the resource demand of their jobs when the system load is high or the system lacks spare resources.

Second, most jobs accept approximated results, in terms of deadline or result accuracy. For example, the executions of some offline data analysis jobs can be paused and postponed for a long time (in hours) [40]. Some web searching and real-time advertisement jobs, which are online jobs, allow a small fraction of incomplete results for some tasks in order to have timely completion for the whole job [6, 64]. Similar observation have also been reported in [2]. In turn, this allows us to schedule the jobs with more flexibility in terms of resource allocation.

In summary, we generalize the workload in the productive cluster as online

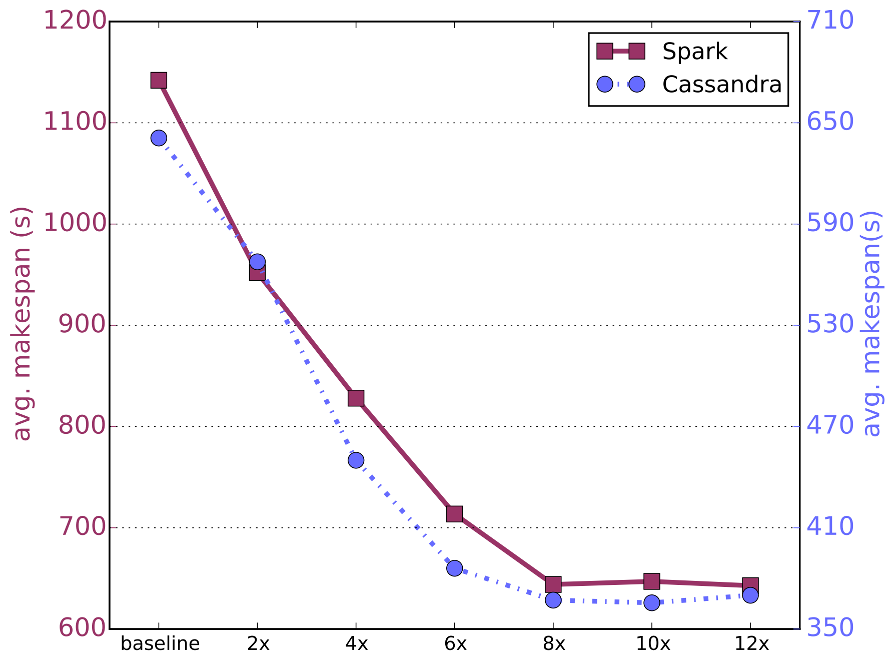


Figure 5.2: Performance gains by increasing the scale of resource containers in the isolated cluster

and offline jobs based on its characteristics and allocation policies, and the descriptive distributions within the workloads that allow us to schedule jobs with guidances. Thus, we can take advantage of running jobs in more flexible scales of resources assignment based on the various conditions.

5.1.2 Performance Gains with Prices

In this subsection, we first investigate the performance gains achieved by online and offline applications when we increase the scale of resource containers in an isolated cluster, and then explore the performance impact of scheduling conflict in a shared cluster (i.e., with distributed schedulers).

Figure 5.2 illustrates the average makespan by analyzing the impact of increasing the scale of resource containers on two representative job applications, one being offline and the other online. The offline job is a Spark application running a mix of 17 machine learning and graph algorithms, such as Support Vector

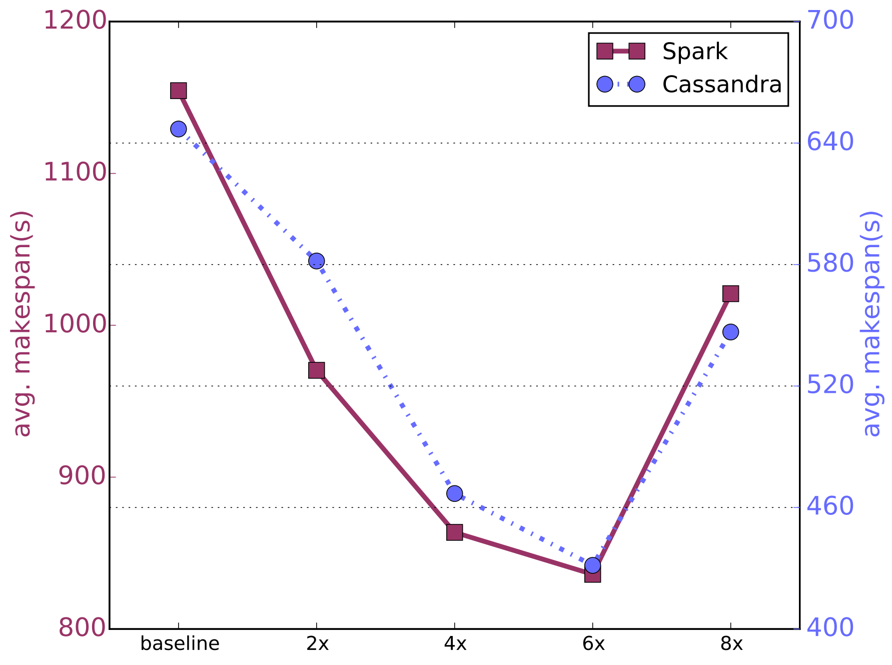


Figure 5.3: The impact of increasing resource scale on performance in the shared cluster

Machine, Matrix Factorization, K-means and Page Rank from SparkBench, on a small dataset [21], while the online job is a Cassandra service performing a mix of write and read requests [30]. For Spark, we report the average makespan by increasing the number of resource containers (each with 2 CPU cores and 6G memory) from 1 to 12, with increment of 2. We also record the mean makespan when increasing the number of resource containers in the same fashion for running the Cassandra service. The red line in Figure 5.2 illustrates the performance trend of Spark. It shows that the makespan is improved by up to 45% and the maximum performance is reached with the configuration of 8 resource containers. The makespan then remains stable when even more resource containers are used. The Cassandra service (the blue line in the figure) manifests a similar trend, i.e., the performance improves as the number of resource containers increases, but remains stable after the number of resource containers reaches 8.

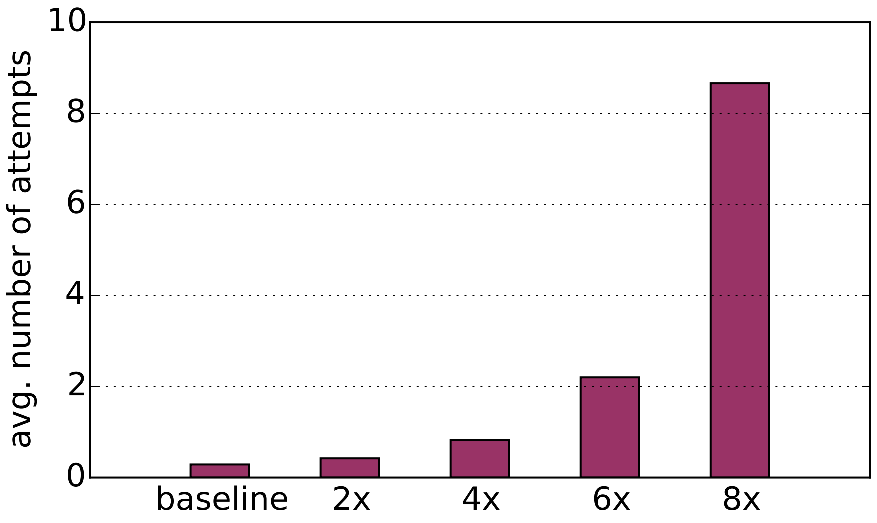


Figure 5.4: The average number of scheduling attempts

In the shared cluster, distributed schedulers compete with each other. They all have a copy of global state of the cluster. In this architecture, the performance trend becomes completely different, as in Figure 5.3. In the experiments, the two schedulers for Spark and Cassandra jobs share a cluster with 50 resource containers and they schedule their jobs at the same time. Each scheduler makes its decision independently and selects the required number of resource containers from its copy of cluster state and schedule the job to these resource containers. If two schedulers select the same resource container, the scheduling conflict occurs and the target resource container will reject one of the two jobs. As the result, the rejected job will be rescheduled and the other job can run in the resource container. In Figure 5.3, the red line depicts the average makespan of Spark jobs. It can be observed that the jobs still benefit from the increase of resource containers at the early stage, but the makespan starts to increase when the number of resource containers is more than 6. This is because more resource containers are requested by the schedulers, there is the higher possibility of scheduling conflict. Consequently, the number of scheduling attempts increases.

Figure 5.4 shows the average number of scheduling attempts by both schedulers as the schedulers demand more resource containers. Each scheduling attempt will incur the extra time, which includes the time spent in machine selection, network delay and task initialization. The machine selection for each Spark job is a random selection, taking the time between 2 and 4 seconds, whilst the Cassandra job adopts a heuristic algorithm for selection, which consumes between 15 and 25 seconds. Therefore, we can see that once the number of scheduling attempts are big, the scheduling overhead will surpass the performance benefit gained by increasing the number of resource containers and therefore impair the performance significantly.

Based on the above benchmarking experiments, we can conclude that the job performance can improve when we increase the number of resource containers. In a shared cluster, however, the benefit will be cancelled if too many resource containers are requested by distributed schedulers, due to the rescheduling overhead. Therefore, it is necessary to propose rational scheduling strategies in the cloud with distributed schedulers, in which the schedulers are aware of the resource competition and do not greedily request the resource containers at the maximum level.

5.1.3 Game Strategy: Ideas and Challenges

A shared cluster contains multiple autonomous schedulers, which compete for shared but limited resources and have the incentive to request more resource containers to increase its QoS as we discussed in Section 5.1.2. As such, the scheduling scenario in a shared cluster is best modelled as an *non-cooperative* game among rational and strategic players (schedulers). The players are rational because they want to maximize their own gain. They are strategic because they can choose their strategies(i.e., scheduling decisions) that influence other players.

In the game theory, the payoff of a player depends not only on his own strategy, but also on another player's strategy. A popular way of characterizing this dynamics is through *Nash Equilibrium (N.E)*. The payoff of one player is dependent on the choice of strategies. A player might decide to unilaterally switch his strategy to improve his payoff. This switch in the strategy will affect other players by changing their payoff. Therefore, other players might decide to shift their strategies as well, which in turn affects the player who originates the change of strategy. The collection of players is regarded as being at the N.E if no player can improve his payoff by unilaterally switching his strategy.

