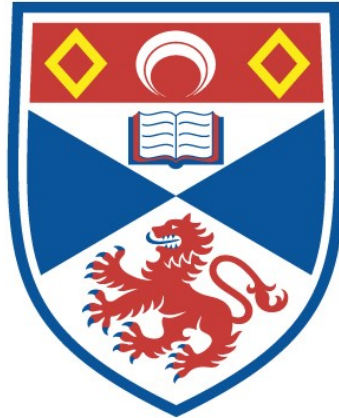


OPTIMISING THE USAGE OF CLOUD RESOURCES
FOR EXECUTING BAG-OF-TASKS APPLICATIONS

Long Thanh Thai

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



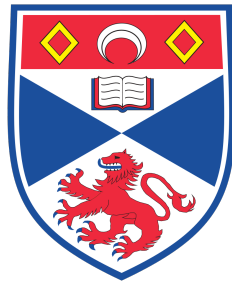
2017

Full metadata for this item is available in
St Andrews Research Repository
at:
<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:
<http://hdl.handle.net/10023/15642>

This item is protected by original copyright

Optimising the Usage of Cloud Resources for Executing Bag-of-Tasks Applications



Long Thanh Thai

School of Computer Science
University of St Andrews

This thesis is submitted in partial fulfilment for the degree of
Doctor of Philosophy

August 2017

Abstract

Cloud computing has been widely adopted by many organisations, due to its flexibility in resource provisioning and on-demand pricing models. Entire clusters of machines can now be dynamically provisioned to meet the computational demands of users. By moving operations to the cloud, users hope to reduce the costs of building and maintaining a computational cluster without sacrificing the quality of service.

However, cloud computing has presented challenges in scheduling and managing the usage of resources, which users of more traditional resource pooling models, such as grid and clusters, have never encountered before. Firstly, the costs associated with resource usage changes dynamically, and is based on the type and duration of resources used; this prevents users from greedily acquiring as many resources as possible due to the associated costs. Secondly, the cloud computing marketplace offers an assortment of on-demand resources with a wide range of performance capabilities. Given the variety of resources, this makes it difficult for users to construct a cluster which is suitable for their applications. As a result, it is challenging for users to ensure the desired quality of service while running applications on the cloud.

The research in this thesis focuses on optimising the usage of cloud computing resources. We propose approaches for scheduling the execution of applications on to the cloud, such that the desired performance is met whilst the incurred monetary cost is minimised. Furthermore, this thesis presents a set of mechanisms which manages the execution at runtime, in order to detect and handle unexpected events with undesirable consequences, such as the violation of quality of service, or cost overruns.

Using both simulated and real world experiments, we validate the feasibility of the proposed research by executing applications on the cloud with low costs without sacrificing performance. The key result is that it is possible to optimise the usage of cloud resources for user applications by using the research reported in this thesis.

Acknowledgements

This thesis would not have been possible if it were not for the support that I have been fortunate enough to receive for the last three and a half years. I would like to express my deepest gratitude to:

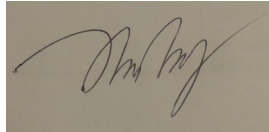
- My primary supervisor Dr. Adam Barker, and secondary supervisor Dr. Blesson Varghese for their expert supervision, guidance, patience and trust.
- My parents and elder-brother for their unconditional love and support, without which I would not have had the courage to pursue this PhD.
- The School of Computer Science, St Leonard College, the EPSRC Working Together project, which has been led expertly by Professor Ian Miguel, and the EPSRC IAA for their generous financial support.
- Dr. Juan Ye, Dr. Alex Voss, Dr. Mike Weir, Dr. Tristan Henderson, Dr. Edwin Brady, and Dr. John Thomson for their constructive comments and suggestions.
- The staff and academics at the School of Computer Science for creating a fascinating, challenging and welcoming research environment.
- My fellow PhD students for sharing ideas, times, and (mostly) drink together to make my journey highly enjoyable.
- My best friends, Hung and Ha, and their wives, Van and Thao, for their hospitality and long lasting friendship.
- The small but friendly Vietnamese community in St Andrews for reminding me of how wonderful my country and its people are.
- And for anyone that I have had a pleasure to meet in the last few years, thank you for being a part of my journey to be (hopefully) a better person.

Declaration

I, Long Thai, hereby certify that this thesis, which is approximately 34,500 words in length, has been written by me, and that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree. I was admitted as a research student and as a candidate for the degree of Doctor of Philosophy in September 2013; the higher study for which this is a record was carried out in the University of St Andrews between 2013 and 2017.

Date:

Signature of candidate:

A rectangular box containing a handwritten signature in black ink on a light brown background. The signature is cursive and appears to read 'Long Thai'.

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date:

Signature of supervisor:

Permission

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis: *Access to printed and electronic publication of this thesis through the University of St Andrews.*

Date:

Signature of candidate:

Date:

Signature of supervisor:

Table of contents

List of figures	xv
List of tables	xix
1 Introduction	1
1.1 Cloud Computing	1
1.2 Bag-of-Tasks Applications	4
1.3 Research Hypotheses	6
1.3.1 Scheduling approaches can minimise running costs of BoT applications on the cloud and achieve the desired Quality of Service provided as user defined deadlines	6
1.3.2 Unexpected events, such as performance variation, can be detected and handled by the execution management mechanisms at runtime	7
1.4 Contributions	7
1.5 Publications	8
1.6 Organisation	10
2 Literature Review	13
2.1 Related Work	13
2.1.1 Overview of the Survey Methodology	13
2.1.2 Scheduling in a Homogeneous Environment	15
2.1.3 Scheduling in a Heterogeneous Environment	17
2.2 Taxonomy	19
2.2.1 Functionality	21
2.2.2 Requirements	22
2.2.3 Dynamic Scheduling	23
2.2.4 Parameter Estimation	25
2.2.5 Solving Methods	26

2.2.6	Application Heterogeneity	27
2.3	Discussion	28
2.3.1	Current Trends	28
2.4	Requirements Analysis	31
2.4.1	Heterogeneous Environment	31
2.4.2	Satisfying Deadlines While Minimising the Monetary Cost	31
2.4.3	Flexible Execution	32
2.4.4	Trade-off Aware Solving Methods	32
2.5	Chapter Summary	32
3	Mathematical Representation of the Research Problem	35
3.1	Environment Modelling	35
3.2	Job Execution Modelling	38
3.3	Problem Modelling	41
3.4	Chapter Summary	43
4	Workload Assignment	45
4.1	Utility Functions	46
4.1.1	Finding Preceding and Succeeding Workloads	46
4.1.2	Calculate Permissible Delay	47
4.1.3	Shift Workloads	47
4.1.4	Execution Pre-emption	48
4.2	Workload Assignment Algorithm	49
4.3	Chapter Summary	51
5	Execution Scheduling	55
5.1	The Exact Approach	56
5.2	Single Job Scheduling Approach	60
5.2.1	The Hybrid Scheduling Approach	61
5.2.2	Heuristic Single Job Scheduling	64
5.2.3	Handling Multiple Jobs Using Single Job Scheduling Approaches	66
5.3	Chapter Summary	67
6	Execution Management	69
6.1	Dynamic Scheduling	69
6.1.1	Progress Monitoring	70
6.1.2	Progress Categorisation	70

6.1.3	Dynamic Reassignment	74
6.2	Handling Unknown Applications	75
6.2.1	Determine the Sampling Duration	76
6.2.2	Schedule the Sampling Phase	76
6.3	Chapter Summary	79
7	Design and Implementation	81
7.1	Data Transfer Object (DTO)	82
7.1.1	Application	82
7.1.2	Job	82
7.1.3	InstanceType	83
7.1.4	Instance	83
7.1.5	Workload	83
7.2	Components	83
7.2.1	Assignment Service	84
7.2.2	Scheduler	85
7.2.3	Reassignment Service	87
7.2.4	Unknown Handler	87
7.2.5	Cloud Manager	88
7.2.6	Executor	89
7.3	Supported Features	93
7.3.1	Submission Handling	93
7.3.2	Execution Monitoring and Management	96
7.4	Chapter Summary	96
8	Evaluation	97
8.1	Introduction	97
8.2	Comparing the Scheduling Approaches	97
8.2.1	Environment Set-up	98
8.2.2	Experiment Results and Discussion	101
8.2.3	Discussion	107
8.3	Evaluating the Unknown Handling Mechanism	110
8.3.1	Environment Set-up	110
8.3.2	Experiment Results	114
8.3.3	Discussion and Summary	120
8.4	Dynamic Reassignment	120
8.4.1	Experiment Set-up	120

8.4.2	Experiment Results and Discussion	121
8.5	Cloud Experiments	126
8.5.1	Environment Set-up	126
8.5.2	Experiment Results	127
8.5.3	Discussion	130
8.6	Chapter Summary	130
9	Conclusion	133
9.1	Thesis Summary and Contributions	133
9.2	Lessons Learned	135
9.3	Future Work	137
	References	139
	Appendix A Experiment Results	147
A.1	Scheduling Approaches	147
A.2	Evaluating the Unknown Handling Mechanism	156
A.3	Dynamic Reassignment	156
A.4	Cloud Experiments	160

List of figures

2.1	Taxonomy of Cloud Scheduling Frameworks	20
3.1	Example of the Life of an Instance	38
3.2	Example of a Workload Assigned to an Instance	39
3.3	Example of a VM Life Based on Assigned Workload	41
4.1	Activity Diagram for the Workload Assignment Process presented by Algorithm 4.5	52
5.1	Example of Execution Order in an Instance	57
6.1	Gap for Receiving Extra Workload	73
7.1	Data Transfer Objects	82
7.2	The Components of the Software Framework	84
7.3	Static Scheduler Hierarchy Structure	85
7.4	Job Submission Handling Process	94
7.5	Execution Management Process	95
8.1	Experiment Result of Three Scheduling Approaches When the Number of Jobs Varies	103
8.2	Experiment Result of Three Scheduling Approaches When the Number of Tasks Varies	104
8.3	Experiment Result of Three Scheduling Approaches When the Number of Instance Types Varies	106
8.4	Experiment Result of Three Scheduling Approaches When the Deadline Varies	108
8.5	The Task Execution Times of the Applications Retrieved by Running Sampling Execution. The error bars illustrate the standard errors.	113

-
- 8.6 The total cost and violation of different settings: in the **known** setting, the task execution times are already available. In the **unknown** setting, the task execution times are not available but going to be estimated during an execution. The **medium.8**, **medium.10**, **large.4**, **large.5**, **xlarge.2**, and **xlarge.3** settings have a fixed number of VMs of a given instance type to execute tasks, thus do not rely on the knowledge regarding the task execution times. The error bars illustrate the standard errors. 116
- 8.7 The violation costs of different settings: in the **known** setting, the task execution times are already available. In the **unknown** setting, the task execution times are not available but going to be estimated during an execution. The **medium.8**, **medium.10**, **large.4**, **large.5**, **xlarge.2**, and **xlarge.3** settings have a fixed number of VMs of a given instance type to execute tasks, thus do not rely on the knowledge regarding the task execution times. 117
- 8.8 The number of used ATUs per instance Type of each setting: in the **known** setting, the task execution times are already available. In the **unknown** setting, the task execution times are not available but going to be estimated during an execution. The **medium.8**, **medium.10**, **large.4**, **large.5**, **xlarge.2**, and **xlarge.3** settings have a fixed number of VMs of a given instance type to execute tasks, thus do not rely on the knowledge regarding the task execution times. 118
- 8.9 Resource Utilisation of each setting: in the **known** setting, the task execution times are already available. In the **unknown** setting, the task execution times are not available but going to be estimated during an execution. The **medium.8**, **medium.10**, **large.4**, **large.5**, **xlarge.2**, and **xlarge.3** settings have a fixed number of VMs of a given instance type to execute tasks, thus do not rely on the knowledge regarding the task execution times. 119
- 8.10 Average Number of Violated Tasks for Each Approach When Dynamic Scheduling Is Turned On/Off. The error bars illustrate the standard errors. 123
- 8.11 Average Amount of Violated Time for Each Approach When Dynamic Scheduling Is Turned On/Off. The error bars illustrate the standard errors. 124
- 8.12 Average Cost of Each Approach When Dynamic Scheduling Is Turned On/Off. The error bars illustrate the standard errors. 125
- 8.13 Total costs of each approach. The first three bars represent the total cost of the exact, hybrid, and heuristic approaches which aim to build a heterogeneous cloud clusters. The last three plots represent the total costs of the homogeneous cloud clusters which consist of VMs of only one instance type. 128

8.14 Violation of Each Approach. The error bars illustrate the standard errors. . . 129

List of tables

2.1	Taxonomy of BoT Scheduling Methodologies	29
8.1	Summary of the Independent and Dependent Variable	99
8.2	AWS Instance Types	111
8.3	Job Specification	112
8.4	Experiment Results	115
A.1	Summary of Solving Times in Milliseconds of the Experiment in which the Number of Jobs Varied	148
A.2	Summary of Total Costs in Dollar of the Experiment in which the Number of Jobs Varied	149
A.3	Summary of Solving Times in Milliseconds of the Experiment in which the Number of Tasks Varied	150
A.4	Summary of Total Costs in Dollars of the Experiment in which the Number of Tasks Varied	151
A.5	Summary of Solving Times in Milliseconds of the Experiment in which the Number of Instance Types Varied	152
A.6	Summary of Total Costs in Dollars of the Experiment in which the Number of Instance Types Varied	153
A.7	Summary of Solving Times in Milliseconds of the Experiment in which the Deadline Varied	154
A.8	Summary of Total Costs in Dollars of the Experiment in which the Deadline Varied	155
A.9	The results of the experiment evaluating the Unknown Handling Mechanism	156
A.10	Cost in Dollars of Each Approach When Dynamic Scheduling Is Turned On/Off	157
A.11	Number of Violated Tasks for Each Approach When Dynamic Scheduling Is Turned On/Off	158

A.12 Average Amount of Violated Time in Seconds for Each Approach When Dynamic Scheduling Is Turned On/Off	159
A.13 The results of the experiments evaluating the feasibility of the proposed research in real life cloud, i.e. AWS	160

List of Algorithms

4.1	Find Position	46
4.2	Calculate Permissible Delay	47
4.3	Shift Workloads	48
4.4	Workload Pre-emption	48
4.5	Workload Assignment	50
5.1	Create Initial Plan	64
5.2	Transform Plan	65
5.3	Single Job Scheduling	66
6.1	Progress Categorisation	72
6.2	Dynamic Reassignment	74
6.3	Select Sampling VMs	77
6.4	Schedule Sampling Phase	78
8.1	Generate Task Execution Times	99

Chapter 1

Introduction

1.1 Cloud Computing

Cloud computing, since it was first introduced in October 2007 when Google and IBM cooperated to build a data centre helping students remotely program and research via the Internet [41], has become not only a new trend of information technology research but also a successful business model which has been widely applied. This paradigm refers to the idea of *outsourcing not only applications but also the operating platform and hardware infrastructure through visualisation*.