Pure & Mixed Strategy

Although each player has a set of available strategies to choose, sometimes the player will only choose one of the strategies as it has the maximum payoff, which is called the *pure strategy* in game theory. However, the agreement on pure strategy among players is not always guaranteed. In this situation, the players can select a strategy by randomizing over the set of pure strategies based on a certain probability distribution. This is called the *mixed strategy*, in which the N.E is guaranteed. We define the set of mixed strategies for player i to be $S_i = \prod(A_i)$. Then, the set of mixed strategy profile is simply the Cartesian product of the individual mixed strategy set, $\{S_1 \times \cdots \times S_n\}$. $s_i(a_i)$ denotes the probability that a pure strategy will be selected under the mixed strategy s_i. The set of the pure strategies that are assigned positive probability forms the mixed strategy s_i, which is called the *support* of s_i. Namely, the support of a mixed strategy s_i is the set of pure strategies $\{a_i | s_i(a_i) \geq 0 \text{ and } \sum s_i(a_i) = 1\}$.

Expected Utility Payoff

Due to the randomness of mixed strategy, we use the idea of expected utility from decision theory to represent the payoff of a mixed strategy. For a given game, $G(k, \vec{A}, \vec{u})$, where k is the number of players in this game, $\vec{A}$ is the vector of each player's strategies set, and $\vec{u}$ is the vector of expected utility cost for each player with mixed strategy profile $\{s_i = (s_1, \cdots, s_n)\}$. Therefore, the expected utility for player i with the mixed strategy a_i is defined in Eq. 5.1.

$$u_i(s_i) = \sum_{a_i \in A_i} u_i(a_i) \prod_{j=1}^{n} s_j(a_j) \tag{5.1}$$

Note that a pure strategy is a special case of a mixed strategy, because it assigned one specific strategy with probability one and others with probability zero. The expected utility payoff of pure strategy can be treated in the same way. Therefore, we focus on getting the mixed strategy as our strategy profile in this work.

Strategy Profile with Nash Equilibrium

Now we will look at the game from the perspective of an individual player instead of the outside supervisor. The purpose of each player is to maximize his expected payoff. This expected payoff not only depends on the strategy chosen by himself, but also on the strategy chosen by his competitors. Thus, this player would know how to choose his best response if he knows the strategies that his competitors are going to play. Specifically, player i's best response to s_{-i}, his competitors' strategy profile, is a strategy profile $s_i^* \in s_i$, and the expected off $u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i})$.

Because none of the players can know what strategies his competitors would adopt, it is not practical to deliver the best response for the players. However, we can leverage the idea of best response to define the most important concept in the non-cooperative game, $N.E$, which is a strategy profile $\{s = (s_1^*, \cdots, s_k^*)\}$ if and only if all player play his best response to others. For example, a given two-player(p_1 and p_2) game, each player has two strategies with his own pay-off matrix A or B, respectively. The pair of mixed strategies (p, q), one for player p_1 and one for player p_2, respectively, is a N.E if all other mixed strategies p' for player p_1 will be $p' \cdot A \cdot q \leq p \cdot A \cdot q$ and for all other mixed strategies q' for player p_2 will be $p \cdot B \cdot q' \leq p \cdot B \cdot q$. Two equations indicate those two players cannot improve its payoffs by switching their mixed strategy from p, q to any other mixed strategy p' or q'.

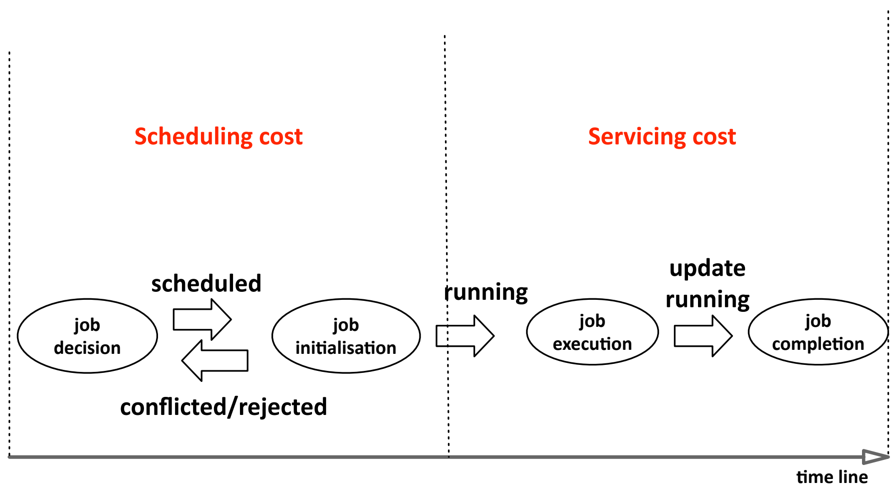


Figure 5.5: The life cycle of submitted job in the shared cluster

Therefore, the N.E is a stable strategy profile we want to achieve in this strategical scenario as no player would want to alter his strategy and all of the players play the best response to against others' strategy.

5.2 Design Issues in GRACES

To enable GRACES to schedule jobs with the awareness of the scheduling conflicts, we need to first understand the performance penalty caused by the conflict. We break the life cycle of a submitted job into two stages as in the Figure 5.5: scheduling stage and servicing stage. The length of the servicing stage is the execution time of the job. The scheduling stage is defined as the period between the time when the scheduler begins to start the first scheduling attempt for the job and the time when the resource containers that the job is running in the scheduled physical machine.

More specifically, the scheduling stage consists of two parts: job decision and job initialization. Job decision is the time spent by a scheduler in making scheduling decisions (i.e., determining which physical machines are allocated to run the resource containers requested by the job). Job initiation is the time spent in transferring the job and related packages to the allocated physical machines (so that the job starts running in the resource containers of the physical machine). However, during this stage, if different schedulers attempt to initiate their jobs on the same machine, the conflict occurs and only one job can be actually initiated and other jobs will be rejected and undergo rescheduling.

5.2.1 Scheduling Cost

The cost in the scheduling stage is defined as the *scheduling cost*, which is the time duration between the first scheduling attempt and the time when this job starts to run in the allocated resource containers. In the shared cluster, a scheduling attempt by a scheduler or the job initialization on the target machines may experience a series of conflict, failure and rescheduling until this job starts execution successfully. The time spent in job decision and initialization is the unavoidable, required time cost for a job to run on the shared cluster, which is denoted by J_{req}. The increasing number of rescheduling will significantly affect the job's execution time due to the accumulation of J_{req}. The scheduling cost for a job can be calculated by Eq. 5.2, where J_{sched} is the scheduling cost and E_{sched} is the expected number of scheduling attempts before it succeeds. The key problem thus comes down to determining the expected number of scheduling attempts, E_{sched}. Theorem 5.2.1 states the method to calculate E_{sched} and proves the correctness of the method.

$$J_{sched} = E_{sched} \cdot J_{req} \tag{5.2}$$

Theorem 5.2.1 *The expected number of scheduling attempts is* $E_{sched} = \frac{1}{\mathcal{P}_{wo}}$, *where* $\mathcal{P}_{wo}$ *is the probability that a scheduling attempt does not experience failure.*

Proof Assume k is the number of scheduling attempts, and $\mathcal{P}_i$ is the probability that the scheduler makes exactly k scheduling attempts for the job before it succeeds, which can be calculated by the $\mathcal{P}_{wo} \cdot (1 - \mathcal{P}_{wo})^{k-1}$. Eq. 5.3 gives the steps for deriving the expected number scheduling attempts.

$$\begin{aligned} E_{sched} &= \sum_{k=1}^{\infty} k \cdot \mathcal{P}_i(X=k) \\ &= \sum_{k=1}^{\infty} k \cdot \mathcal{P}_{wo}(1-\mathcal{P}_{wo})^{k-1} \\ &= \frac{\mathcal{P}_{wo}}{(1-\mathcal{P}_{wo})} \sum_{k=1}^{\infty} k \cdot (1-\mathcal{P}_{wo})^{k} \\ &= \frac{\mathcal{P}_{wo}}{1-\mathcal{P}_{wo}} \cdot \frac{1-\mathcal{P}_{wo}}{(1-(1-\mathcal{P}_{wo})^2)} \\ &= \frac{1}{\mathcal{P}_{wo}} \end{aligned} \tag{5.3}$$

Next, we derive $\mathcal{P}_{wo}$ for both lenient and strict allocations.

Lenient Allocation

Before we start to calculate the probability of a scheduling attempt without failure, we use Eq. 5.4 to derive the probability of scheduling attempts being conflicted and rejected, $\mathcal{P}_c$ is the probability of a scheduling attempt will be conflicted with others, w_i is a predefined weight which ranges from 0 to 1 as a common agreement by distributed schedulers to decide whether a job will be rejected or not when the conflict is happened. The higher the value of weights, the higher chance it will be rejected when the conflict is happened [95]. Thus, to get $\mathcal{P}_{wo}$, we need to confirm $\mathcal{P}_c$ by the requesting demands.

$$\mathcal{P}_{wo} = 1 - \mathcal{P}_c \cdot w_i \tag{5.4}$$

Since a lenient allocation will be accepted if there is no oversubscribed allocation on the target machine, the conflicted probability of one scheduling allocation is the number of selections that are scheduled allocation from schedulers that divides the number of oversubscribed allocation. For the number of selections from schedulers that is the product of each scheduler's allocation for its resource containers from the entire cluster. We denote it as $\prod^{\mathbb{S}} \mathbb{M}^{R_{si}}$, where $\mathbb{S}$ is the number of schedulers, $\mathbb{M}$ is the total number of resource containers within the cluster, and R_{si} is the number of resource containers required by scheduler i.