Although there is no unified definition of Cloud computing as different researchers or projects focus on different aspects and functionalities in different points of view, there is the a of characteristics which are able to provide the general view of the technology [20, 16, 66, 46]:

- **Parallel and distributed system:** users access cloud's services via the Internet which means a system has to handle many requests at the same time without having any concurrent access problems. In order to perform this task successfully, a cloud system must be able to perform parallel task and support concurrent sharing and aggregation resources which are geographically distributed.
- **Virtualisation:** all services, especially platforms and infrastructures, are virtualised in order to hide the underneath architecture from users. The main benefit of virtualisation is to allocate resources effectively, e.g. physical machines can be divided into smaller virtual ones or grouped together to form a powerful virtual machine. Moreover, virtualisation creates isolation between machines so that a fault on one machine does not affect the rest.

- **Scalability:** as cloud computing aims to provide services on demand in order to prevent under and/or over resource utilisation, a system must be able to dynamically allocate or deallocate resources on the demand without interrupting the operation.
- **Transparency and abstract:** the level of abstraction can be used to distinguish different kind of utility computing. Some vendors allow a user to interfere with a (virtual) hardware level, i.e. *infrastructure as a service (IaaS)*, while others require users to run applications on a standard built environment, i.e. *platform as a service (PaaS)*. Last but not least, some organisations provide their services based on Cloud computing, i.e. *software as a service (SaaS)*.
- **Service Level Agreement (SLA):** it is the contract between users and providers in order to define what and how services are provided, i.e. Quality of Service (QoS).
- **Pay-as-you-go:** a user only pays for what he/she uses, i.e. there is no up-front investment that user has to pay for. This feature is very attractive for start-ups or organisations who do not have the large capital to build their own data centres.

A cloud computing environment can be deployed in four different models [46]:

- **Public cloud:** which is offered to the general public by organisations that own large data centres such as Amazon, Google, and Microsoft.
- **Private cloud:** which can be used by only one organisation which owns the infrastructure in which a cloud is deployed.
- **Hybrid cloud:** is a combination of public and private cloud. Normally, an organisation tries to run most of its operation on the private cloud, however, resources from public cloud can be added in order to handle peak workload, this model is also called **cloud bursting**.
- **Community cloud:** which is offered to an exclusive group of organisations.

In this thesis, we mainly focus on a public cloud setting, in which users pay for the amount of resources that they use. Cloud providers often offer different pricing plans for users to select from based on their requirements and/or preferences. The most common pricing schemes are:

- **On-demand resources:** can be acquired by a user at any time and have fixed prices. This is the most commonly used resource type due to its flexibility and guarantee, i.e. a user is guaranteed to be able to acquired additional resources at any time. However, it is also the most expensive.

- **Reserved resources** [4]: refer to a certain amount of resources which are reserved specially for a user with a lower price. It is suitable for a user who has a brief estimation of required resources in a long period of time, e.g. from one to three years. The price reduction can go up to more than 60% [3].
- **Spot resources** [5], or preemptible virtual machines [8]: provide an auction-like environment in which many users are bidding against each other. As a result, it is possible to acquire resources which are cheaper compared to on-demand resources. However, the price can change dynamically depending on the number of bidders. Moreover, resources can be terminated anytime without notice in order to be reallocated to the highest bidders.

Recently, cloud computing has become popular by offering an opportunity for organisations, especially those with limited financial capacity, the ability to access computing resources. Instead of paying an upfront investment, cloud users can employ a pay-as-you-go pricing scheme in which they only need to pay for the amount of resources which are actually used. Moreover, the elasticity of cloud computing allows its users to add or remove resources at runtime seamlessly. For instance, more virtual machines (VMs) can be added to a user's cluster when the resource demands are high in order to accommodate a peak workload and ensure the desired performance. Similarly, when the resource demands are low, a user can remove some VMs in order to reduce the cost. Cloud providers (such as Amazon Web Service (AWS) [1], Microsoft Azure [6], or Google Cloud Platform [7]) are often large organisations with many data centres around the world. Hence, it is possible for them to satisfy all users' resource demands. In other words, cloud resources are virtually unlimited. Finally, cloud providers offer a wide variety of machine types, each of which has different hardware specification and performance. Hence, users can select a hardware specifications, called an **instance type**, or a combination of many to suit the nature of their applications. For example, a user can select compute-optimised machines for CPU intensive applications, or memory-optimised machines for memory intensive software. The flexibility of cloud computing resources is not limited to hardware specification but can be further extended to software stack, e.g. operating systems and pre-installed software, and even geographical locations as cloud providers normally have many data centres all over the world.

Even though cloud computing offers many advantages to users, it also introduces unique challenges which need to be taken into account. Firstly, due to the pay-as-you-go pricing scheme, every decision of using cloud resources results in monetary cost. Hence, a user cannot simply employ a greedy approach which acquires as many resources as possible. Even though cloud resources are virtually not limited, a user's budget is. As a result, it requires

users to have a careful scheduling plan in order to acquire enough resources to achieve the desired performance without resulting in unreasonable monetary cost. Secondly, the wide variety of instance types offered by cloud providers can be overwhelming for users. More specifically, a user must consider tens, or even hundreds, of different options. Those issues can be further complicated when a Quality-of-Service (QoS), e.g. a desired performance that a user wants to achieve, is taken into account. Last but not least, the heterogeneity of cloud resources also extends to the lower hardware infrastructure. In other words, a user's VMs can reside on physical machines with different hardware specification, which results in unexpected performance variation. For instance, a VM running on a new physical machine will have better performance in comparison to another VM of the same type running on an old physical machine.

1.2 Bag-of-Tasks Applications

For the past decades, the computational problems faced by academic and industrial have significantly increased in complexity and volume. As a result, there has been a shift from running the computation on a mainframe or high performing computers to distributing the computation of a collection of commodity machines. In other words, it is common nowadays to split a complex computation into many smaller tasks each of which can be executed independently on a single machine. This trend in research and development explains the raise in popularity of MapReduce programming model [27], and computing frameworks such as Apache Hadoop [9] and Apache Spark [11]. The general idea behind these models and frameworks is to break a computation into multiple related stages (e.g. mapping and reducing stages). Each stage consists of a number of tasks that can be executed independently and in parallel and can be considered as an individual application, called **Bag-of-Tasks (BoT)**.

It should be noted that a BoT application does not always need to be a part of any larger application. For instance, one of the popular types of BoT application is a simulation application, e.g. Monte Carlo simulation [38]. This type of application aims to find the pattern within a seemingly random process by repeating it hundred or thousand of times. For example, the Molecular Dynamics Simulation is used to calculate the trajectory of the particles and the forces they exert [37]. Moreover, Monte Carlo simulation has been widely used in the finance sector for portfolio evaluation [33], personal financial planning [48], etc. Similarly, a parameter sweep application [22] is a BoT application which consists of a set of parameters. In each execution, each parameter is set to a specific value within a predefined range. The goal is to find the optimal value for each parameter based on the predefined

criteria. For instance, a parameter sweep application can be used to find a set of parameters for machine learning techniques, such as support vector machines [23].

In this thesis, we focus on the execution of BoT applications on the cloud. BoT applications are widely used by scientific communities and commercial organisations whose applications are too complicated to be executed on a single machine, even a high performing one. For instance, BoT jobs dominate the number of applications submitted to and usage of CPU time in grid environments [35]. Similarly, the jobs executed on Facebook data centres are reported to be mostly independent tasks [30]. However, scheduling the execution of BoT applications can be challenging due to its unique characteristics. More specifically, since the tasks can be executed in parallel, it is easy to naively acquire as many resources as possible. However, this greedy approach cannot be applied in the cloud environment in which there is a monetary cost incurred. Cloud computing resources can be unlimited but a user's wallet is definitely not.

Besides BoT, there are two other application types that are common nowadays. The first one is a workflow which can be represented as a directed acyclic graph (DAG), in which nodes are tasks and edges are dependencies between tasks. As a result, a task in a workflow cannot be executed unless all of its parents, i.e. tasks whose outputs are used as its input, are executed. The second type of application is user-facing, or interactive, application which refers to a service which directly interacts with users [31, 26]. As a result, user-facing application must always be online, i.e. resources must always be allocated to it. One of the main differences between scheduling BoT applications compared to workflow and user-facing ones is the resource consumption pattern. More specifically, a BoT application generally requires the same amount of resources throughout its execution since all tasks can be executed in parallel. On the other hand, the amount of resources required by a workflow application can change at different times based on the number of tasks that can be executed in parallel. Similarly, a user-facing application requires a different amount of resources based on the workload it needs to handle. However, at any time, it needs to maintain the minimal amount of resources to ensure the availability. As a conclusion, scheduling the execution of a BoT application is simpler compared to the other two. However, we believe that this research direction is worth pursuing because a workflow application and the underlying mechanism of a user-facing application can be broken into different dependent stages, each of which is a BoT application.

1.3 Research Hypotheses

This thesis investigates two central hypotheses regarding optimising the usage of cloud computing resources. They are related to the goal of obtaining the desired performance of BoT applications running on the cloud while minimising the incurred monetary cost. These hypotheses are considered through out this thesis and examined empirically in our evaluation.

1.3.1 Scheduling approaches can minimise running costs of BoT applications on the cloud and achieve the desired Quality of Service provided as user defined deadlines

Novel multi-objective scheduling approaches that account for both the amount of resources required for executing multiple applications in a cloud cluster and workload mapping onto these resources are required for minimising the cost of executing BoT applications while maintaining the desired level of performance.

In this thesis, the desired performance of an application is represented as a user-defined **deadline** which indicates a time within which all tasks of an application must be executed. The deadline constraint is widely employed by both industrial and academic communities for BoT applications.

We believe that satisfying the deadline and minimising the cost can be achieved by carefully constructing and assigning tasks to a **heterogeneous cloud cluster** that consists of VMs created from different instance types. The combination of different virtualised hardware specifications creates a flexible computing environment which accommodates a workload that changes dynamically. However, this is not a simple task since a wide variety of instance types with varying costs and performance are offered by cloud providers. As a result, it is necessary to have an execution scheduling mechanism that is not only aware of applications and their requirements but also the cloud environment including the performance and pricing of different instance types.

Notably, the QoS of an application consists of other criteria such as security, reliability, availability, etc. In this thesis, we solely focus on the performance aspect since it can be easily used to judge the quality of an application. Moreover, other objectives (e.g. security, reliability, availability) are normally handled by the cloud providers and a user has limited or even no control over them.

1.3.2 Unexpected events, such as performance variation, can be detected and handled by the execution management mechanisms at runtime

As mentioned earlier, performance variation is unavoidable in the cloud environment. This can result in undesirable consequences such as deadline violation or additional cost overheads. As a result, it is necessary to have a dynamic mechanism which monitors an execution, detects and handles any potentially harmful events as quickly as possible.

Furthermore, certain information that are required for the scheduling process may not be available prior to an execution. In this thesis, we investigate and propose a mechanism for estimating the information required to schedule the execution of applications in the cloud. We argue that it is necessary to estimate those properties during runtime and then use them to optimise cloud usage, even though it may result in cost overheads.

1.4 Contributions

In summary, this thesis provides the following contributions:

- A mathematical model that represents the execution of multiple BoT jobs, each of which has different characteristic and requirement, on the cloud. This model provides a complete view of the problem that we are aiming to address as well as the environment in which the problem takes place.
- Three different approaches for optimisation cloud resource usage. Each approach has a different degree of optimality and complexity. More precisely, the more complex an approach is, the more optimal its solution is. As a result, those approaches present the trade-off between solution optimality and complexity, which is represented by the solving time.
- The dynamic mechanism that manages the execution during runtime. This contribution can be further divided into two components. The first one detects and handles unexpected events during an execution that can potentially result in an undesired consequence. The second component deals with unknown parameters by estimating them via a sampling process.
- The software framework which materialises the proposed research into a complete application. It is able to perform an end-to-end execution which starts when jobs are submitted and finishes after all tasks are completely executed.