Algorithm 4 Pseudocode for enumerating the number of oversubscribed scheduled allocations.

```
1:  $T$ : the number of machine configuration types.
2:  $\mathbb{A}$ : the number of conflicted allocations.
3:  for $i := 0 \rightarrow$ T do
4:      $R_i$ : the number of resource containers in $T_i$
5:      $\mathbb{A}'_{cf}$ : the conflicted allocations on $T_i$.
6:      $T_i^{num}$ : : the number of machines $T_i$ has
7:      for $R_{s_0} := \max(R_{s_0}) \rightarrow 1$ do
8:          $\cdots \cdots$
9:          for $R_{s_j} := \max(R_{s_j}) \rightarrow 0$ do
10:             $\cdots \cdots$
11:             for $R_{s_{(\mathbb{S}-1)}} := \max(R_{s_{(\mathbb{S}-1)}}) \rightarrow 0$ do
12:                 $\mathbb{A}_i$ is an allocation for schedulers on $T_i$.
13:                 for $k := 0 \rightarrow \mathbb{S} - 1$ do
14:                     Append $R_{k_s}$ on $\mathbb{A}_i$
15:                 end for
16:                 if size($\mathbb{A}_i$) > size($T_i$) then
17:                     Append $\mathbb{A}_i$ on $\mathbb{A}'_{cf}$
18:                 end if
19:             end for
20:         end for
21:     end for
22:     $\mathbb{A} \mathrel{+}= \binom{T_i^{num}}{1} \cdot \sum_{k}^{\mathbb{A}'_{cf}} \prod_{s}^{\mathbb{S}} \binom{R_i}{R_{k|s|j}}$
23: end for
```

For the number of oversubscribed allocation, we design Algorithm 4 to enumerate all allocations on the target machine, then we calculate and sum those combinations to get the total number of oversubscribed allocations. Because this algorithm aims at the oversubscribed allocation on the machine capacity, the total number of oversubscribed allocation will be the sum of oversubscribed allocation on each type of machine configurations. In the first loop, we start from the number of machine configuration types within cluster T, at line 3. For each type of machine configurations, T_i, we set the number of available resource containers as R_i, and T_i^{num} as the number of machines that the machine configuration T_i has in this cluster. Then, we iterate each scheduler's allocation that can have the maximum number of resource containers till zero resource container from R_i so that we can enumerate schedulers' allocations(line 6-12). At line 13 and 14, we form each scheduler's allocation(R_{k_s}) from the above as $\mathbb{A}_i$, which is one allocation for schedulers on T_i. Consequently, if this allocation has oversubscribed the size of T_i, it will be a conflicted allocation, and we append this allocation into $\mathbb{A}'_{cf}$ for calculating the number of combinations. After we have enumerated all conflicting allocations, we calculate the number of combinations for this conflicting allocation as $\binom{T_i^{num}}{1} \cdot \prod_{k=0}^{\mathbb{S}-1} \binom{R_i}{R_{k_s}}$, which is the number of combinations from T_i^{num} to select one machine, and times the product of each scheduler's combinations of resource container from this machine. In the end, we sum all combinations for each machines configuration to get the total number of conflicted allocations(line 17). For example, a cluster has 1 machine configuration with 3 identical machines, and each machine can be assigned with 4 resource containers. One scheduler, $\mathbb{S}_1$, requests 3 resource containers, and the another, $\mathbb{S}_2$, requests 4 resource containers. Then, Algorithm 4 enumerates all conflicted allocations, $3\mathbb{S}_1 + 4\mathbb{S}_2$, $3\mathbb{S}_1 + 3\mathbb{S}_2$, $\cdots$, and $1\mathbb{S}_1 + 4\mathbb{S}_2$, which we denote it as the result of $\mathbb{A}'_{cf}$ from line 16. Then, the number of combinations that is conflicted allocations will be the sum of each combinations of conflicted allocations, which is $\binom{3}{1} \cdot \binom{4}{3} \cdot \binom{4}{4} + \binom{3}{1} \cdot \binom{4}{3} \cdot \binom{4}{3} + \cdots + \binom{3}{1} \cdot \binom{4}{1} \cdot \binom{4}{4} = 234$. On the other hand, the number of selection on the scheduled allocation for schedulers, $\prod^{\mathbb{S}} \mathbb{M}^{R_{si}}$, is $3^3 \cdot 3^4 = 2187$. The conflicted probability for those two schedulers will be $\frac{234}{2187} = 0.11$, and if we set two schedulers with equal chance to be rejected if two of them have the conflict, $\mathcal{P}_{wo}$ by $1 - 0.11 \cdot 0.5 = 0.945$ for Theorem 5.2.1 to calculate the expected number of scheduling attempts as $\frac{1}{0.945} \approx 1.06$, which is close to our simulation result.

From the above, we can derive the complexity of Algorithm 4 as $|T| \cdot |\mathbb{S}| \cdot |R|$, where $|T|$ is the number of machine configuration types, $|\mathbb{S}|$ is the number of schedulers and $|R|$ is the number of available resource containers.

Strict Allocation

For the strict allocation, the scheduler specifies the particular resource containers for the allocation, and this allocation will be regarded as the conflict if the specified resource containers have also been requested by others. Conflict exists among resource containers rather than machines.

$$\mathcal{P}_{wo} = \frac{\binom{\mathbb{M} - \mathbb{E}(R_c) \cdot w_i}{R}}{\binom{\mathbb{M}}{R}} \tag{5.5}$$

To get the probability of one strict allocation without failure, we use Eq. 5.5 where the combinations of requested resource containers from the non-failed resource containers that divide the the combinations of requested resource containers and the total number of resource containers within the cluster, and we denote the non-failed resources containers as the expected number of conflicted resource containers multiplies its weight, $\mathbb{E}(R_c) \cdot w_i$.

$$\begin{aligned} E(R_c) &= \sum_{i=0}^{\sum(\vec{R}) - \max(\vec{R})} \cdot \vec{R}_i \cdot \mathcal{P}_i \\ &= \mathbb{E}_1 + \sum_{s_i=1}^{\mathbb{S}} \sum_{j=0}^{R_{s_i}} \cdot R_x \cdot \frac{\mathbb{E}_2}{\prod_{s_r=0}^{\mathbb{S}} \binom{\mathbb{M}}{R_{s_r}}} \end{aligned} \tag{5.6}$$

where:

$$R_x : \max(\vec{R}) \mathrel{+}= 1$$

$$\mathbb{E}_1 : \max(\vec{R}) \cdot \prod_{s_i=0}^{\mathbb{S}} \frac{\binom{\max(\vec{R})}{R_{s_i}}}{\binom{\mathbb{M}}{R_{s_i}}}$$

$$\mathbb{E}_2 : \binom{\max(\vec{R})}{R_{s_i} - j} \cdot \binom{\mathbb{M} - \sum^{s_i} R_{s_i}}{j} \cdot \prod_{s_k}^{\mathbb{S}-2} \binom{\max(\vec{R})}{R_{s_k}}$$

It is straightforward that we can directly use Eq. 5.5 if there are only two schedulers compete each other. But the situation will be more complicated if there are multiple schedulers, the scheduler needs to confirm the expected number of conflicted resource containers from others. In strict allocation, this expected value ranges from the least conflict, which is the maximum number of resource containers from one of scheduled allocations as all of those allocations have conflicted with others, by the increment of one resource container, to the maximum conflict, which is sum of resource containers from all scheduled allocations as none of those allocations have conflicted with others at all. Then, we use Eq. 5.6 to calculate the expected number of conflicted resource containers, where we denote the number of conflicted resource containers as a random variable with the probability p_i. Theorem. 5.2.2 proves its correctness.

Theorem 5.2.2 *Given the number of resource containers per scheduler requesting, the expected number of conflicted resource containers can be derived by Eq. 5.6, where* $\mathbb{M}$ *is the total number of resource containers in the cluster, and* $\mathbb{S}$ *is the number of schedulers.*

Proof As the expected number of conflicted resource containers is the sum of each random variable multiplied by its probability from the least to the maximum, we firstly calculate the least case, $\mathbb{E}_1$. Then, the least number of conflicted resource container, $\max(\vec{R_i})$, multiplies with its probability, where the probability is the product of combinations of each scheduler requesting from the maximum resource containers, which we denote as $\prod\limits^{\mathbb{S}} \binom{\max(\vec{R})}{R_{s_i}}$, and it divides the product of combinations of the requesting resource containers by each scheduler and the total number of resource containers within the cluster, which is represented as $\prod\limits^{\mathbb{S}} \binom{\mathbb{M}}{R_{s_i}}$.

For the rest of sum, it is the sum of one increment on the maximum resource containers allocation to the sum of all allocations, and the number of sum iteration is equal to the number of schedulers and the number of resource containers by each scheduler requesting. Therefore, we denote it as $\sum\limits_{s_i=1}^{\mathbb{S}} \sum\limits_{j=0}^{R_{s|i}} \cdot R_x$, where we use R_x to replace $\max(\vec{R}) \mathrel{+}= 1$ as for the increasing increment on the random variable. The probability for each one of random variables is the product of combinations with each particular number of conflicted resource containers happening that divides the product of all combinations by schedulers as the denominator in $\mathbb{E}_1$, the numerator thus is the product of combinations of one of schedulers gradually decreasing his choices from the one with maximum resource containers, and the combinations of this scheduler incrementally choosing non-conflicted resource containers from others with an additional product of combinations that others still choose resource containers from the maximum resource containers by one particular scheduler, which we denote it as $\mathbb{E}_2$.

Therefore, the expected number of conflicted resources containers can be derived by Eq. 5.6 based on each resource containers per scheduler requesting. For example, the previous exemplar cluster has 3 identical machines, each machine can have 4 resource containers. One scheduler, $\mathbb{S}_1$, requests 3 resource containers, and another scheduler, $\mathbb{S}_2$, requests 4 resource containers. Thus, the possible conflicted resource containers will be $4, 5, 6, 7$. We then use Eq. 5.6, the expected number of conflicted resource containers will be $4 \cdot \frac{\binom{12}{4} \cdot \binom{4}{3}}{\binom{12}{3} \cdot \binom{12}{4}} + 5 \cdot \frac{\binom{12}{4} \cdot \binom{4}{2} \cdot \binom{12-4}{1}}{\binom{12}{3} \cdot \binom{12}{4}} + 6 \cdot \frac{\binom{12}{4} \cdot \binom{4}{1} \cdot \binom{12-4}{2}}{\binom{12}{3} \cdot \binom{12}{4}} + 7 \cdot \frac{\binom{12}{4} \cdot \binom{12-4}{3} \cdot}{\binom{12}{3} \cdot \binom{12}{4}} = 6.0$. Moreover, for one scheduler to request one resource container from this cluster competing with those two schedulers, combined with Eq. 5.5 and all schedulers have the equal chance to be rejected, we can derive $\mathcal{P}_{wo} = \frac{\binom{12-6 \cdot 0.5}{1}}{\binom{12}{1}} = 0.75$. The expected number of scheduling attempt will be $\frac{1}{0.75} \approx 1.3$, which is also close to our simulation result on strict allocation.

5.2.2 Servicing Cost

The second key metric that we define it as the *servicing cost*, which we use it to measure the cost whilst the job is running with the assigned resource containers. A running job with too few resource containers during its duration time is regarded as the *servicing cost*. Let R_{tar} be the target of required resource containers for a submitted job, which is set by the users at the job submission to achieve its ideal performance. We then note R_{asg} as the assigned resource containers for a scheduler choosing to run this job. Hence, comparing R_{asg} against R_{tar} that allows us to judge the quality of a running job as in Eq. 5.7, where J_{serc} is the job's *servicing cost* and J_d is the job duration.

$$J_{serc} = \frac{R_{tar}}{R_{asg}} \times J_d \tag{5.7}$$

If J_{serc} is equal to J_d, we have an ideal assignment that the job running with ideal resource containers. Otherwise, the larger gap between resource containers and job requirement, the higher J_{serc} will incur. Then, a scheduler uses Eq. 5.7 to gauge the *serving cost* when it is choosing the resource containers for the job.

5.2.3 Game-theoretic Solution for Shared Schedulers

To make rational and strategical decision on jobs, the scheduler needs the utility function to represent the payoff gains by choosing a strategy. The payoff for one scheduler to choose a strategy is dependent on others. Each scheduler makes his best response to others. Thus, the utility function is applied for the scheduler to quantify whether a strategy choice is the best response or not.

Utility Function

The utility function indicates player's utility gain based on the choice of strategies, and the strategies set of job scheduling is about a particular scale of resource containers from the baseline to the maximum, which we denote as $R_{\alpha i}$, the R_α scale of resource containers for player i to request. Based on Theorem. 5.2.1, we can deduce the expected number of scheduling attempts with the scale of R_α resource containers. Put the scheduling cost together, we can have the utility function for player i requesting scale of $R_{\alpha i}$ resource containers as in Eq. 5.8.

$$J_{cost} = E_{sched} \cdot J_{req} + J_{serc} \tag{5.8}$$

In addition, as this utility function results the performance cost, whilst the utility function within the game theory represents the payoff benefit a player can gain, we multiply a "-1" on Eq. 5.8 as the payoff benefit a scheduler can gain. In other words, the smaller performance cost a strategy can have, the better payoff benefit a scheduler can get.

N.E for the Participating Players

Since we represent each distributed cluster scheduler in the cluster as a player with the strategies set, this scenario has been formalized as a multi-player game among distributed schedulers. For a cluster with k schedulers, there are $k \cdot (A_1 \times A_2 \times \cdots \times A_k)$ matrices, where A_i is the payoff matrix for a scheduler based on its strategies set. Each player chooses $(x_1, x_2, \cdots, x_i)$ to cover his strategies set that can equalize his opponents' expected payoff on their strategies set, where x_i is the probability value ranges between 0 to 1. With another condition that the sum of this probability distribution is equal to 1, we can combine these linear functions and use the linear function solver to calculate the mixed strategy profile for each player.

We use an implementation of Gambit [80] library as our linear equations solver to get the N.E solution, and the time complexity is $O(n^{\log \frac{n}{\epsilon}})$ [29], where n is the number of players and ϵ is the number of strategies at most where the NE strategy profile will be. When all players prefer not to switch or is indifferent between his strategies, the set of strategies is a N.E.

Table 5.2: An exemplar three-players' payoff matrix

		1	2	3
	1	a_{111}, b_{111}, c_{111}	a_{112}, b_{112}, c_{112}	a_{113}, b_{113}, c_{113}
1	2	a_{121}, b_{121}, c_{121}	a_{122}, b_{122}, c_{122}	a_{123}, b_{123}, c_{123}
	3	a_{131}, b_{131}, c_{131}	a_{132}, b_{132}, c_{132}	a_{133}, b_{133}, c_{133}
	1	a_{211}, b_{211}, c_{211}	a_{211}, b_{211}, c_{211}	a_{213}, b_{213}, c_{213}
2	2	a_{221}, b_{221}, c_{221}	a_{222}, b_{222}, c_{222}	a_{223}, b_{223}, c_{223}
	3	a_{231}, b_{231}, c_{231}	a_{232}, b_{232}, c_{232}	a_{233}, b_{233}, c_{233}
	1	a_{311}, b_{311}, c_{311}	a_{312}, b_{312}, c_{312}	a_{313}, b_{313}, c_{313}
3	2	a_{321}, b_{321}, c_{321}	a_{322}, b_{322}, c_{322}	a_{323}, b_{323}, c_{323}
	3	a_{331}, b_{331}, c_{331}	a_{332}, b_{332}, c_{332}	a_{333}, b_{333}, c_{333}

The approach of a strategical game is firstly to construct the payoff matrix that records players with strategies, and its related payoffs based on the choice of strategies. Consequently, for a game with k players that needs $k(A_1 \times A_2 \times \cdots \times A_k)$ matrices. Each player then chooses one probability distribution, which is the mixed strategy with the N.E, overs his strategies set that has more or equal benefit gains than other probability distributions. And since the mixed strategy is the probability distribution, the value ranges between 0 to 1 and the sum of this probability distribution is equal 1. After combined those linear functions, we can use linear function solver to calculate the mixed strategy profile for each player.

Taking a three-players game as an example, each player has three strategies to choose against others, the payoff for each player we denote it as a, b and c, respectively. Thus, this example of three-player game will be specified by three $3 \times 3 \times 3$ matrices as in the Table 5.2.

When player a chooses his strategy s_1, the payoffs are

$$\begin{array}{cc} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} a_{111}, b_{111}, c_{111} & a_{112}, b_{112}, c_{112} & a_{113}, b_{113}, c_{113} \\ a_{121}, b_{121}, c_{121} & a_{122}, b_{122}, c_{122} & a_{123}, b_{123}, c_{123} \\ a_{131}, b_{131}, c_{131} & a_{132}, b_{132}, c_{132} & a_{133}, b_{133}, c_{133} \end{bmatrix} \end{array}$$

, where the (i, j)-th entry is the triple payoffs for player a, b and c choosing the strategy profile (s_1, s_i, s_j).

As we indicated in the Section 5.1.3, the N.E strategy profile is where each player chooses his best response to against others. A mixed strategy profile is when each player plays the strategies with positive probabilities. Only if for any player keeping all the other players' strategies fixed, the payoffs to player i from each of player i's pure strategies are equal. Otherwise, player i could improve his own payoff by omitting those pure strategies leading to lesser payoffs. In other word, each player adopts one probability distribution overs his strategies to equalize the payoff of each strategy. Thus, these conditions give a list of polynomial equations, the unknowns in those equations are the probabilities assigned by each player to their pure strategies to form their mixed strategies. Only the solution with positive numbers represent the N.E mixed strategy profile, and the sum of each player's mixed strategy is 1.

Then, in this exemplar three-player game, u_1^a, as player a's expected payoff from choosing s_1 that has expressed in Eq. 5.9, where σ^b and σ^c are the probabilities for player b and c choosing their related strategies, respectively.

$$\begin{aligned} u_1^a = \; & a_{111} \cdot \sigma_1^b \cdot \sigma_1^c + a_{112} \cdot \sigma_1^b \cdot \sigma_2^c + a_{113} \cdot \sigma_1^b \cdot \sigma_3^c \\ & + a_{121} \cdot \sigma_2^b \cdot \sigma_1^c + a_{122} \cdot \sigma_2^b \cdot \sigma_2^c + a_{123} \cdot \sigma_2^b \cdot \sigma_3^c \\ & + a_{131} \cdot \sigma_3^b \cdot \sigma_1^c + a_{132} \cdot \sigma_3^b \cdot \sigma_2^c + a_{133} \cdot \sigma_3^b \cdot \sigma_3^c \end{aligned} \tag{5.9}$$

Player 1's payoff from choosing pure strategy s_2 and s_3 are given by Eq. 5.10 and Eq. 5.11.

$$u_2^a = a_{211} \cdot \sigma_1^b \cdot \sigma_1^c + \cdots + a_{233} \cdot \sigma_3^b \cdot \sigma_3^c \tag{5.10}$$

$$u_3^a = a_{311} \cdot \sigma_1^b \cdot \sigma_1^c + \cdots + a_{333} \cdot \sigma_3^b \cdot \sigma_3^c \tag{5.11}$$

Now, we can equalize the above three linear equations, and one additional equation that the sum of one mixed strategy probability distribution for each player as 1. Similarly, we can have linear equations and equalize relations for b and c. We use linear function solver to solve these linear equations to get each player's mixed strategy probability distribution. We use an implementation of Gambit [80] library as our linear equations solver to get the N.E solution, and the time complexity is $O(n^{\log \frac{n}{\epsilon}})$ [29], where n is the number of players and ϵ is the number of strategies at most where the NE strategy profile will be. When all players prefer not to switch or is indifferent between his strategies, the set of strategies is a N.E.

5.3 Performance Evaluation

In this section, we implement different strategies and our Nash solver within GRACES and compare their performance. Our experiments were carried out with both simulations and production AWS clusters.

Trace-driven simulator

To precisely evaluate GRACES with more parameter, we built a trace-driven simulator that performs scheduling on the scale of Google's production cluster. We use the publicly available Google trace [94, 109] that collects the detailed jobs and tasks information from a large Google cluster during a month period. This trace records the CPU and memory capacities for each machine within the cluster. The cluster is made of heterogeneous machines. All values given in this trace were normalised according to their maximum. Our simulation were constructed using the data from this trace. We list our cluster configuration in Table. 5.3. The default cluster in our simulation has 1000 machines. This leads to a 60% system utilizations which allows us to explore how the system performs at a high conflict cost.