- Experiment results that show the improvement offered by the proposed approaches in comparison to the existing methods. We not only evaluate each component separately but also demonstrate the applicability of our research as a complete solution in the real world environment.

1.5 Publications

The work detailed in this thesis has resulted in a number of peer reviewed publications as follows:

1. Long Thai, Blesson Varghese and Adam Barker. Executing Bag of Distributed Tasks on the Cloud: Investigating the Trade-offs Between Performance and Cost. In Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2014), pages 400-407, IEEE Computer Society.

This paper acts as the starting point of this PhD research as it investigated the trade-off between performance and cost in cloud computing environment. By showing that the increase in performance results in the increase in monetary cost, it presented the problem of scheduling the execution of BoT application on the cloud so that the desired performance could be achieved with the minimum cost.

2. Long Thai, Blesson Varghese and Adam Barker. Executing Bag of Distributed Tasks on Virtually Unlimited Cloud Resources. In Proceedings of the 5th International Conference on Cloud Computing and Services Science (CLOSER), pages 373-380, 2015.

This paper extended the previous one by removing the limit regarding the amount of available cloud resources. In other words, it was possible for a user to acquire as much resource as he or she needed. However, doing so greedily could result in high monetary cost. The paper showed that by carefully scheduling the execution on the cloud, the better performance could be achieved with the same cost compared to other naive approaches.

3. Long Thai, Blesson Varghese and Adam Barker. Budget Constrained Execution of Multiple Bag-of-Tasks Applications on the Cloud. In Proceedings of the 8th IEEE International Conference on Cloud Computing (CLOUD 2015), 975-980, IEEE Computer Society.

4. Long Thai, Blesson Varghese and Adam Barker. Task Scheduling on the Cloud with Hard Constraints. In Proceedings of the 11th World Congress on Services (IEEE SERVICES 2015), pages 95-102, IEEE Computer Society.

In these two papers, we finally viewed the cloud as a highly flexible computing environment with not only wide variety but also high amount of computing resources. The papers presented the heuristic algorithms that determined the amount of resources and scheduled the execution so that all requirements regarding performance and cost could be satisfied. The Sections 4.2 and 5.2.2 are based on the work of these papers.

5. Long Thai, Blesson Varghese, and Adam Barker. 2016. Minimising the Execution of Unknown Bag-of-Task Jobs with Deadlines on the Cloud. In Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing (DIDC '16). ACM, New York, NY, USA, 3-10.

The first contribution of this paper was the development of the hybrid scheduling approach which combined both exact and heuristic algorithms. This approach is elaborated in Section 5.2.1. It also introduced a mechanism for handling unknown applications but estimating the information required for the scheduling process. Section 6.2 will provides detail explanation of this mechanism.

6. Long Thai, Blesson Varghese, and Adam Barker, 2016. Algorithms for optimising heterogeneous Cloud virtual machine clusters . in 8th IEEE International Conference on Cloud Computing Technology and Science . IEEE , 8th IEEE International Conference on Cloud Computing Technology and Science , Luxembourg , 12-15 December .

In this paper, we developed the exact algorithm which guaranteed optimal execution in which the monetary cost was minimised. The detailed explanation for this approach will be presented by Section 5.1. This approach was compared with the hybrid approach presented in the previous paper in order to demonstrate the trade-off between complexity and optimality of scheduling approaches. Finally, the paper also presented a dynamic scheduling mechanism which was able to detect and handle potential deadline violation caused by performance variation. The content of Section 6.1 is largely based on this work.

Additionally, during this PhD, the author has had opportunities to participate in different research projects which have yielded a number of peer reviewed publications:

1. Long Thai, Adam Barker, Blesson Varghese, Ozgur Akgun and Ian Miguel. Optimal Deployment of Geographically Distributed Workflow Engines on the Cloud. In Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2014), pages 811-816, IEEE Computer Society.

2. Blesson Varghese, Ozgur Akgun, Ian Miguel, Long Thai and Adam Barker. Cloud Benchmarking for Performance. In Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2014), pages 535-540, IEEE Computer Society.
3. Adam Barker, Blesson Varghese and Long Thai. Cloud Services Brokerage: A Survey and Research Roadmap. In Proceedings of the 8th IEEE International Conference on Cloud Computing (CLOUD 2015), pages 1029-1032, IEEE Computer Society.
4. Blesson Varghese, Lawan Thamsuhang Subba, Long Thai and Adam Barker. Container-Based Cloud Virtual Machine Benchmarking. In Proceedings of the IEEE International Conference on Cloud Engineering (IC2E 2016), pages 192–201, IEEE Computer Society.
5. Blesson Varghese, Lawan Thamsuhang Subba, Long Thai and Adam Barker. DocLite: A Docker-Based Lightweight CloudBenchmarking Tool. In Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2016), pages 213–222, IEEE Computer Society.
6. Blesson Varghese, Ozgur Akgun, Ian Miguel, Long Thai and Adam Barker. Cloud Benchmarking For Maximising Performance of Scientific Applications. To appear in IEEE Transactions on Cloud Computing.

1.6 Organisation

This thesis has nine chapters, some of which have already been reported in our peer-reviewed publications. Chapter 1 introduces this thesis by presenting its motivations and hypothesis.

Chapter 2 reviews existing work in optimising cloud resource usage for BoT application. Based on the the reviewed literature, we construct the taxonomy which can be used to classify the challenges that will be addressed by the research in this thesis.

Chapter 3 builds a mathematical model that represent the problem of optimising the execution of BoT applications on the cloud. This chapter is the result of the incremental research process resulting in the paper [59–64].

Chapter 4 introduces a set of algorithms which assign tasks to given VMs without resulting in any violation and will be re-used later by other mechanisms, most of which have been presented in [63, 64].

Chapter 5 presents and discusses three different approaches for scheduling the execution of BoT jobs on the cloud. Those approaches represent the trade-off between optimality and

complexity of the optimisation process. More precisely, an optimal solution can be achieved by a complex optimisation process. On the other hand, a less complex process can only find a sub-optimal solution but takes less time. This chapter is the result of [61, 63, 64].

Chapter 6 describes two different mechanisms to manage the execution of BoT jobs on the cloud in real-time. The first one aims to detect and handle potential violations which are caused by undesired performance variation. The second one helps to estimate characteristics of unknown applications. These two mechanisms have been presented in [63, 64].

Chapter 7 presents the software framework which materialises the research presented in the previous chapters.

Chapter 8 presents and discusses the experiment performed in order to evaluate the proposed research and compare it with other existing methods. The results detailed in this chapter have partly presented and discussed in [63, 64].

Chapter 9 concludes this thesis by summarising its content and suggesting future work.

Chapter 2

Literature Review

This chapter reviews the existing work in research and development in scheduling the execution of BoT applications on the cloud. It begins with Section 2.1 which discusses all the existing world. The goal is to provide a detailed survey of the state of the art. Based on the surveyed publications, Section 2.2 constructs a taxonomy that covers different aspects and characteristics of a scheduling system. Then Section 2.4 presents a list of challenges and objectives that will be addressed by the research in this thesis. Section 2.5 summarises this chapter.

2.1 Related Work

In this section, we present and review existing papers, publications, and frameworks in scheduling the execution of BoT jobs in the cloud.

2.1.1 Overview of the Survey Methodology

This section focuses on reviewing the existing work on scheduling the execution of BoT application(s) on the cloud. The publications were manually searched and discovered via Google Scholar. To ensure the quality of the publication, we only selected works from major journals and respected conference venue such as IEEE Transactions on Services Computing, IEEE Transactions on Parallel and Distributed Systems, IEEE Conference of Cloud Computing (CloudCom), IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), IEEE International Conference on Cloud Computing (CLOUD), International Conference on Autonomic Computing (ICAC), IEEE International Conference on Utility and Cloud Computing (UCC), IEEE Transactions on Parallel and Distributed

Systems, ACM International Workshop on Data-Intensive Distributed Computing (DIDC), ect.

To ensure the relevance of the literature, all of the selected work must satisfy the following criteria:

- The application must be BoT applications, as a result, we will not consider workflow and user-facing applications.
- The execution must be performed fully or partly on the cloud, hence, other models of resource pool, such as grid computing and data centre, are not considered.
- The monetary cost must be considered since this is the unique challenge of the cloud environment.
- A cloud must be a black-box environment, which means a user may not have control over internal operations.

We managed to find 12 publications that satisfied all above criteria. We thoroughly analysed each selected work by looking for the answers for the following questions:

1. *In what environment is the work presented?* This question focuses on the characteristic of the cloud cluster, i.e. homogeneity and heterogeneity. Furthermore, it looks at the cloud providers, e.g. the employed pricing scheme, the applications to be executed, etc.
2. *What does the work aim to achieve?* This question aims to address what users want to achieve by executing their applications on the cloud, e.g. cost saving or performance improvement. We are also interested in the criteria which are used by the users to evaluate the success of the scheduling mechanism and the execution.
3. *How does the work achieve the objectives?* This question can be divided into two smaller ones:
 - (a) *By which approach can the solution, i.e. scheduling plan, be found?* Execution scheduling is an optimisation problem which aims to achieve the predefined goals by tuning and configuring different involved parameters. Hence, it is necessary to explore which methodologies adopted by the researchers to solve this optimisation problem.
 - (b) *What does the solution look like?* By this question, we want to address the activities that need to be taken in order to reconfigure the cloud cluster given a solution, or a scheduling plan, so that a users' goals can be met.

This section considers both homogeneous and heterogeneous cloud environments. We define a cloud environment to be homogeneous if every VM in a cloud cluster is of the same pre-defined instance type. We define a cloud environment to be heterogeneous if VMs of different instance types are available for an application.

2.1.2 Scheduling in a Homogeneous Environment

In homogeneous cloud environments, given that only one instance type is used to create VM, both the expected performance and pricing are the same on all available VMs. This simulates an ideal data center and simplifies optimisation.

Hybrid Clouds

In a hybrid cloud, both private and public clouds are used. Candeia et al. [21] propose a framework that schedules an application such that the deadline is met while monetary costs are minimised. The scheduling problem is modelled to simulate different scenarios by determining the number of public cloud VMs to be rented. The scenario that results in the highest profit is selected.

Bicer et al. [17] not only considered the monetary cost of renting cloud VMs, but also the overhead for synchronising the private and public clouds. For example, transferring data between two clusters. A mathematical model for predicting the execution time and the total cost of a hybrid cloud cluster is used. This model calculates the number of VMs to be rented from public cloud providers in order to satisfy the deadline or budget constraints.

Spot VMs

The performance of a cloud environment is normally improved by using preemptible VMs, also referred to as spot VMs. These VMs are obtained through a bidding process and may be terminated by the provider without any notice so that it is allocated to a higher bidder. The pricing of spot VMs is normally lower than on-demand VMs. However, the price may fluctuate dynamically over time based on the number of bidders. In this context, research in scheduling focuses on finding an effective bidding strategy for scheduling applications, or managing an application in the event of sudden termination, or both.

Yi et al. [71] develop a checkpointing mechanism that saves the progress of application execution at different points in time. This minimises the amount of execution time an application would lose if the VM is suddenly terminated. The framework monitors the bid prices in real-time in order to predict a termination. When such an event is predicted, the current process is saved.

Instead of using fixed bid prices, AMAZING [58] uses Constrained Markov Decision Process to find an optimal bidding strategy. The proposed approach takes deadlines into account and calculates the probability of different bidding options. When a predicted bidding price is too high, the framework saves the current process and waits for the next billing cycle to bid again.

Lu et al. [42] use spot resource for executing BoT jobs. The authors focus on the robustness of the system by using on-demand VMs, which are usually more expensive. However, the impact of termination is minimised since on-demand VMs are used as a backup. Whenever spot instances are terminated, the workload is immediately offloaded onto on-demand VMs.

Menache et al. [47] suggests switching to on-demand resources when there are no spot instance available to ensure the desired performance is always achieved. The decision to use on-demand resources is based on either a user-defined deadline or a policy to allocate a fixed number of on-demand VMs.

Reserved VMs

Costs in cloud environments can be reduced by using reserved VMs. This requires upfront payment for the VM, but is generally available at lower costs than on-demand VMs. This pricing scheme is useful if a user has a long term plan regarding the usage of the resource. Over provisioning, when a user reserves resources that is not entirely utilised may be a problem that will need to be tackled when reserving VMs. To mitigate this, cloud environments consisting of both reserved and on-demand VMs are employed. A significant proportion of the workload is assigned to reserved VMs to increase their running time. On demand instances may be added in order to temporarily handle resource bursts in the workload.

Yao et al. [70] presents an approach for satisfying job deadlines while minimising monetary cost by using both on-demand and reserved VMs. Heuristic algorithms that aim to pack as many jobs as possible into reserved VMs are proposed for increasing utilisation during the lease period. The remaining jobs are assigned to on-demand VMs. This resulted in achieving the desired performance at the lowest cost.

Shen et al. [56] use reserved VMs to optimise cloud environments to achieve cost savings. Integer Programming is used to model the assignment of tasks on VMs and the cost is minimised by determining the number of reserved and on-demand VMs. This scheduling problem is solved periodically in order to take into account newly submitted workloads.

2.1.3 Scheduling in a Heterogeneous Environment

Having presented existing research on scheduling in homogeneous cloud environments, we now consider research that makes use of a wider variety of VM types offered by providers. In this section, we present the existing methodologies in scheduling in heterogeneous environments. Compared to homogeneous cloud environments, a heterogeneous environment can be designed to offer more flexibility. This is conducive for applications that have a preference on the hardware specification or configuration of the VM. However, this is more challenging and the framework must take into account the trade-off between cost and performance for different VM types.