In each simulation, once the cluster scheduler receives a job, a scheduling decision on the target machines is made with a scheduling cost, and any resource container conflicted with others from this scheduling decision that will be re-

Table 5.3: Cluster configuration

Percentage	CPUs	Memory
50%	0.5	0.5
30%	0.5	0.25
8%	0.5	0.76
6%	1	1
5%	0.5	0.3
1%	0.25	0.25

jected and reschedule based on the predefined weight. To keep the simplicity, we set each job has equal weight to be rejected and reschedule in experiments.

If the scheduled job has allocated successfully, it will run with assigned resource containers by a predefined job duration until it completes. The simulator then records the performance cost and the number of scheduling attempts during the experiments. In our experiments, we set each job's scheduling time as a random number between 30 and 90 seconds. This value is close to that of large production clusters [18, 107]. This time cost also includes all of the overheads occurred before the job starting running, such as the task initialization, data backup, network latency and security checking. In the experiments, we present the average results across multiple runs, all runs with small changes on the results.

Workloads

We extract the workload information from the Google trace and give the CDFs of memory and CPUs demands in Figure 5.6. As we can see from this trace, most tasks are small and requires less than 20% of a typical cluster machine's resources. We also adopt the distributions of the number of tasks per job and job duration from Section 5.1.1 to simulate a job's demands on resource containers, size of tasks, and job duration. To create our experimental workload, we retain the job requirements on resources demands. The jobs were however, not performance tolerant in terms of resource demands. Hence, we set the performance tolerance for the jobs randomly between $20 - 40\%$, which are based on real job

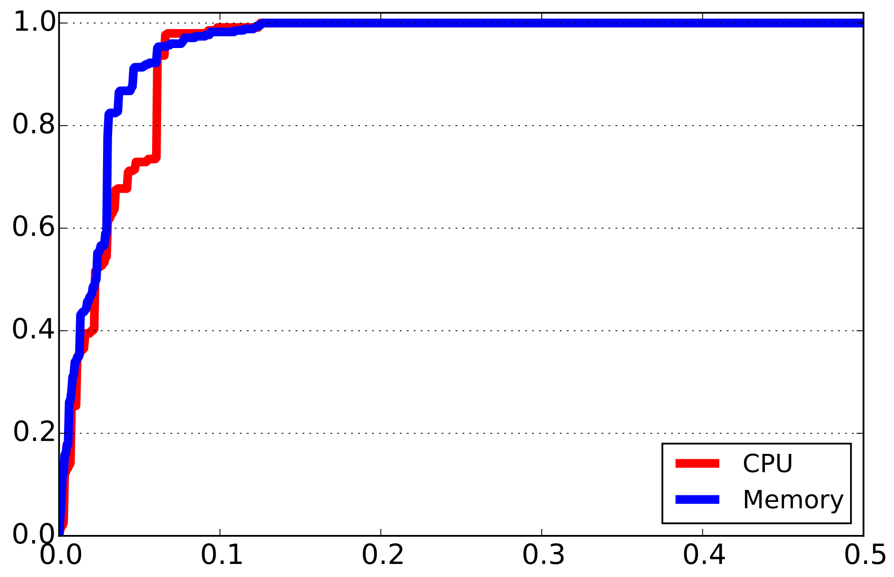


Figure 5.6: CDFs of memory and CPU for workloads

applications and productive scenarios [2, 6] on their the percentage of required demands or completed results. We apply it in the jobs for the strategies set. For instance, a job which requires 100 tasks and has 30% performance tolerance needs at least needs 70 of its 100 tasks to be scheduled. Its strategies set will range from 70 to 100.

Moreover, before the experiments, we partition jobs into groups. Jobs from the same group have the same scheduler. This partition is based on each job's requirement of resource containers, number of tasks and job duration. Then, schedulers concurrently make scheduling decisions based on its job requirements and the strategies set from others. This is consistent with our description regarding one scheduler is responsible for one particular type of jobs and they do not have detailed information on other scheduling decision except the general strategies set from the baseline to the maximum.

Metrics

Since both job scheduling and execution are affected by the choice of strategies, the qualities of both are important metrics to evaluate. We therefore compare the performance cost and the number of scheduling attempts for the scheduler with the Nash solver and other predefined strategies.

According to the assigned performance tolerance, we proportionally scale down the strategies on the resource containers assignment from the baseline to the maximum. The strategies we represent in this paper, which include baseline (BL), median-1 (M-1), median-2 (M-2), median-3 (M-3), max (MX), random (RM), and Nash (NS). NS is the best response strategy adopted by players to achieve the N.E. RM is the scheduler with equal probability to randomly choose any one of BL, M-1, M-2, M-3 and MX. For example, a job with 40% performance tolerance means the BL will be 60% of resource containers assignment compared to his initial job demand, 70% of resource containers assignment is the M-1, 80% is M-2, 90% is M-3, and 100% is the MX strategy to choose the initial job demands without being degraded.

In addition, for Section 5.3.2, we use the percentages of improvement on the performance cost as in Eq. 5.12, which compares the performance cost based on different strategies with the worst case where we set a job will be eventually accepted with its baseline resource containers assignment after it has been rejected over 10 times, which is the standard maximum times to be rejected in production clusters.

$$Improvement\% = \frac{Worst - Adopted}{Worst} \times 100\% \tag{5.12}$$

5.3.1 Comparing Strategies with Increasing Concurrent Jobs

In this experiment, we investigate how different strategies adopted by a scheduler performs with *concurrent and competitive* (CC) jobs, CC job are synthetic jobs that have identical demands to scheduled jobs, but each one of these tasks randomly occupies resource containers on a random machine. The number of tasks increases from 100 to 200 for both online and offline jobs, respectively. The scheduler adopts one of the strategies from BL to NS for each run to evaluate

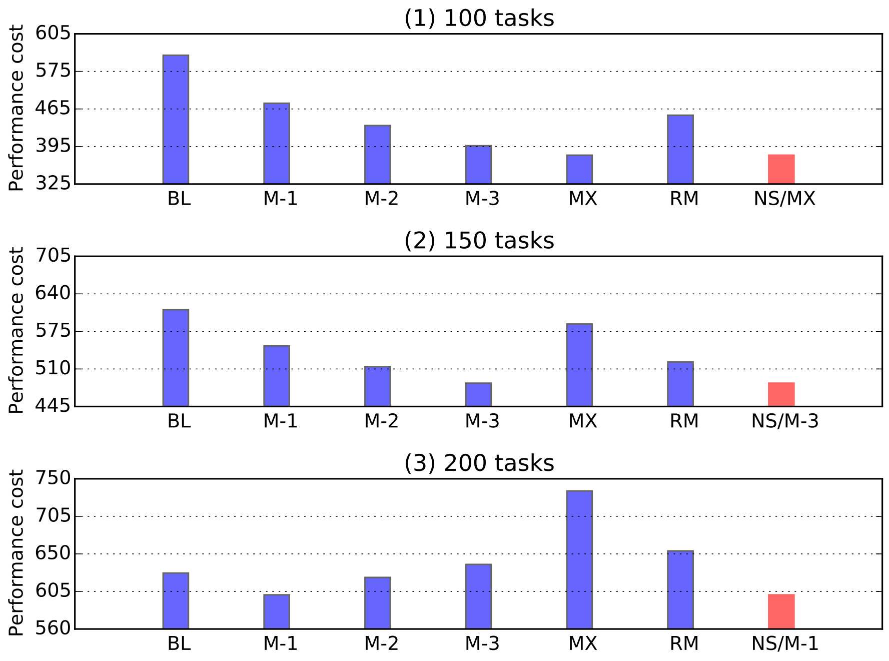


Figure 5.7: Performance cost on offline jobs

its performance cost.

Figure 5.7 and Figure 5.8 show that the NS strategy does result in reduction of performance cost on both online and offline jobs. The red bars from these two figures indicate the performance cost from the NS compared to the outcomes from other strategies that are denoted by the blue bars, NS performs consistently better than other strategies. When the CC job has 100 tasks, NS chooses MX as the NS to its best response for both online and offline jobs to minimise its minimum performance cost. We denote this choice under the red bars. Other strategies, especially for BL, schedule jobs with small resource containers that miss the benefit of reducing servicing cost by assigning more resource containers, and NS reduces the performance cost by 32% and 20% for both online and offline jobs. At 150 tasks run, MX starts to perform poorly on the reduction of performance cost because it increases the scheduling cost significantly compared to other strategies, and the advantage of MX on the servicing cost has been eliminated by the scheduling cost. For the CC job with 200 tasks and the system

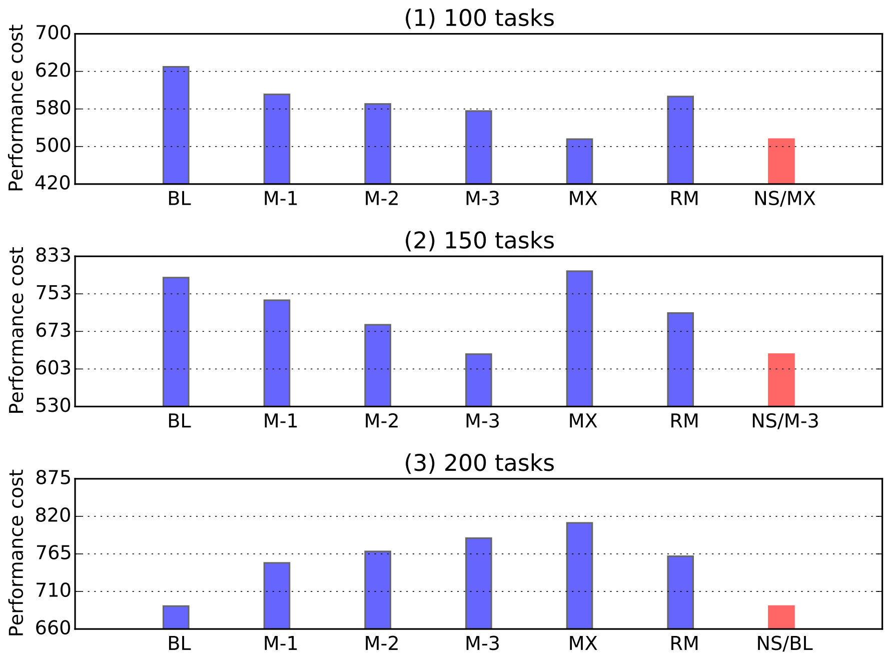


Figure 5.8: Performance cost on online jobs

load is high, NS reduces the most performance cost by choosing BL and M-1 for online and offline jobs, respectively, and strategies with high resource containers that have suffered the expensive performance cost as they have not adjusted to high load of cluster utilization by reducing their job resource demands.