Public Clouds with On-demand VMs

There is research that focuses on executing a single BoT application. Oprescu et al. [49] present BaTS which is a budget-constrained scheduler for executing BoT job on the cloud. The problem is modelled as a Bounded Knapsack Problem and is solved using dynamic programming. The objective is to identify the number of VMs of each type for an application so that the total monetary cost does not exceed the budget constraint while not compromising performance. This research is extended to include the replication of tasks from running VMs onto idle VMs with the intention of decreasing the overall execution time [50].

Ruiz-Alvarez et al. [55] model the problem of minimising the cost of executing BoT jobs on the cloud using Integer Linear Programming. The execution of the application is divided into multiple intervals, each of which might correspond to one billing cycle (for example, one hour). In order to execute all tasks within a deadline, the number of tasks required to be executed within each cycle is estimated. Then the model selects the number of VMs so that all tasks are executed within the interval.

HoseinyFarahabady et al. [34] focus on the trade-off between performance and cost in scheduling BoT application on the cloud. This trade-off represents a user's preference. For instance, a user might want to achieve high performance while knowing that it would result in higher monetary cost. For this an algorithm using the Pareto frontier is employed by distributing tasks onto VMs of different types for execution.

There is also research that considers the execution of multiple BoT applications. Mao et al. [45] propose an approach to schedule the execution of multiple BoT jobs on the cloud with both deadline and budget constraints. In this approach, prior knowledge (i.e. the number of tasks of each job that a VM of a certain type could execute within an hour). The scheduling problem is then modelled as an Integer Programming problem and generates a plan with the

number of VMs of each type that can meet both deadline and budget constraints. Scheduling is performed periodically at the end of each billing cycle.

Lampe et al. [39] determine the mapping between BoT jobs and VMs so that all jobs can be executed within their deadlines with a minimum cost. Two different approaches are proposed for solving the problem. The first approach is modelled as a Binary Integer Problem and the second approach is based on heuristic algorithms. The latter approach repeatedly selects the cheapest VM to execute a list of jobs. Based on simulation studies, it is observed that the approaches require a significantly large amount of time to find a solution that can reduce the overall costs.

Gutierrez-Garcia et al. [32] present a policy-based approach for scheduling BoT execution on the cloud. A portfolio of 14 heuristic algorithms, each of which use a different task ordering and resource mapping policy. Experimental results indicate that the effectiveness of the algorithm depends on the characteristics of the workload.

Zou et al. [72] employ the Particle Swarm Optimisation (PSO) technique to execute multiple BoT jobs with deadline on the cloud while minimising the cost. Additional constraints in terms of the number of CPU cores and the amount of memory each job requires is considered. The traditional PSO technique is compared with a self-adaptive learning PSO (SLPSO) which has greater chances of finding either a better local optimal or even a global optimal.

OptEx [57] is a scheduling framework built on Apache Spark [11]. The framework does not require prior knowledge regarding the execution time of a job on a VM. Instead, this knowledge is acquired by profiling the execution of the job to construct a prediction model that estimates a job completion time based on the number of VMs in the environment. This estimation is used by a Non-linear Programming model to calculate the number of VMs of each type that are required to execute a job within a deadline. Workload assignment is performed using a built-in Spark mechanism.

Public Clouds with Spot VMs

Chard et al. [24] employ spot resources to achieve cost savings. An iterative process repeatedly bids for spot VMs to execute jobs. The maximum bid price is always kept lower than the on-demand resource price. However, if there is a job that is waiting for more than a predefined amount of time, it will be executed using the cheapest on-demand VM that can execute a task within the deadline.

Hybrid Cloud

Wang et al. [68] propose a framework that incorporates a heuristic algorithm which greedily assigns tasks to the best performing physical machine in a private cloud cluster. However, if no physical machine is available on the private cloud, the framework provisions VM from a public cloud based on a user defined budget.

Van Den Bossche et al. [65] use priority queues for scheduling BoT execution on the hybrid cloud. Each job is associated with a specific deadline and is added a queue when it is submitted. A mechanism that periodically scans the queue and estimates if the jobs can meet their deadlines using a private cluster is developed. If this is not possible, a job is moved onto a VM with the cheapest VM type.

Kang et al. [36] propose a framework that minimises the cost of a hybrid cloud for executing BoT job with deadline. Tasks are first considered to be executed on either private machines or existing cloud VMs since they do not incur monetary costs. However, if no existing resources are available, then the framework selects the cheapest VM which can execute the tasks within the deadline.

Duan et al. [28] employ game theory based scheduling on hybrid clouds. A multi-objective scheduling mechanism in which not only the makespan and monetary cost are minimised, but also the bandwidth and storage limit are not exceeded is proposed. A *K-player cooperative game* approach in which the players represent the applications that share the same private cluster is used. The algorithm aims to assign both private and public resources to each player so that the makespan and cost are kept to a minimum while the storage and bandwidth constraints are satisfied.

Pelaez et al. [53] argue that a scheduling framework should also take into account the variation of task execution during runtime. Therefore different approaches to estimate the task execution time during execution, which is used to constantly update the scheduling plan. Based on the updated information, tasks are assigned to the cheapest VMs that is can execute the tasks within the deadline by taking into account the variation of task execution time.

2.2 Taxonomy

This section identifies common themes, characteristics, requirements and challenges based on the publications surveyed in the previous section. This taxonomy is shown in Figure 2.1 by illustrating different characteristics and categories of a cloud scheduling framework.

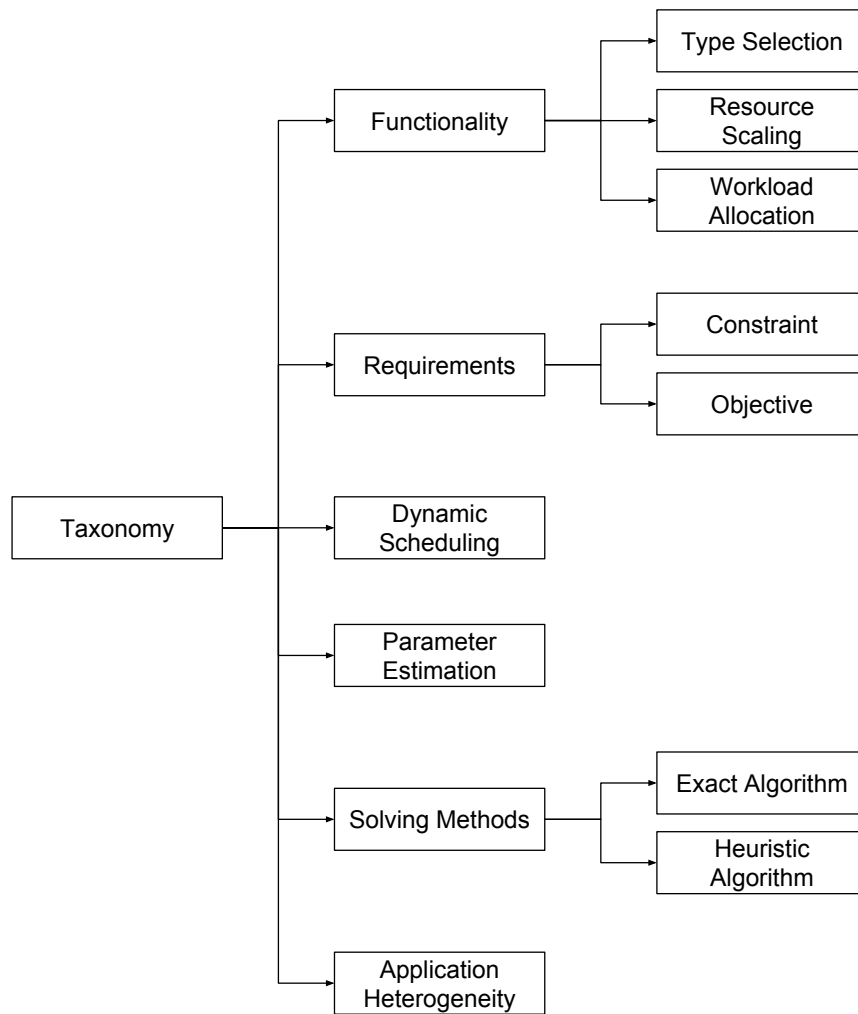


Fig. 2.1 Taxonomy of Cloud Scheduling Frameworks

2.2.1 Functionality

Scheduling BoT jobs on the cloud consists of three different types of functionality:

1. **Type selection** involves determining the combination of instance types that are used in the cloud cluster. Scheduling frameworks must be aware of the difference in not only prices, but also performance across all instance types.
2. **Resource scaling** calculates the number of VMs for each instance type. This functionality directly affects the incurred monetary costs (as more VMs are added to the cloud cluster for an application, the more expensive it will be).
3. **Workload allocation** functionality assigns workloads to the VMs running in the cloud cluster. The allocation needs to take into account the performance of a VM, its current state (i.e. knowledge of the workload already on a VM), and an application's requirements.

Obviously, type selection is not covered by methodologies that only support a homogeneous cloud cluster, for example, a cluster where all resources consist of the same VM type. However, a user must decide in advance which instance type is the most suitable for an application.

The most straightforward method for type selection is to run the application on a few or even all available instance types in order to determine which instance would be the most suitable (this is suggested by AWS [2]). However, this method can be not only time consuming but also expensive, due to the number of available instance types and applications. Researchers have proposed approaches to find the most suitable instance type of a user's applications. For instance, Varghese et al. [67] proposed a framework for instance type selection by matching VM performance, obtained via benchmarking, with an application's characteristics.

Some researchers have chosen to exclude workload allocation. Since Sidhanta et al. [57] built their OptEx framework on the Spark framework, they relied on the existing built-in mechanisms for workload allocation. On the other hand, Mao et al. [45] assumed prior knowledge regarding the fixed distribution of task allocations to each VM type. Finally, other researchers have chosen to describe the required performance of a job as the number of CPU hours and degree of parallelism, which eliminates the need for workload allocating [70, 47]

The **resource selection** process, which consists of resource scaling and type selection (if a cloud cluster is heterogeneous) can be performed separately from the workload allocation process. More specifically, the former will first determine the total amount of resources within the cloud cluster, then the latter will assign the workload to each VM. This approach usually

simplifies the scheduling process. For instance, many researchers have adopted a mechanism in which the workload is sequentially assigned to the first idle instance [17, 21, 71, 58, 49].

Finally, the majority of the existing work treats resource selection and workload allocation as a interrelated process [56, 50, 39, 32, 24, 42, 65, 36, 72, 28, 34, 68, 53]. In other words, the decision making process must consider not only the amount of required resources, but also the allocation of the workload onto those resources. More specifically, workload can only be assigned to VMs that are created. Similarly, VMs must only be created if there will be workload assigned to them. We believe that this is the most demanded requirement for running any application on the cloud, due to the incurred monetary costs which do not occur in any other cluster systems like grid or in-house data centres. As a result, it is necessary to provide users with a framework that helps them keep track of the cost in order to avoid unnecessary spending. Furthermore, achieving the desired level of performance is really important for running any type of application. For BoT applications, the desired performance is normally represented as a deadline. The violation of deadline constraints can lead to undesired consequences for the user such as financial penalties or a customer's dissatisfaction with the service [29].

2.2.2 Requirements

Requirements describe the criteria that are used to either determine if the execution of applications on the cloud is successful, or evaluate the quality of the execution. The requirements are set by a user and it is the goal of any cloud usage optimisation methodology to satisfy those requirements.

Requirements can be divided into two categories. **Constraints** are criteria that must be satisfied. Failure to satisfy those constraints is normally unacceptable. Occasionally, it is possible to violate a constraint given there is no other solution. However, the violation of such constraint, i.e. *soft constraint*, should be kept minimal. A constraint is normally represented as a threshold with specific value such as deadline or budget constraints [45, 17].

On the other hand, **objectives** are used to measure the quality of an application's execution on the cloud. For instance, if cost saving is an objective, the cheaper the execution is the better. An objective is normally represented as a goal to minimise, or maximise one or more parameters, e.g. minimise the monetary cost. Since using cloud resources incurs monetary costs, it is necessary for a user to be aware of the cost that he/she has to pay. As a result, one of the most common requirements for optimising cloud usage is *cost minimisation* [42, 68, 56, 71, 21]. On the other hand, *performance maximisation* is an objective, which aims to minimise an execution makespan [28, 32]. There are other more specific objectives

such as the trade offs between performance and cost [34]. In this research, a user was asked to provide a numerical value which, represented his/her preferences over cost or performance.

The majority of the existing work considers both constraints and objectives whilst making a scheduling decision. For instance, some researchers have focused on optimising cloud usage by *minimising the monetary cost while satisfying the deadline constraint* [65, 72, 53, 39, 36, 55, 57, 70, 58, 47], which aims to achieve the desired performance with the minimal cost. Researchers [50, 50] have addressed the problem of *performance maximisation with a budgetary constraint* with the objective of obtaining with the maximum performance within a budgetary constraint. Chard et al. aimed to help users to *acquire the desired amount of resources with the minimum cost* [24].

2.2.3 Dynamic Scheduling

Section 2.2.4 has presented the parameters used to schedule a BoT application on the cloud based on a given set of requirements. It is clear that the optimisation decision is made based on the value of those factors. However, these values can dynamically vary during runtime. As a result, the initial scheduling decision may become obsolete and inaccurate. In this section, we discuss the reason for parameter variations and how it is handled by the existing work.