Moreover, because the synthetic CC job does not have the performance tolerance, it always chooses one particular "strategy" (100, 150, and 200) in each round of experiments, NS thus always chooses one of his strategies as his best response to be the pure NE.

It is obvious that scheduler with online jobs has generally more performance cost than offline jobs, because it adopts the strict allocation that incurs more scheduling cost compared to the lenient allocation. We report the average number of scheduling attempts during the experiments in Figure 5.9. The number of scheduling attempts is expectedly increasing with the scheduler adopting more and more aggressive strategies. Although NS cannot guarantee the minimum number of scheduling attempts in each run, it has reduced the most perfor-

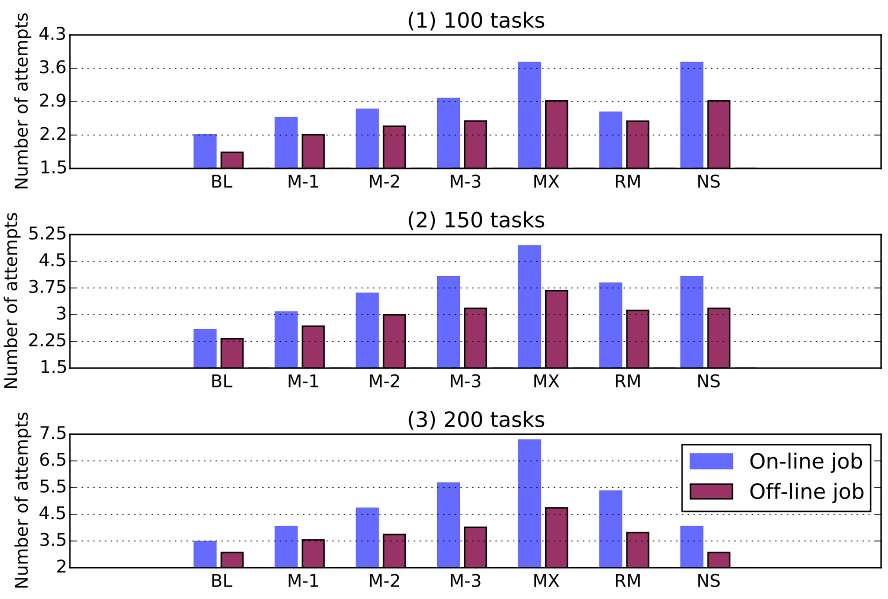


Figure 5.9: The average number of scheduling attempts

mance cost during the experiments, since NS has focused on the overall job performance rather the minimum number of scheduling attempts in terms of the strategy selection.

5.3.2 Comparing Strategies with Multiple Schedulers

This subsection analyses the experiment results on multiple schedulers with different strategies on the shared cluster. Schedulers schedule their jobs based on one of the strategies from BL to NS to measure the impact of strategy selection in each round of experiment. We evaluate four schedulers with the other two schedule online jobs and another two schedule offline jobs.

Figure 5.10(1) gives the average improvements of all schedulers with different strategies. The average improvement of schedulers with NS has outperformed other strategies, and the average improvement of schedulers with MX has the worst outcome, unsurprisingly. This is because the scheduling cost becomes too expensive and overweights the benefit of servicing cost with maximum resources when all schedulers adopt the MX.

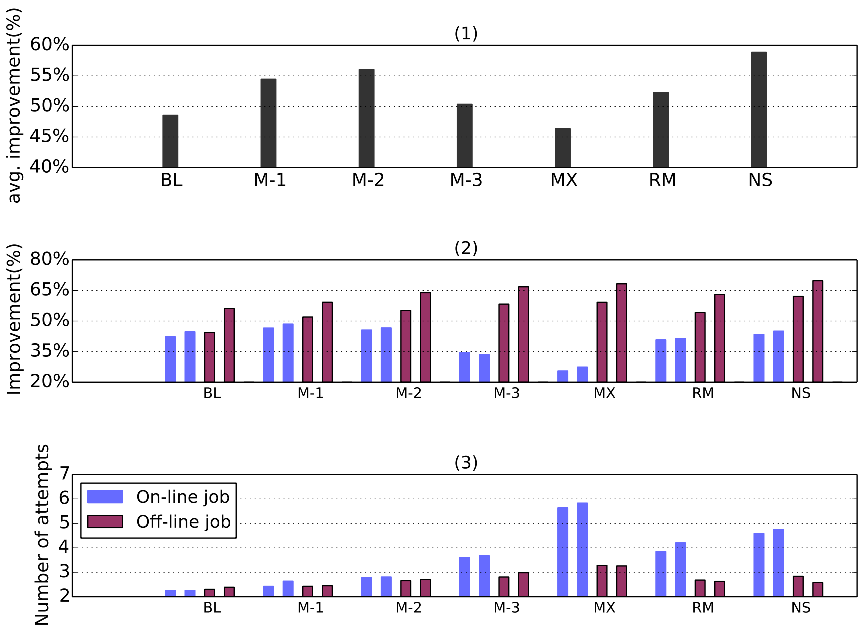


Figure 5.10: The improvement of 4 schedulers with different strategies

We then display the percentages of improvement for each scheduler in Figure 5.10(2). The percentages of improvement on schedulers for online jobs(red bars) are generally larger than the schedulers for offline jobs(blue bars) around 30%, as the online jobs are more sensitive than offline jobs in terms of performance tolerance and the longer job duration, and they will be jeopardized severely if the worst case is happened on them. The scheduler for online jobs thus will choose to tolerate more scheduling cost at the selection of NS, despite it has more decision time cost on reschedule than the scheduler for offline jobs. For the NS strategy, there is not a pure N.E solution that exists for this game. The strategies set for all schedulers could not achieve their best response against others in pure strategies. Then, N.S will use the mixed strategy that one probability distribution covers his strategies set to equalize the payoff of his pure strategies. In our N.S profile, one of offline schedulers only adopts his M-2 and M-3 strategies with probabilities as 27% and 73%, respectively, and 0% to other strategies. Another offline scheduler adopts his M-1 and M-2 strategies with 43% and 57%, respectively, whilst one online scheduler adopts his M-2 as 22% and M-3 as 78%, but

0% to other strategies, respectively, and another online scheduler chooses his M-3 strategy only. Due to the NS strategies adopted by all schedulers choosing their best response, all schedulers can achieve the stable and better performance than other strategies set.

In Figure 5.10(3), we illustrate the average number of scheduling attempts where the offline jobs have more scheduling attempts than online jobs. Since offline jobs require more number of resource containers than online jobs. Offline jobs have more re-scheduling attempts than online jobs. More importantly, the decision time cost of scheduler for offline jobs is smaller than the counterpart of schedulers for the online jobs. This means that the best response strategy for offline jobs can take more scheduling attempts with a reasonable range than online jobs. In this group of experiments, the schedulers for offline jobs generally have two times more than online jobs, in terms of the number of scheduling attempts. However, In spite of the NS strategies for both online and offline jobs that have achieved the maximum improvements in performance cost, NS cannot always guarantee the minimum number of scheduling attempts for the scheduling, because NS considers the trade-off between scheduling cost and performance cost for the jobs, which sustains an acceptable scheduling cost but achieving the performance gains as much as possible within the job tolerance degree.

5.3.3 Real Experiment on AWS

In this subsection we present results from our AWS testbed, which contains 50 EC2 instances, each instance has 2 CPU cores and 6G memory. One scheduler is for handling Spark and another scheduler is for the Cassandra, respectively. The same workload for both two applications from Section 5.1.2 are used here. To eliminate the task interference, any machine receives the scheduling requesting from a different scheduler will randomly reject one of them. The schedulers use our developed Nash model solver to choose its NS or predefined strategies, which range from 2 to 8 resource containers, to allocate its jobs. The inter-job arrival

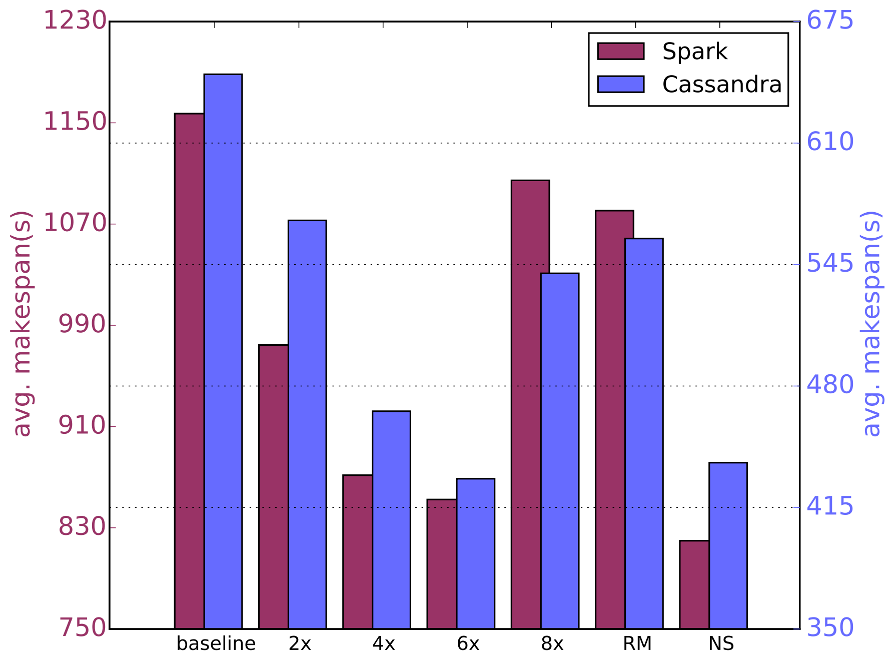


Figure 5.11: The average makespan of AWS cluster schedulers with different strategies

time is around two minutes in this experiment. Figure 5.11 illustrates the average makespan of Spark and Cassandra in red and blue bars, respectively. We can see the NS ones for Spark and Cassandra both can achieve better makespan outcomes than other strategies, where Spark chooses his 6x and 8x with 36% and 64% probability, and 42% and 58% probability for Cassandra deciding his 6x and 8x, respectively. RM performs terribly in this experiment, as the gap of those jobs performance amongst strategies is significant and its performance has been hurt deeply at low resource containers strategies. Moreover, Figure 5.12 indicates the average number of scheduling attempts for two schedulers, which increases with the strategy choosing more resource containers. NS has an acceptable number of scheduling attempts than others with the guaranteed job performance.