Causes of Parameter Variation

The cost of on-demand and reserved resources remain for the most part constant. However, spot instance prices vary depending on the number of bidders and their bidding prices.

Performance-related parameters can vary during runtime because of one or a combination of the following reasons. As the task's execution time is usually an average value, its actual value can be either greater or less than this average value due to factors such as the input size. Furthermore, since a cloud is running on top of a heterogeneous cluster consisting of machines of different hardware types, it is possible for VMs of them same type to have different levels of performance since they are located on different hardware infrastructure [52, 51]. For instance, Ward and Barker [69] showed that the performance of different AWS's VMs of the same type widely varied up to 29%. Pettijohn et al. [54] and Chiang et al. [25] also reported such performance variation between VMs on the same type but on different data centres. Moreover, the performance of the same VM can change over time due to the workload of the physical machine. Leitner and Cito [40] showed that the performance of IO bandwidth on the same instance could fluctuate up to 30%. Moreover, Netflix reported that the performance degradation due to CPU stolen time could be high enough to make it more cost saving to replace a VM by a new one [10].

In order to effectively scale an application during runtime, a user may expect that resources should be available as soon as she requests for them. However, in reality, it normally takes a noticeable amount of time for cloud resources to be made available. This type of delay, called **instance start up time**, is common among all cloud providers and can vary from a few seconds up to a few minutes [44]. If a user does not take instance start up time into account, the requested resources may be available too late to handle the peak workload.

Dynamic Scheduling

As mentioned in the previous section, there are many reasons, which cause parameter variation. This section presents **dynamic scheduling**, which is performed during runtime in order to handle unexpected events that may result in requirements violation. Our discussion focuses on two aspects of dynamic scheduling: i) how it is triggered and ii) how it is performed.

The simplest way to trigger dynamic scheduling is to perform it periodically, normally right after the monitoring process which updates the parameters to reflect the current state of the execution [21]. On the other hand, dynamic scheduling can also be triggered periodically at the end of each billing cycle in order to decide if VMs can be terminated for cost saving purpose [45, 58].

Other researchers have chosen the more specific trigger. For instance, Bicer et al. [17] proposed to perform dynamic scheduling when each group of jobs was executed. On the other hand, since Oprescu et al. [50] aimed to exploit idle VMs, which have no tasks to execute while still being in the current billing cycle, to decrease makespan, the authors started the dynamic scheduling process when an idle VM was detected. Other researchers have proposed to dynamically reschedule when the requirements are predicted to be violated. The potentially violated requirements can be constraints such as deadline [36, 65, 24] or budget [49]. If spot instances are used, rescheduling is required in case of unexpected termination [71, 42]. Finally, rescheduling should be performed when the requirements change [28].

Dynamic scheduling can be performed by simply re-running the optimisation process given the updated parameters [17, 21, 45, 58, 71, 49, 36, 28]. However, since it can be time consuming to perform the whole optimisation process again, there are other, simpler approaches. For instance, some researchers have focused on re-allocating tasks between VMs in order to improve performance [50] while others have proposed approaches to dynamically resize the cluster at runtime [65, 24, 42].

2.2.4 Parameter Estimation

After understanding the functionality, it is necessary to be aware of not only the parameters or factors, which are involved in the decision making process, but also their availability.

Monetary Factor

Since a user must pay in order to use cloud resources, the first and foremost parameter that he or she needs to be aware of is the price of cloud resources. As discussed earlier, there are three popular pricing schemes for cloud resources: on-demand, spot, and reserved resources. The price of on-demand and reserved resource is always available. However, spot resource pricing is unknown since it changes over time depending on the number of bidders and their bidding prices. As a result, some researchers have decided to set a fixed bidding price, which is lower than the cost of the same amount of on-demand resource.

Performance Factor

The performance factor defines the performance of a specific application running on a VM instance type. Notably, the performance factor is specific to each unique pair of application and instance type. For instance, **task execution time** is the time it takes for a VM to execute one task of an application. By using task execution time, researchers can have a fine-grained control and view of an execution. As a result, this type of parameter is used by the majority of the existing research [65, 21, 28, 34, 36, 47, 49, 50, 53, 72]. Other researchers have used not only the task execution time, but also the data transferring time, which denotes the amount of time it takes to transfer data between private and public clouds [17]. On the other hand, the research of Sidhanta et al. [57] represented performance in a more coarse-grained way by using **job execution time**, i.e., **makespan**.

The performance factor can also be indirectly represented using different forms. For instance, some existing work used **resource capacity**, i.e. the number of tasks can be executed by each instance type within a billing cycle, as the performance factor [45, 55]. On the other hand, Chard et al. [24] used the queueing or waiting time to indicate that a task could not wait to be executed more than a certain amount of time.

Resource demands can also be used to indirectly represent the performance. More precisely, a resource demand indicates that a desired performance can be achieved if a certain amount of resource is allocated to execute an application. Resource demand can be represented as the number of VMs required by the application in each billing cycle and has been used by some researchers as a performance factor [39, 56, 68]. On the other hand, other

researchers have represented a resource demand as the number of VM hours required to execute an application [58, 70, 71].

Most of the time, researchers assume that the performance parameters are available prior to the optimisation process. However, other researchers have decided not to make this assumption and incorporated the process to estimate the performance parameters as a part of the optimisation process. For instance, Oprescu et al. [49] performed a sampling execution, in which a portion of a job was executed on VMs of all available types in order to estimate the task execution time. On the other hand, Sidhanta et al. [57] estimated the job execution time using profiling techniques.

2.2.5 Solving Methods

Optimising the execution of a BoT job on the cloud can take two algorithmic approaches as presented in literature. The first approach is referred to as Exact algorithms and the second approach is heuristic based algorithms which are considered in this survey. These algorithms typically generate a **scheduling plan** for the BoT job on the cloud.

Exact Algorithm

The approach of using **exact algorithm** is proposed in literature. The scheduling problem is represented as a mathematical model and then solved using an existing solver to find the optimal solution [43].

The popular approach is using Linear Programming, in which a set of linear formulas are used to model the problem [58]. Integer Linear Programming is another choice which requires all decision variable to be integers [45, 55, 21], or Binary Integer Programming in which the decision variables are binary values (either one or zero) [39]. Other exact algorithms that are reported in literature include Non-Linear Convex Optimisation Problem [57].

Exact algorithms guarantee an optimal solution. However, they require a significant amount of time to solve and obtain a scheduling plan. Therefore, these are not suitable for real-time system in which a decision must be made in a timely manner.

Heuristic Algorithm

Researchers have adopted the **heuristic algorithm** approach to reduce time taken by exact algorithm. Heuristic algorithms aim to find a solution in a relatively shorter amount of time, but do not guarantee global optimality. For generating scheduling plans there may be little difference between local and global optimal solutions.

One of the simplest heuristic approach is the greedy algorithm which makes the best decision possible given knowledge of the current state. For instance, scheduling algorithms that iteratively select the cheapest instance type during each iteration have been proposed [36, 65, 24, 53]. Greedy algorithms which select the best solution given the current states of multiple criteria are proposed [68].

Heuristic algorithms can incorporated rule-based approaches in which the scheduling decision is based on a set of pre-defined rules. For instance, Gutierrez-Garcia et al. [32] use a set of rules defining the order and allocation of tasks to VMs. Other research use rules to acquire resources for achieving the desired performance [17, 42, 56, 47].

Another heuristic algorithm is based on dynamic programming, which breaks the scheduling problem into smaller sub-problems [49, 50, 70]. Meta-heuristic approaches, such as Particle Swarm Optimisation, is a general purpose approach which is employed in this space [72].

Custom heuristic algorithms are also employed. For instance, Yi et al. [71] use approximation techniques to predict the termination time of spot VMs. The scheduling algorithm of Duan et al. [28] is based on the game theory approach. The cloud bursting approach proposed by HoseinyFarahabady et al. [34] uses the concept of Pareto optimality.

2.2.6 Application Heterogeneity

Application heterogeneity indicates the variety of applications to be scheduled for execution on the cloud. In other words, methodologies that do not support application heterogeneity are *only able to schedule a single application*. On the other hand, application heterogeneity is supported when *multiple BoT applications are scheduled at the same time*. Which means that the cloud cluster is shared between multiple applications, each of which performs differently, e.g. has a different task execution time, on the same hardware specification. Supporting application heterogeneity is challenging since each application prefers different VM type. For example, a computation-intensive application prefers a CPU-optimised machine to a memory-optimised instance. As a result, a scheduling mechanism must take into account instance type preferences of all applications in order to select a suitable combination of resource types in a cloud cluster.

Some researchers have supported application heterogeneity by splitting a cluster into smaller sub-clusters, each of which execute only one application [47, 24, 65]. In other words, there is not resource sharing between jobs, i.e. each VM only executes tasks of one job. However, this approach is not efficient since it limits the flexibility of a cloud cluster. For instance, if only few tasks of a job are assigned to a VM, it would be wasteful to not use that VM to execute tasks of other jobs.

Resource sharing between applications have been investigated by other researchers. The simplest approach is to predefine the distribution of jobs on each instance type, as adopted by Mao et al. [45]. On the other hand, other researchers have proposed to assign a group of jobs, instead of just a single one, to be executed on a sub-cluster [70, 34]. The authors have developed mechanisms which create a group of jobs so that the resource wastefulness in each sub-cluster can be minimised. Finally, the most of complicated but also most efficient approach is to assign all jobs to all instances without predefined task distribution or job grouping [56, 39, 32, 72, 68, 36, 28, 53]. This approach results in a workload assignment in which each VM receives a different task distribution. As a result, it can potentially maximise resource efficiency. However, this approach is challenging since it may have to consider a countless possibilities of workload allocation between jobs and VMs.

2.3 Discussion

Based on the survey of existing work presented in Section 2.1 and the taxonomy developed in Section 2.2, this section will summarise the current trends in this research area. Further, we propose research directions that will improve the usage of cloud resources.

2.3.1 Current Trends

Table 2.1 summarises all methods reviewed in this paper and categorises them using our proposed taxonomy framework.

Functionality: Resource scaling is a key feature that is supported by all research. It is noted that type selection can only be obtained on frameworks that support heterogeneous cloud environments. The majority of existing research supports workload allocation since it provides fine-grained control over task distribution between jobs and VMs.

Requirement: Performance constraints (for example, deadline constraint) with cost objective (for example, cost minimisation) is the most popular requirement. The main reason for this is to achieve a desirable level of performance while being aware of the monetary cost incurred in real time. This is unique to using cloud resources in contrast to grid or cluster computing.

Dynamic Scheduling: Less than half of the surveyed research support dynamic scheduling. For the sake of simplicity a number of papers assume that the performance of cloud computing resources remain unchanged during execution. However, this assumption does not hold on real clouds and when heterogeneous resources are used.

Table 2.1 Taxonomy of BoT Scheduling Methodologies

	Functionality			Requirement				Dynamic Scheduling	Parameter Estimation	Solving Method		Application Heterogeneity
				Constraint		Objective				Exact	Heur	
	Type Selection	Resource Scaling	Workload Allocation	Cost	Perf	Cost	Perf					
[58]		X	X		X	X		X		X		
[21]		X	X		X	X		X	X			
[47]		X	X		X	X		X			X	
[70]		X	X		X	X				X	X	
[56]		X	X		X	X				X	X	
[65]	X	X	X		X	X		X		X	X	
[39]	X	X	X		X	X			X	X	X	
[72]	X	X	X		X	X				X	X	
[57]	X	X			X	X			X			
[24]	X	X	X		X	X		X		X	X	
[68]	X	X	X		X	X				X	X	
[53]	X	X	X		X	X		X		X	X	

Parameter Estimation: Only four papers we surveyed support parameter estimation. This reflects the common belief that the necessary parameters can be obtained prior to executing a job. However, obtaining this information prior to execution may not be feasible given that a cloud environment is usually shared between many users.

Solving Method: The majority of research adopt a heuristic approach, since this approach finds solutions faster than alternate exact algorithm approaches. Approaches that are required for real-time systems need to converge on a scheduling plan as quickly as possible. Although exact algorithms are guaranteed to find an optimal solution, they are not widely used since they are time consuming and is adopted in only seven research papers we surveyed.

Application Heterogeneity: The vast majority of existing work supports application heterogeneity - scheduling the execution of multiple applications that perform differently on the same type of VM. This reflects the trend in developing cloud environments that can be shared between different users with different applications.

Based on Table 2.1, we summarise the current trends in optimising the usage of cloud resources as follows:

- **Supporting heterogeneous cloud environments:** Cloud computing environment need to be flexible in accommodating multiple applications with diverse needs by supporting VMs with different hardware specification. Therefore, cloud environments are supporting VMs of different instance types.
- **Minimising monetary cost while ensuring the desired performance:** Monetary cost is one of the most important concerns of cloud users. Hence, there is research to produce scheduling plans that keeps the cost as low as possible without sacrificing the desired quality of service.
- **Handling unexpected events at runtime:** In any large scale and real-time system, unexpected events, such as missing information or performance variation during runtime, inevitably occur. Hence, mechanisms are put in place to detect and handle such events in order to prevent, or at least minimise their impact.
- **Using heuristic algorithms:** Heuristic algorithms are popularly used in the space of optimising resources for the execution of BoT applications. This is because they can produce timely results although there is a trade-off with the optimality of solutions obtained.

2.4 Requirements Analysis

It can be seen from Table 2.1 that there is no existing work that covers all aspects of optimising the usage of cloud resources for executing BoT application. For instance, even though the work of Oprescu et al. [49, 50] supported both dynamic scheduling and parameter estimation, the authors only focused on scheduling a single application on the cloud. Similarly, none of the existing work that supports application heterogeneity is able to perform parameter estimation. Furthermore, none of the papers surveyed by us investigates combining both exact and heuristic algorithms. This thesis aims to address those shortcomings.

Based on the review of the existing work and the resulted taxonomy, we are able to construct the set requirements that need to be addressed by the research in optimising the execution of BoT job on the cloud. These requirements are used as a guide for the research presented in this thesis.

2.4.1 Heterogeneous Environment

In order to make full use of the wide variety of instance types provided by the providers, we believe that our framework must support the heterogeneous cloud cluster. By doing so, we will be able to create a flexible cluster which can dynamically adapt to the demand that keeps changing in real-time.

Our research also aims to support the execution of multiple applications, i.e. application heterogeneity. Furthermore, those applications will be submitted in real time, which means the workload that the cluster must handle change dynamically. We believe that this assumption reflects a cloud cluster in real world which has to handle a variety of different submitted applications.

To support the heterogeneous environment is very challenging since it requires a full awareness of the diversity of not only the resource types but also the applications.

2.4.2 Satisfying Deadlines While Minimising the Monetary Cost

In this research, we choose to focus on satisfying the deadline constraints while minimising the monetary cost. We believe that this is the most demanded requirement for running any application on the cloud due to the incurred monetary cost which does not happen in any other cluster system like grid. As a result, it is necessary to provide users which a framework that helps them keep track of the cost in order to avoid unnecessary spending. Furthermore, achieving desired performance is really important for running any type of application. For BoT application, the desired performance is normally represented as a deadline. The violation

of deadline constraints can lead to undesired consequences such as financial penalties or a customer's dissatisfaction with the service [29].

Satisfying deadline constraint and minimising monetary costs are inter-related requirements. More precisely, the deadline constraint can only be satisfied when the cloud VM cluster achieve a desired capacity by acquiring, and paying for, cloud resources. This problem is further complicated when there are multiple jobs, each of which not only has different deadline but also performs differently on the same VM type.

2.4.3 Flexible Execution

Since our goal is to develop a real time system, it is very important to be prepared for any unexpected events. For instance, since parameter variation is unavoidable, the proposed framework must be able to constantly re-evaluate its decision and make a new one when necessary. More specifically, we plan to develop a dynamic scheduling mechanism which is able to detect any potential issues during an execution and perform necessary action to reduce, or completely prevent, those issues.

Most of the current research assume the prior knowledge regarding task execution time. However, this assumption may not be applied since a user may submit a new application to a cluster for execution. As a result, it is necessary for have a mechanism which estimates task execution time during runtime instead of relying on existing knowledge.

2.4.4 Trade-off Aware Solving Methods

As mentioned in Section 2.2.5, there is a trade-off between using exact and heuristic algorithms. Hence, a part of this research is to investigate that trade-off. More precisely, we are going to develop different scheduling approaches ranging from exact to heuristic. Intensive experiments are performed in order to compare the differences in optimality and solving time between each approach.

Such kind of work was already performed by Lampe et al. [39]. However, we believe that this should be investigated more thoroughly in order to not only compare different approaches but also find out the affect of other factors on the their performance.

2.5 Chapter Summary

This chapter has presented and reviewed the existing work in scheduling the execution of BoT jobs on the cloud. The classification framework has been constructed in order to better analyse the work and provide a systematic view. Finally, we present a set of requirements

which need to be taken into account while developing a new research approach for optimising the execution of BoT jobs on the cloud. These requirements will be addressed by the rest of this thesis.

Chapter 3

Mathematical Representation of the Research Problem

This chapter constructs a complete but concise mathematical model that represents the problem of optimising the usage of cloud resources. The first goal of this chapter is to provide to the reader a coherent view of not only multiple parties involved but also the relationships and constraints between them. Secondly, the mathematical notations and formulas introduced in this chapter will be used by the later ones in order to create the concise representation and understandable representation solutions for the complex scheduling problem in this thesis. The portion of this chapter has been published as "Minimising the Execution of Unknown Bag-of-Task Jobs with Deadlines on the Cloud" [63]. The research is then extended and published as "Algorithms for optimising heterogeneous Cloud virtual machine clusters" [64].

This chapter is structured based on the incremental approach. In other words, the problem is broken into different components, each of which is represented as one section and is built on top of the previous ones. Section 3.1 introduces a mathematical representation of the cloud environment, the applications and their jobs. Then the execution of jobs on VMs are mathematically modelled in Section 3.2. All of the mathematical representations are then combined to construct an optimisation problem of minimising the monetary cost of executing BoT jobs with deadlines on the cloud in Section 3.3. Section 3.4 summarises this chapter.

3.1 Environment Modelling

This section introduces and mathematically models all the involved parties, namely the cloud and the application(s), and the relationship between them.

Let T be a set of **instance types** offered by cloud providers. Each instance type is a blueprint that defines the virtualised hardware specification of a VM (e.g. number of CPU cores, amount of memory and storage. . .). Moreover, VMs of the same instance type cost the same amount of money for each **Accountable Time Unit (ATU)**, e.g. hour. The cost per ATU of a VM type $t \in T$ is denoted as p_t . Notably, cloud providers can update their offered instance types in order to add new types, remove obsolete ones, and update the specification and/or their prices once or twice a year. However, due to the rarity of those events, we assume that the set T and all instance types in it are unchanged.

Let A be a set of **BoT applications** executed on the cloud. It is possible for users to have full knowledge which applications are executed on the cloud. This is particularly true in a data centre environment in which the same set of applications are executed repeatedly and/or periodically. It is also possible to execute new/unknown applications which have never been executed before. Which means that new applications can be added to A during runtime.

Let $e_{a,t}$ be a **task execution time** which is the average amount of time in second taken to execute one task of an application $a \in A$ on an instance of type $t \in T$. In other words, task execution times are the mapping that represents the performance of users' applications on cloud providers' instance types. It should be noted that *task execution times are not exact values but average ones*. Which means that the actual values can vary during runtime.

Let J be a set of **jobs** that are submitted to be executed on the cloud. Each job $j \in J$ belongs to one application $a_j \in A$. The reason behind separating applications and jobs is to make it possible for the same application to be executed multiple times, each of which corresponds to one job. In fact, this is a common practice in industrial data centres whose workloads are mostly recurring [29, 19].

A job j can be divided into n_j tasks. Which means that a job is executed when all of its tasks are executed. Let s_j be an instant of time at which a job j is submitted, i.e. **submission time**. Similarly, d_j is a job's **deadline**, i.e. an instant of time by which all tasks of j must be completely executed. It is obvious that a job's deadline must be in the future compared to its submitted time, i.e. $d_j > s_j$. The amount of time from a job's submission to its deadline is called **available execution time of a job** and denoted as e_j :

$$e_j = d_j - s_j \quad (3.1)$$

In other words, all of its tasks must be fully executed within in e_j seconds.

Let V be a set of cloud VMs or instances. For each instance $v \in V$, its type is $t_v \in T$. A task execution time of a job on a VM is equal to a corresponding value of a job's application on an instance type of a VM, i.e. $e_{j,v} = e_{a_j,t_v}$.

In order to make the size of V fixed, we assume that there is an upper bound for a number of VMs of each instance type that a user can have. In fact, many cloud providers have enforced this policy in order to limit a size of a user's cloud cluster. For instance, AWS allows a user to create only 10 instances for each type by default. Even though this value can be increased upon request, there is always a limit for resources that a user can have. Hence, we assume that V contains the maximum number of VMs of each type that a user is allowed to have.

The **creation time** and **termination time** of an instance $v \in V$ are denoted as cr_v and te_v , respectively. Let $r_v \geq 0$ denote a VM's **running time**, which is the difference between its termination time and a creation time.

$$r_v = te_v - cr_v \quad (3.2)$$

When an instance's termination time equals to its creation time, i.e. the running time is zero, an instance is not created.

It should be noted that it takes a certain amount of time of a newly created instance to be ready for execution. Hence, let β be an **VM creation overhead** of any instance. Similar to task execution time, creation overhead is an average value instead of an exact one. It has been reported that the actual creation overhead for each VMs can vary significantly [44].

An instance is only ready to execute workloads after β seconds after its creation. Let re_v denote an instance of time at which an instance is ready to execute workload, i.e. $re_v = cr_v + \beta$.

The Figure 3.1 presents an example for an instance's life. It is created at cr_v . After β second, an instance is ready at re_v . An instance is terminated at te_v . Finally, a running time of an instance is r_v , which is the amount of time from cr_v to te_v .

Assuming that each ATU consist of 3600 seconds, i.e. one hour, its number of ATUs can be calculated by dividing its running time to 3600 and then rounding up to the nearest ATU:

$$atu_v = \lceil \frac{r_v}{3600} \rceil \quad (3.3)$$

The cost of an instance can be calculated by multiplying its number of ATUs to the cost per ATU of its type:

$$c_v = atu_v \times c_{it_v} \quad (3.4)$$

Finally, the total cost of using cloud computing to execute all tasks is the summation of the cost of all VMs:

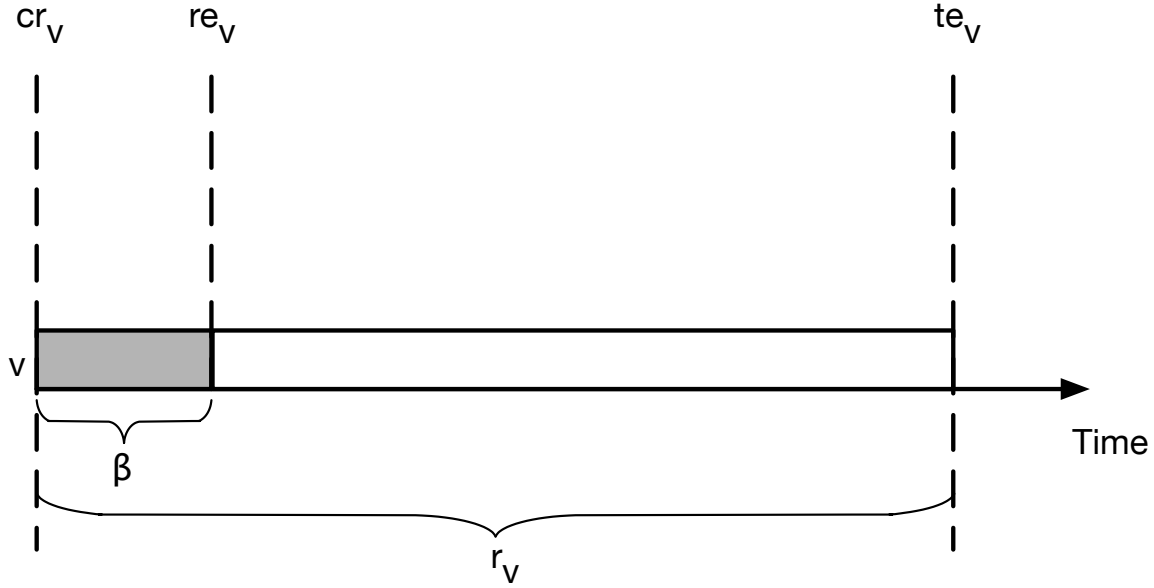


Fig. 3.1 Example of the Life of an Instance

$$c = \sum_{v \in V} c_v \quad (3.5)$$

3.2 Job Execution Modelling

The previous section has presented the mathematical representation of the cloud environment which includes the applications and their jobs. In this section, we attempt to model the execution of jobs on VMs.

After being submitted, a job's tasks must be assigned to one or more VMs to be executed. The assignment of some tasks of a job $j \in J$ to an instance $v \in V$ is represented as a **workload** $w_{j,v}$. In other words, each workload $w_{j,v}$ is a subset of a job j that is assigned to be executed on an instance v , i.e. $\bigcup_{v \in V} w_{j,v} = j$. Moreover, a workload is empty, i.e. $w_{j,v} = \emptyset$ when there is no task of j that is assigned to be executed on v .

Given a workload w , its job is denoted as j_w . Similarly, an instance that a workload is assigned to is v_w .

On an instance, the start and finish times of a workload w are denoted as st_w and fi_w respectively. Which means that an instance v_w starts executing tasks of a job j_w at the time st_w and finishes an execution at the time fi_w . Thus, the execution time of a workload is the difference between its finish time and start time:

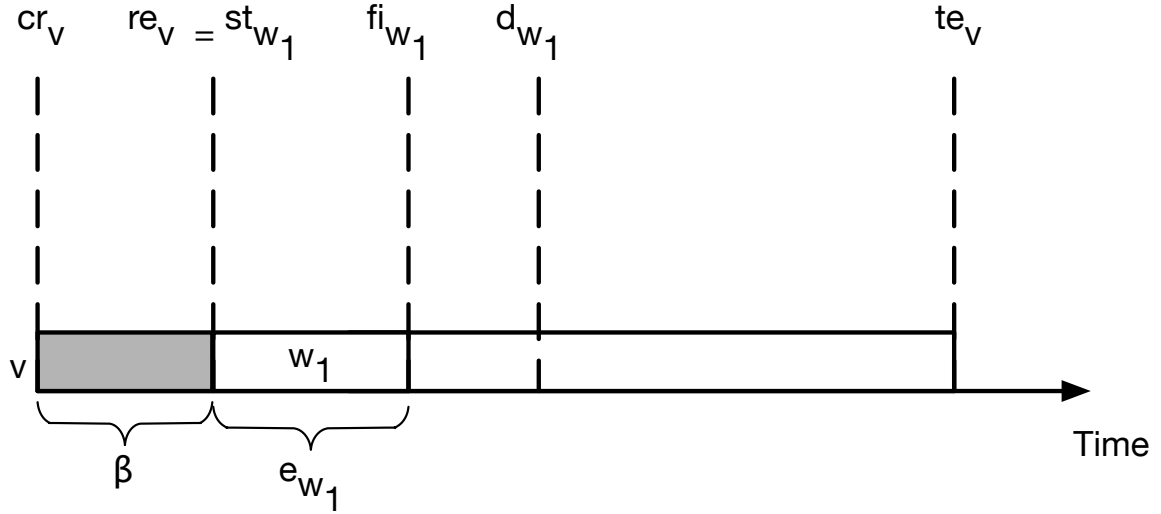


Fig. 3.2 Example of a Workload Assigned to an Instance

$$e_w = fi_w - st_w \quad (3.6)$$

The execution time of a workload must be non-negative. If an execution time is zero that means no task of a job is assigned to be executed by an instance.