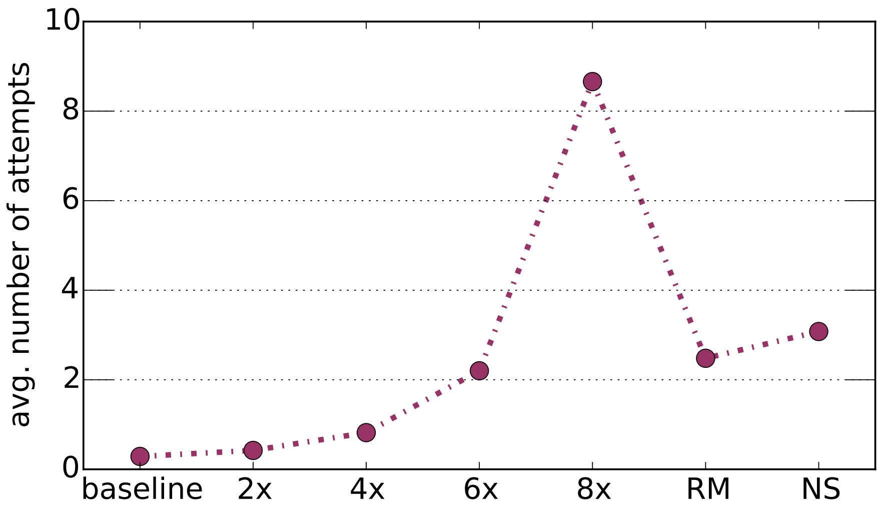


Figure 5.12: The average scheduling attempts of AWS cluster schedulers with different strategies

5.4 Summary

This chapter has investigated the related influence and improvements on job performance within the shared cluster environment, and we present GRACES, a game-theoretical framework with the awareness of performance target and resources competition for distributed cluster schedulers. GRACES is derived from validated analytical methods, such as Nash Equilibrium (N.E) in the game theory. It strategically adjusts scheduling policies on incoming jobs with respect to the performance target and other competitors. We have formalized the expected number of scheduling attempts and the performance cost for the distributed schedulers with different kind of jobs to guide the choice on the scheduler's scheduling policy. The performance evaluation of the GRACES uses both simulation with Google production workload and a real testbed in the AWS with typical offline application and online service. The experiments verify the effectiveness of the GRACES that is able to achieve the improved performance outcomes under the shared and conflicted cluster architecture. In the next chapter the contributions made in chapter 3, 4 and 5 are concluded, alongside a discussion and directions for further work to support this research.

CHAPTER 6
Conclusions and Future Work

The work presented in this thesis investigates the strategies to improve the performance in clouds. Since the development of cloud computing accelerates the evolution of traditional infrastructure, this brings the challenges to the industry and academic community in many aspects. Moving “everything” into the cloud has become the most popular trend for not only cloud tenants but also cloud providers. The cloud tenants need to accurately plan and implement their services and infrastructure in the cloud to maximize their service performance whilst reducing the unnecessary cost. On the other hand, it is a crucial issue for the cloud provider to deliver the required resources to its tenants with guaranteed QoS and minimize the cost. The existing approaches presented in chapter 2 indicate that there are still a lot of room for improvement with regards to the above-mentioned issues.

6.1 Conclusions

In this thesis, we develop a framework to determine the provision of computing resources required for cloud services to deliver the desired level of QoS. The frameworks capture the interaction relations among different services in the cloud. Next, we extend the above framework to model bandwidth provision so as to not only meet the external communication demand but also the internal communication relations. and evaluate it with simulations at the scalability of industrial level. The real world AWS cloud testbed with the framework has been implemented to verify its efficiency and effectiveness. After obtaining resource provision, we develop the VM-to-PM placement methods under the constraints on computing resources and communication cost.

Moreover, we propose the game-theoretical methodology for distributed job schedulers to exploit the trade-off between resource conflicts and resource demand. In this thesis, the parallel scheduling behaviour by distributed schedulers is modelled as a non-cooperative game and the Nash Equilibrium point is solved for the game, which represent the best scheduling behaviour of distributed schedulers.

Finally, we conduct the experiments to evaluate the effectiveness of the proposed methods for resource provision, VM placement and distributed scheduling. The experiments are carried out with simulated workload, the workload trace from production cloud and the real cloud testbed.

6.2 Discussion

By understanding the performance demand of cloud computing at different scales and angles, we have achieved three main goals: the resources provision under interactive and complex workflows, the resource allocation with various constraints and objectives, and job scheduling within the emerging cluster architecture.

Whilst the benefit of these goals is clear, there are certain limitations to our research. We discuss them in this section.

The first limitation of the research lies in the cloud-IO model that provisions the required resources for both external and internal demands. The IO model is used for determining the long term resource planning in economy, which means that if there is any fluctuation in the original service communication, the cloud-IO model would not be able to reflect this change unless its consumption matrix has been updated in time. Hence, we need an additional resource provision framework that can catch these changes and respond with new resources provision. One of potential solutions to this issue is the recommended system, as we mentioned in the Section 6.3. It can predict and recommend the suitable resources in time under the changing patterns amongst services. Since we have previous service communication records already, it is not difficult to make reasonably accurate prediction based on current techniques [74] in the recommend system.

Another limitation is related to the VM-to-PM allocation. Our two elaborated algorithms can handle a cloud with a large scale of services and VMs. However, with the increasing scale of a cloud, they still have the shortage of the slow convergence speed, compared to simpler greedy algorithms. Although the greedy algorithms cannot achieve as good communication cost and PM consumption as our algorithms, they can achieve acceptable results in some scenarios. Due to this advantage of speed, simple greedy algorithms will still be one of options to handle a very large scale of cloud [10]. Therefore, investigating the trade-off between speed and quality in different algorithms becomes an interesting research issue when we face different scales of cloud.

The final limitation is that our job scheduling focuses on reducing conflict cost amongst distributed cluster schedulers. Scheduling decisions on assigned resources are arguably most important to service jobs. However, there are a number of additional metrics that we do not consider, but are likely to be of interest to the maintainers of such systems, such as resource initialization, cluster reliability and the searching space of available machines for scheduling decisions. Many of these are linked to job performance, and could thus jeopardize the QoS if we failed to consider them carefully. Later in this chapter we present some directions for future work where these challenges are explored and investigated.

6.3 Directions for Future Work

Following on the work presented in this thesis, further work is planned in the following aspects.

• Cloud providers, such as Amazon, Google and Microsoft, have provided many different configurations and services for their tenants. For example, Amazon has 44 different EC2 configurations and over 100 services so far [4], each of which has the advantage in some particular scenarios. The number of possible configurations is expected to increase further as the market continues to increase. It is difficult for cloud tenants to get familiar with all configuration details and make correct selection of configurations for a given application scenario. The technique of recommended system [74], such as Matrix Factorization and Stochastic Gradient Descent, could be applied to build a recommended system to aid tenants' configuration selection and implementation planning.

- When a scheduler manages thousands of machines in a large-scale cloud, it is expensive for the scheduler to search all machines and find the optimal machine set when making scheduling decisions for each job. [18, 107] propose a method that randomly selects a subset of all machines. The searching algorithm is only applied to the subset of the machines. However, the chance of finding the most suitable machines is not guaranteed because of the randomness. Therefore, a more "intelligent" resource manager needs to be designed. It can borrow the ideas from the Information Retrieval regarding compression, clustering and ranking. This "intelligent" resource manager can not only increase the chance of finding the suitable machines, but also alleviate the stress for the schedulers.
- The latency of job initialization is an unavoidable price to be paid in Clouds. However, starting up the required resources before the job arrives could significantly reduce the latency and speed up the job execution. In order for this method to be effective, the starting time of future jobs need to be accurately predicted. The techniques in the time series analysis can be applied here to establish a predictable framework for job initialization.
- Achieving high availability is always an important objective in the cloud environment. Currently, academic community and industries highly rely on the Paxos algorithm [76] to guarantee the high availability. However, it is not always suitable and economical for the emerging cloud environment, especially for the massive scale of services and machines. We plan to investigate new techniques for achieving high availability in the massive scale of cloud systems in the future.

References

[1] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Reoptimizing data parallel computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 281–294, 2012.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[3] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, pages 31–40. ACM, 2012.

[4] Amazon. Aws products, 2015. URL `https://goo.gl/pPafiL`.

[5] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association, 2012.

[6] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. Grass: trimming stragglers in approximation analytics. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 289–302. USENIX Association, 2014.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[8] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 13–24. ACM, 2007.

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

[10] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. OShea. Chatty tenants and the cloud network sharing problem. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 171–184. USENIX Association, 2013.

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

[12] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *Micro, Ieee*, 23(2):22–28, 2003.

[13] S. Bazarbayev, M. Hiltunen, K. Joshi, W. H. Sanders, and R. Schlichting. Content-based scheduling of virtual machines (vms) in the cloud. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 93–101. IEEE, 2013.

[14] O. Beaumont, L. Eyraud-Dubois, C. Thraves Caro, and H. Rejeb. Heterogeneous resource allocation under degree constraints. *Parallel and Distributed Systems, IEEE Transactions on*, 24(5):926–937, 2013.

[15] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[16] A. Beloglazov and R. Buyya. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of

service constraints. *Parallel and Distributed Systems, IEEE Transactions on*, 24(7):1366–1379, 2013.

[17] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 431–442. ACM, 2012.

[18] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 285–300. USENIX Association, 2014.

[19] J. Bredin, R. T. Maheswaran, C. Imer, T. Başar, D. Kotz, and D. Rus. A game-theoretic formulation of multi-agent resource allocation. In *Proceedings of the fourth international conference on Autonomous agents*, pages 349–356. ACM, 2000.

[20] M. Cardosa, M. R. Korupolu, and A. Singh. Shares and utilities based power consolidation in virtualized server environments. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*, pages 327–334. IEEE, 2009.

[21] S. T. Center. Spark benchmark suite, 2015. URL `https://goo.gl/1toQT3`.

[22] S. Chaisiri, B.-S. Lee, and D. Niyato. Optimal virtual machine placement across multiple cloud providers. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pages 103–110. IEEE, 2009.

[23] K. Chard and K. Bubendorfer. High performance resource allocation strategies for computational economies. *Parallel and Distributed Systems, IEEE Transactions on*, 24(1):72–84, 2013.

[24] C. Chen, L. He, H. Chen, J. Sun, B. Gao, and S. A. Jarvis. Developing communication-aware service placement frameworks in the cloud economy. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[25] C. Chen, L. He, B. Gao, C. Chang, K. Li, and K. Li. Modelling and optimizing bandwidth provision for interacting cloud services. In *Service-Oriented Computing*, pages 305–315. Springer, 2015.

[26] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz. Analysis and lessons from a publicly available google cluster trace. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95*, 2010.

[27] A. Cockcroft. Data flow at netflix, 2011. URL `https://goo.gl/4hlmwm`.

[28] P. Costa, A. Donnelly, G. O'Shea, and A. Rowstron. Camcubeos: a key-based network stack for 3d torus cluster topologies. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 73–84. ACM, 2013.

[29] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou. The complexity of computing a nash equilibrium. *SIAM Journal on Computing*, 39(1): 195–259, 2009.

[30] DATASTAX. Apache Cassandra Stress Tool, 2015. URL `https://goo.gl/2NPTPS`.

[31] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[32] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[33] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[34] C. Delimitou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 41(1):77–88, 2013.

[35] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 127–144. ACM, 2014.

[36] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110. ACM, 2015.

[37] R. Duan, R. Prodan, and X. Li. Multi-objective game theoretic schedulingof bag-of-tasks workflows on hybrid clouds. *Cloud Computing, IEEE Transactions on*, 2(1):29–42, 2014.

[38] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. *ACM SIGCOMM Computer Communication Review*, 29(4):95–108, 1999.

[39] E. Feller, L. Rilling, and C. Morin. Energy-aware ant colony based workload placement in clouds. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 26–33. IEEE Computer Society, 2011.

[40] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112. ACM, 2012.

[41] B. Gao, L. He, and C. Chen. Modelling the bandwidth allocation problem in mobile service-oriented networks. In *Proceedings of the 18th ACM In-*

ternational Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, pages 307–311. ACM, 2015.

[42] Y. Gao, H. Guan, Z. Qi, Y. Hou, and L. Liu. A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *Journal of Computer and System Sciences*, 79(8):1230–1242, 2013.

[43] P. Ghosh, N. Roy, S. K. Das, and K. Basu. A game theory based pricing strategy for job allocation in mobile grids. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 82. IEEE, 2004.

[44] P. Ghosh, K. Basu, and S. K. Das. A game theory-based pricing strategy to support single/multiclass job allocation schemes for bandwidth-constrained distributed computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 18(3):289–306, 2007.

[45] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 51–62. ACM, 2009.

[46] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration. In *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 7. IEEE Computer Society, 2006.

[47] D. Grosu and A. T. Chronopoulos. Noncooperative load balancing in distributed systems. *Journal of parallel and distributed computing*, 65(9): 1022–1034, 2005.

[48] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. *ACM SIGCOMM Computer Communication Review*, 38(4):75–86, 2008.

[49] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 39(4):63–74, 2009.

[50] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COn-ference*, page 15. ACM, 2010.

[51] Y. Guo, A. L. Stolyar, and A. Walid. Shadow-routing based dynamic algorithms for virtual machine placement in a network cloud. In *INFOCOM, 2013 Proceedings IEEE*, pages 620–628. IEEE, 2013.

[52] L. Gyarmati and T. A. Trinh. Scafida: A scale-free network inspired data center architecture. *ACM SIGCOMM Computer Communication Review*, 40(5):4–12, 2010.

[53] M. E. Haque, Y. He, S. Elnikety, R. Bianchini, K. S. McKinley, et al. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 161–175. ACM, 2015.

[54] L. He, S. A. Jarvis, D. P. Spooner, H. Jiang, D. N. Dillenberger, and G. R. Nudd. Allocating non-real-time and soft real-time jobs in multiclusters. *Parallel and Distributed Systems, IEEE Transactions on*, 17(2):99–112, 2006.

[55] L. He, D. Zou, Z. Zhang, C. Chen, H. Jin, and S. A. Jarvis. Developing resource consolidation frameworks for moldable virtual machines in clouds. *Future Generation Computer Systems*, 2013. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2012.05.015.

[56] A. Headquarters. Cisco data center infrastructure 2.5 design guide. 2007.

[57] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009.

[58] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[59] L. Hu, H. Jin, X. Liao, X. Xiong, and H. Liu. Magnet: A novel scheduling policy for power reduction in cluster with virtual machines. In *Cluster Computing, 2008 IEEE International Conference on*, pages 13–22. IEEE, 2008.

[60] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling jobs across geo-distributed datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 111–124. ACM, 2015.

[61] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[62] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[63] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 10. ACM, 2012.

[64] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 219–230. ACM, 2013.

[65] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 407–420. ACM, 2015.

[66] JCodec. Java implementation for video de/encoder. URL `http://jcodec.org/`.

[67] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang. Joint vm placement and routing for data center traffic engineering. In *INFOCOM, 2012 Proceedings IEEE*, pages 2876–2880. IEEE, 2012.

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

[69] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, 2015.

[70] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1287–1294. IEEE, 2012.

[71] S. U. Khan and I. Ahmad. A cooperative game theoretical technique for joint optimization of energy consumption and response time in computational grids. *Parallel and Distributed Systems, IEEE Transactions on*, 20 (3):346–360, 2009.

[72] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the

linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[73] D. Kliazovich, P. Bouvry, and S. U. Khan. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283, 2012.

[74] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[75] Y.-K. Kwok, K. Hwang, and S. Song. Selfish grids: Game-theoretic modeling and nas/psa benchmark evaluation. *Parallel and Distributed Systems, IEEE Transactions on*, 18(5):621–636, 2007.

[76] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[77] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *Computers, IEEE Transactions on*, 100(10):892–901, 1985.

[78] W. Leontief. Input-output analysis. *The new palgrave. A dictionary of economics*, 2:860–64, 1987.

[79] B. Li, J. Li, J. Huai, T. Wo, Q. Li, and L. Zhong. Enacloud: An energy-saving application live placement approach for cloud computing environments. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 17–24. IEEE, 2009.

[80] M. A. M. McKelvey, Richard D. and T. L. Turocy. Gambit: Software tools for game theory, version 14.1.0., 2014. URL `http://goo.gl/QobQAd`.

[81] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[82] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[83] Microsoft. Microsoft hyper-v, 2012. URL `https://goo.gl/zDs2Tz`.

[84] B. L. Miller and D. E. Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary Computation*, 4(2):113–131, 1996.

[85] D. Milojičić, I. M. Llorente, and R. S. Montero. Opennebula: A cloud management tool. *IEEE Internet Computing*, (2):11–14, 2011.

[86] B. News. Cloud profit, 2015. URL `http://goo.gl/QrOL9z`.

[87] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pages 124–131. IEEE, 2009.

[88] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[89] S. Penmatsa and A. T. Chronopoulos. Cooperative load balancing for a network of heterogeneous computers. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[90] V. Petrucci, O. Loques, and D. Mossé. A dynamic optimization model for power and performance management of virtualized clusters. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, pages 225–233. ACM, 2010.

[91] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, tech-*

nologies, architectures, and protocols for computer communication, pages 187–198. ACM, 2012.

[92] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, and Y. T. J. R. Santos. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. 2013.

[93] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[94] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, page 84, 2012.

[95] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[96] O. Sefraoui, M. Aissaoui, and M. Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012.

[97] A. W. Services. Amazon ec2 pricing, . URL `http://goo.gl/92Imlv`.

[98] A. W. Services. Aws elastic load balancing, . URL `http://goo.gl/GfqA0m`.

[99] A. W. Services. Aws case study, 2012. URL `https://goo.gl/si6bce`.

[100] A. W. Services. Using amazon sqs with other aws infrastructure web services, 2012. URL `http://goo.gl/C0kLZ9`.

[101] A. W. Services. Aws case study: Nasdaq finqlod, 2012. URL `http://goo.gl/KJbEmh`.

[102] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, 2012.

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

[104] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell. Modeling virtual machine performance: challenges and approaches. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):55–60, 2010.

[105] P. J. Van Laarhoven and E. H. Aarts. *Simulated annealing*. Springer, 1987.

[106] A. Verma, P. Ahuja, and A. Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *Middleware 2008*, pages 243–264. Springer, 2008.

[107] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[108] M. Wang, X. Meng, and L. Zhang. Consolidating virtual machines with dynamic bandwidth demand in data centers. In *INFOCOM, 2011 Proceedings IEEE*, pages 71–75. IEEE, 2011.

[109] J. Wilkes. More google cluster data., Nov 2011. URL `https://goo.gl/OnnkLu`.

[110] A. Williams, M. Arlitt, C. Williamson, and K. Barker. Web workload characterization: Ten years later. In *Web content delivery*, pages 3–21. Springer, 2005.

[111] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: incorporating time-varying network reservations in data centers. *ACM SIGCOMM Computer Communication Review*, 42(4):199–210, 2012.

[112] J. Xu and J. A. Fortes. Multi-objective virtual machine placement in virtualized data center environments. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 179–188. IEEE, 2010.

[113] YARN. Hadoop fair scheduler. URL `http://goo.gl/hWUGhV`.

[114] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox. Analysis of virtualization technologies for high performance computing environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 9–16. IEEE, 2011.

[115] L. Yu and H. Shen. Bandwidth guarantee under demand uncertainty in multi-tenant clouds. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 258–267. IEEE, 2014.

[116] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[117] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *HotOS*, 2007.

[118] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment*, 7(13):1393–1404, 2014.