As mentioned earlier, when an instance is created, it takes a certain amount of time, i.e. β , for an instance to be booted into ready state. In other words, a workload cannot start before a ready time of an instance:

$$st_w \geq re_{v_w} = st_{v_w} + \beta \quad (3.7)$$

Similarly, it is impossible for an instance to execute any workload after being terminated, i.e. after its termination time:

$$fi_w \leq te_{v_w} \quad (3.8)$$

Since each workload is tied to a specific job, its deadline is equal to its job's deadline, i.e. $d_w = d_{j_w}$. In order to avoid deadline violation, all workload finish times must not exceed their deadlines:

$$fi_w \leq d_w \quad (3.9)$$

The example of a workload assigned to an instance is illustrated by Figure 3.2: a workload w_1 is assigned to an instance v . As a workload is executed right after an instance is ready,

its start time is equal to the ready time of an instance, i.e. $st_{w_1} = re_v$. On the other hand, a workload must finish before not only its deadline (i.e. $fi_{w_1} < d_{w_1}$) but also a termination time of an instance (i.e. $fi_{w_1} < te_v$). Finally, a workload execution time is from the start to the finish times, i.e. $e_{w_1} = fi_{w_1} - st_{w_1}$.

We assume that an instance can execute only one task of one workload at the time. Which means, on the same instance, if a workload starts to be executed, it must be finished before another one starts. In other words, given two workloads assigned on the same instance, if a start time of the first one is less than the second one's, its finish time must also be less than or equal to the start time of the second workload:

$$st_{w_1} \leq st_{w_2} \iff fi_{w_1} \leq st_{w_2}, v_{w_1} = v_{w_2} \quad (3.10)$$

The number of tasks of a job in a workload can be calculated by dividing its execution time to the task execution time of an application of a workload's job on a type of its instance:

$$n_w = \lfloor \frac{e_w}{e_{a_{jw}, t_{vw}}} \rfloor \quad (3.11)$$

The floor function is used in Formula 3.11 since each tasks must be fully executed. In other words, task execution pre-emption is not supported.

Let W_j denote all workloads of a job j . Since a job, i.e. all of its tasks, must be fully executed before its deadline, the summation of number of tasks of all workloads belonging to a same job must be equal to its total number of tasks:

$$\sum_{w \in W_j} n_w = n_j \quad (3.12)$$

Since execution overlap in the same instance is not allowed, it is possible to calculate a running time of a VMs based on the execution time of its workloads.

Let W_v be all workloads assigned to an instance $v \in V$, its running time can be calculated by adding the creation overhead to the summation of all assigned workloads:

$$r_v = \begin{cases} \beta + \sum_{w \in W_v} e_w, & \text{if } \sum_{w \in W_v} e_w > 0 \\ 0, & \text{otherwise} \end{cases} \quad (3.13)$$

Formula 3.13 states that a running time of a VM is greater than zero if and only if it actually executes one or more workload. On the other hand, if no workload is assigned to it, i.e. the summation of execution time is equal to zero, an instance is not created and its running time is zero as well. This condition aims avoid creating and paying for idle instances, which do not execute any tasks at all.

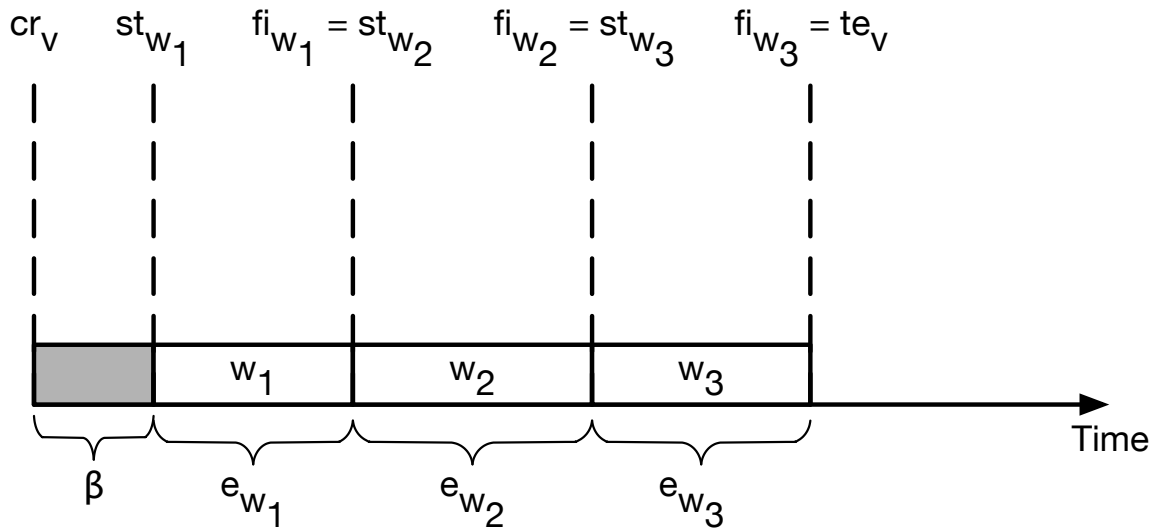


Fig. 3.3 Example of a VM Life Based on Assigned Workload

Figure 3.3 presents an example in which a VM's life can be determined by its assigned workloads. It can be seen that the terminated time of an instance is the finish time of the last workload. As the result, an instance's running time can be calculated by adding β to the sum of all workloads' execution times.

3.3 Problem Modelling

Based on the mathematical formulas constructed in the earlier sections, the problem of statically scheduling multiple BoT jobs with deadlines on the cloud in order to achieve minimum cost is presented by Model 3.14. Verbally, it can be described as *to calculate the creation and termination times of all instances, the start and finish times of all workloads so that all jobs can be executed before their deadline while the monetary cost is kept minimal.*

$$\text{minimise} \quad c = \sum_{v \in V} c_v \quad (3.14a)$$

$$\text{subject to} \quad c_v = atu_v \times pit_v \quad (3.14b)$$

$$atu_v = \lceil \frac{r_v}{3600} \rceil \quad (3.14c)$$

$$r_v = \begin{cases} \beta + \sum_{j \in J} e_{w_{j,v}}, & \text{if } \sum_{j \in J} e_{w_{j,v}} > 0 \\ 0, & \text{otherwise} \end{cases} \quad (3.14d)$$

$$e_w = fi_w - st_w \quad (3.14e)$$

$$st_w \geq cr_{v_w} \beta \quad (3.14f)$$

$$fi_w \leq te_{v_w} \quad (3.14g)$$

$$fi_w \leq d_j = d_{j_w} \quad (3.14h)$$

$$st_{w_1} \leq st_{w_2} \iff fi_{w_1} \leq st_{w_2}, v_{w_1} = v_{w_2} \quad (3.14i)$$

$$n_w = \lfloor \frac{e_w}{e_{a_{j_w} t_{v_w}}} \rfloor \quad (3.14j)$$

$$\sum_{w \in W_j} n_w = n_j, \forall j \in J \quad (3.14k)$$

The breakdown of Model 3.14 is as following:

- Formula 3.14a is the objective to minimise the total monetary cost, which is the summation of the cost of each individual VM.
- Formula 3.14b is used to calculate the costs of each VM.
- Formula 3.14c calculates the number of ATUs of each VM based on its running time.
- Formula 3.14d shows the calculation of a running time of each VM. If a VM it not used, its running time is 0. Otherwise, the running time is the summation of all workload execution times on that VM and the creation overhead time.
- Formula 3.14e calculates an execution time of each workload, which is the difference between its finish and start times.
- Formula 3.14f states that a start time of a workload must be greater than or equal to the ready time of an instance, i.e. a workload cannot start executing on an instance when it is not ready.

- Formula 3.14g states that the finish time of a workload must be less than or equal to the termination time of an instance, i.e. a workload cannot be executed on an instance when it is already terminated.
- Formula 3.14h states that the finish time of a workload must be less than or equal to its job's deadline. This is an important constraint which aims to enforce a desired performance.
- Formula 3.14i prevents execution overlap within each instance by requiring each workload to finish before another one, on the same VM, starts. In other words, there is at most one workload in execution on an instance at any time.
- Formula 3.14j calculates the number of tasks executed in each workload by dividing its execution time to a task execution time.
- Formula 3.14k requires that the total number of tasks executed in all workloads of the same job must be equal to a job's number of tasks, which means that all tasks must be executed.

3.4 Chapter Summary

In this chapter, we have introduced the mathematical representation of the cloud environment, the application and their jobs. Second, the execution of jobs on VMs has been represented as workloads, each of which consist of the number of assigned tasks, the start and finish times. The result of this chapter is Model 3.14 whose solution describes the scheduling of an execution on the cloud so that all jobs are executed within their deadlines while the total monetary cost is minimised. Another contribution of this chapter is the set of annotations and formulas which help to present the scheduling problem that we are trying to investigate and its solution in a concise way. As the result, those formulas and annotations will be frequently used by the next chapters.

Chapter 4

Workload Assignment

In the previous section, we have introduced the concept of a workload, which represents an execution of a job's tasks on an instance. However, a workload cannot be simply constructed by assigning a job's tasks to a VM. On the contrary, the process of constructing a workload, named **workload assignment**, needs to consider different factors of not only a job but also a VM. For instance, if a VM receives too many tasks of a job, it may fail to execute all of them within the predefined deadline. On the other hand, if too little tasks of a job are assigned to each VM, it will require many VMs to execute a job, which will result in unnecessary cost. Not only a current job but also other ones which are already assigned to a VM must be taken into account. For instance, if a VM has received many tasks from different jobs, it may no longer be available to execute any more tasks. Hence, it is necessary for a scheduling approach to have a mechanism which is able to effectively assign tasks for many jobs to a given set of VMs, which is the main focus of this chapter.

The effectiveness of the workload assignment is based on two main criteria. First of all, it must be able to assign as many tasks from a job to a VM as possible. Secondly, it must avoid both deadline violation or incurred monetary cost. It should be noted that the workload assignment process requires a set of instances to be given to it, which means that it is not involved in resource selection, which will be discussed in the later chapter.

This chapter takes a bottom-up approach by first introducing a sets of small algorithms (Section 4.1) which are later combined to construct the complete workload assignment process (Section 4.2). This chapter is summarised by Section 4.3.

4.1 Utility Functions

This section introduces a set of four **utility functions** that provide common functionalities and are used to construct not only the workload assignment but also other algorithms presented later in this thesis. The name utility function is inspired by Java's `util` classes [18].

4.1.1 Finding Preceding and Succeeding Workloads

Each cloud VM can execute many workloads sequentially. Hence, a new workload can be assigned to an instance with existing workloads. It is necessary to find a place in the waiting queue to put a new workload into without resulting in any deadline violations. The idea is to split the existing workload into two sub-lists, the first of which contains all workloads that need to be executed before a new one. The remaining existing workloads, which can be executed after a new workload, is put in the second sub-list.

This process is implemented as a function named *FIND_POSITION* and is presented by Algorithm 4.1. Its inputs are a job from which a new workload is created and a list of existing workloads within a VM. The workloads are sorted by deadline in ascending order. Which means that a workload with a earlier deadline is executed before those with later deadlines.

Algorithm 4.1 Find Position

```

1: function FIND_POSITION( $j, W_v$ )
2:    $W_p \leftarrow \emptyset$ 
3:   for  $w \in W_v$  do
4:     if  $d_w \leq d_j$  then
5:       Add  $w$  to  $W_p$ 
6:     else
7:       break
8:    $W_s \leftarrow W_v - W_p$ 
9:   return ( $W_p, W_s$ )

```

The algorithm loops through all the given workloads and adds those with deadlines smaller than or equal to a job's deadline to the list of preceding workloads, denoted as W_p (Line 5). When a workload whose deadline is greater than a job's deadline is found, the loop is terminated (Line 7). This is due to the fact that workloads are sorted in ascending order, which means the deadlines of all remaining workload must also be greater than a job's deadline. As a result, the set of succeeding workloads, denoted as W_s , are those not in the set of preceding ones (Line 8).

4.1.2 Calculate Permissible Delay

When a new workload is put and executed before existing ones in the same instance, the execution of those existing ones is delayed. This delay can result in a deadline violation when an execution of an existing workload is delayed for too long. Hence, it is necessary to calculate a **permissible delay** for all succeeding workload in order to avoid deadline violation.

A permissible delay of a workload is the amount of time that a workload execution can be delayed without resulting in a deadline violation and can be calculated as the difference between its deadline and finish time. Hence, the permissible delay of a set of workloads is the amount of time to delay their execution without resulting in violation for any of them, its calculation is presented by Algorithm 4.2.

Algorithm 4.2 Calculate Permissible Delay

```

1: function CALCULATE_DELAY( $v, W_s$ )
2:    $delay \leftarrow \max(NOW, re_v) - te_v$ 
3:   for  $w \in W_s$  do
4:      $delay' \leftarrow \min(te_v, d_w) - fi_w$ 
5:     if  $delay > delay'$  then
6:        $delay \leftarrow delay'$ 
7:   return  $delay$ 

```

At the beginning, the $delay$ is set to the maximum value possible, which is the difference between an instance's termination time and either the current time or an instance ready time (Line 2).

Then, the succeeding workloads are considered one by one. For each workload, a delay is the difference between its estimated finish time and the lesser value between its deadline and a instance's termination time (Line 4). An instance' termination time is considered in order to avoid additional cost, which is caused when a workload is delayed long enough for its finish time to exceed a termination time and results in an additional ATU. A total delay is updated if it is greater than a permissible delay of a workload (Line 6). When the loop is finished the final value is returned.

In summary, the Algorithm 4.2 calculates the minimum values between the permissible delays of all given workloads.

4.1.3 Shift Workloads

In an instance, when a new workload is added before its succeeding ones, it is necessary to update the start and finish times of them, i.e. shift their execution. This process is called

workload shifting and is presented by Algorithm 4.3. The inputs are the set of workloads and the amount of time to shift them.

The algorithm loops through all given workloads and updates their start/finish times with the given amount of time. It should be noted that Algorithm 4.3 can shift a workload backward and forward depending on a value of the amount of time to shift. If this value is positive, all workloads are shifted backward, which means they will start and finish later than previously estimated. On the other hand, if this value is negative, all workloads are shifted forward so that they can start and finish earlier than previously estimated.

Algorithm 4.3 Shift Workloads

```

1: function SHIFT_WORKLOADS( $e, W_s$ )
2:   for  $w \in W_s$  do
3:      $st_w \leftarrow st_w + e$ 
4:      $fi_w \leftarrow fi_w + e$ 
5:   return delay

```

4.1.4 Execution Pre-emption

Algorithm 4.4 Workload Pre-emption

```

1: function PREEMPTION( $v$ )
2:    $w_c \leftarrow$  current workload of  $v$ 
3:    $w_1 \leftarrow$  new workload containing all finished tasks of  $w_c$ 
4:    $st_{w_1} \leftarrow st_{w_c}$ 
5:    $fi_{w_1} \leftarrow NOW$ 
6:    $w_2 \leftarrow$  new workload containing all remained tasks of  $w_c$ 
7:    $st_{w_2} \leftarrow NOW$ 
8:    $fi_{w_2} \leftarrow fi_{w_c}$ 
9:   Add  $w_2$  to the front of the queue of unfinished workload
10:  Remove  $w_c$  from the queue of unfinished workload

```

Workload pre-emption happens when an execution of a BoT job on an VM is temporarily stopped so that another job can be executed instead. It should be noted that we only allow workload pre-emption but not task execution pre-emption. More specifically, a task has to be fully executed. Otherwise, if a task execution is halted, it has to be executed from the beginning. On the other hand, when pre-emption is performed on a workload, its remaining tasks will be saved to be executed later. Workload pre-emption is beneficial when there is an urgent job is submitted to an instance and has a deadline which is earlier than all existing workloads', including a workload which is in execution. In this case, it

is necessary to temporarily stop an execution of a current workload. The other benefit of workload pre-emption will be presented in the later chapter.

Algorithm 4.4 shows the logic for workload pre-emption. The algorithm splits a current workload, which is currently in execution, into two new ones. The first one, denoted as w_1 , contains all finished tasks and is marked as a finished workload (Line 3). The second one, denoted as w_2 , contains all the unfinished tasks, including the one currently in execution (Line 6). Then the second new workload is added to the queue of the unfinished workloads, from which the current workload is removed (Lines 9 and 10).

4.2 Workload Assignment Algorithm

The workload assignment process is presented in detail by Algorithm 4.5. It aims to assign as many tasks of a given job to a given list of VMs as possible without resulting in any deadline violation and additional cost. The inputs of the Algorithm are a job j , its remaining number of tasks n_j , and the list of VMs to execute those tasks V . The output of the Algorithm is the number of remaining tasks, i.e. tasks that are not assigned to any instances for execution. Which means that the Algorithm 4.5 does not guarantee to assign all tasks since it depends on the available capacity of the given VMs.

The first step is to *find the position of a new workload among the existing ones* using the *FIND_POSITION* method already presented in the Algorithm 4.1.

The next step is to *calculate a possible start time of a new workload*. Since a new workload can start immediately right after its preceding workload finishes, its possible start time can be the finish time of its preceding workload (Line 6). However, if there is no preceding workloads, a workload can start as soon as possible, i.e. either now or when an instance is ready if it is just created and is not ready yet (Line 8).

Since a new workload will be inserted in between the preceding workloads and the succeeding ones, it is necessary to calculate the available "space" between those two groups. This space is created by *delaying the execution of the succeeding workloads without violating their deadlines*, i.e. permissible delay. This value can be calculated using the *CALCULATE_DELAY* method which we already introduced by Algorithm 4.2.

Next, the algorithm *calculates the possible finish time of a new workload*. The possible finish time is the least value among three possibilities: i) a new workload's deadline, ii) a estimated terminated time of an instance (if there is no succeeding workload), or iii) the latest start time possible of the next workload calculated by adding a permissible delay to a its current start time (Line 10). In other words, the finish time of a new workload must i)

Algorithm 4.5 Workload Assignment

```

1: function WORKLOAD_ASSIGNMENT( $j, n_j, V$ )
2:   for  $v \in V$  do
3:      $W_v \leftarrow$  unfinished workload(s) of  $v$ 
4:      $(W_p, W_s) \leftarrow$  FIND_POSITION( $j, W_v$ )
5:     if  $W_p \neq \emptyset$  then
6:        $st \leftarrow$  finish time of the last workload in  $W_p$ 
7:     else
8:        $st \leftarrow \max(NOW, re_v)$ 
9:      $delay \leftarrow$  CALCULATE_DELAY( $v, W_s$ )
10:     $fi \leftarrow \min((st + delay), d_j, te_v)$ 
11:     $e \leftarrow fi - st$ 
12:     $n \leftarrow \lfloor \frac{e}{e_{j,v}} \rfloor$ 
13:    if  $n > 0$  then
14:       $n \leftarrow \min(n, n_j)$ 
15:       $e \leftarrow n \times e_{j,v}$ 
16:       $fi \leftarrow st + e$ 
17:      if  $W_p = \emptyset$  and  $v$  is executing the first workload in  $W_s$  then
18:        PREEMPTION( $v$ )
19:        SHIFT_WORKLOADS( $e, W_s$ )
20:         $w \leftarrow$  new workload of a job  $j$  on an instance  $v$ 
21:         $n_w \leftarrow n$ 
22:         $st_w \leftarrow st$ 
23:         $fi_w \leftarrow fi$ 
24:        Assign  $w$  to an instance  $v$  between  $W_p$  and  $W_s$ 
25:         $n_j \leftarrow n_j - n$ 
26:        if  $n_j = 0$  then
27:          return 0
28:   return  $n_j$ 

```

not exceed its deadline and an instance's termination time, and ii) not result in any deadline violation to the existing workload after it.

By having the start and finish times, it is possible to *calculate an available execution time of a new workload and the number of tasks that can fit into it (Lines 11 and 12)*. A new workload can be created only if the number of tasks is positive, i.e. it is possible to assign some tasks of a job to an instance (Line 13).

If the assignment is possible, the next step is to *find an actual number of tasks to be assigned*, which is the least value between the number of tasks calculated above and the number of unassigned tasks (Line 25). This is done in order to avoid assigning more tasks than necessary to an instance. Based on the actual number of tasks to be assigned, an execution time and a finish time are updated (Lines 15 and 16).

Workload execution pre-emption may be required when an instance is executed the first workload in a list of succeeding ones (Line 17), i.e. a current workload has a deadline which is greater than a new workloads. Workload pre-emption is performed by calling the function *PREEMPTION* whose logic is already presented by the Algorithm 4.4.

Next, all the succeeding workloads need to be shifted backward in order to create space for a new workload. This can be done using the *SHIFT_WORKLOADS* method presented by the Algorithm 4.3.

After that, based on the calculated start time, finish time, and a number of tasks, a new workload is created and assigned to an instance (from Line 20 to Line 24).

Finally, a number of unassigned tasks is updated by being subtracted with a number of assigned tasks (Line 25). If this value is 0, which means all tasks of a job are assigned, the Algorithm terminates (Line 27). Otherwise, it continues to the next VM.

The Algorithm 4.5 is graphically illustrated by the Figure 4.1 which outlines the main steps.

4.3 Chapter Summary

In this chapter, we have presented a greedy algorithm that assigns tasks of a job to a set of given instances while avoiding not only a potential deadline violation to both new and existing workloads but also extra cost. The algorithm is constructed using a set of smaller logics called utility functions.

Compared to other existing algorithms [65, 58], our proposed approach offers some improvements. First of all, instead of assigning each individual task to any available VM, it calculate the number of tasks that a VM can receive which reduce the complexity of the computation. Secondly, not only resource availability (i.e. the number of VMs and their

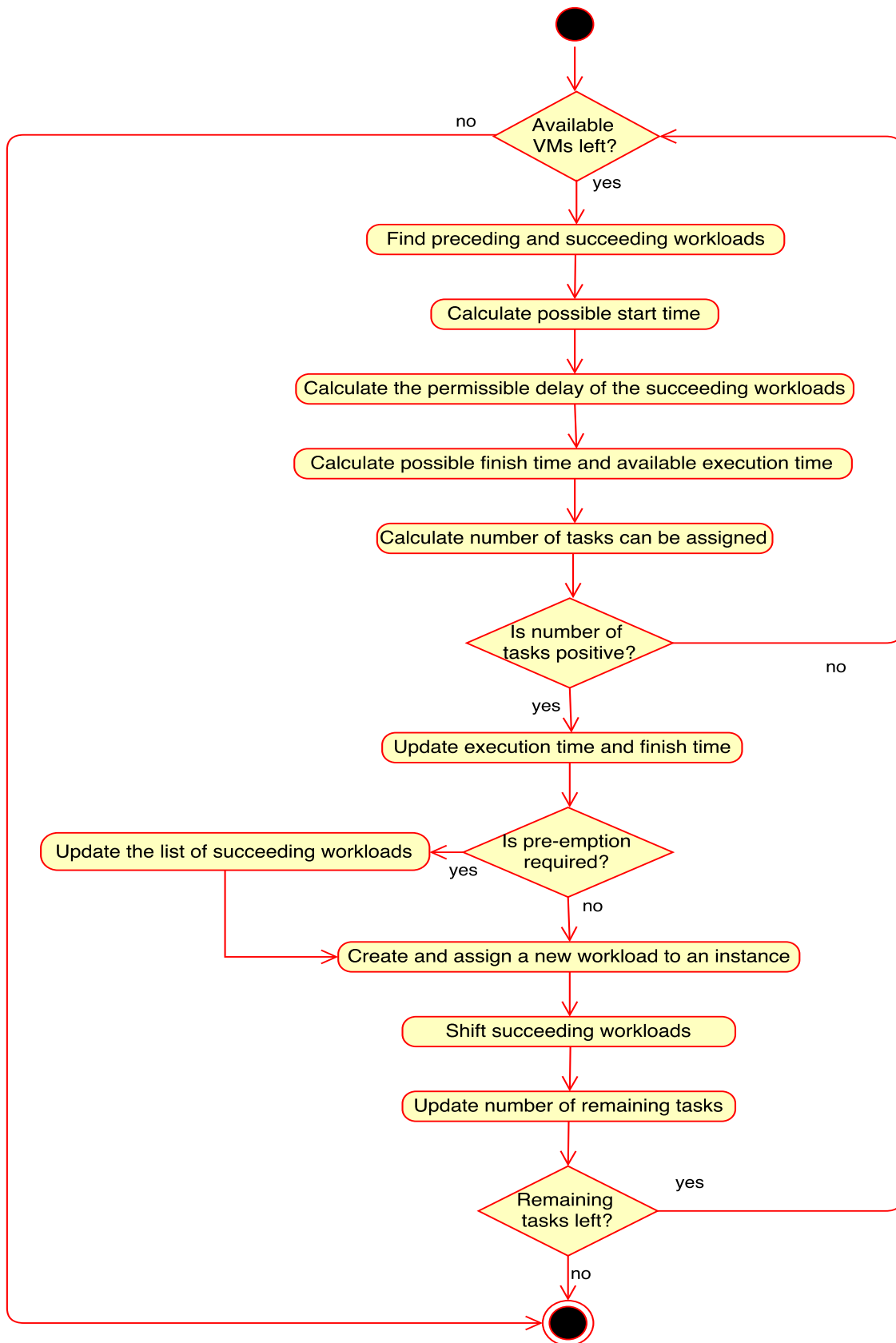


Fig. 4.1 Activity Diagram for the Workload Assignment Process presented by Algorithm 4.5

current workloads) but also resource requirements (i.e. jobs and their deadlines) are taken into account. As a result, the algorithm is able to assign as many tasks as possible while avoiding deadline violation. Finally, the urgency of the workload is also considered. More specifically, a workload execution can be delayed or pre-empted so that the more urgent one can be executed first. As a result, the workload assignment algorithm is crucial to the effectiveness of all of the scheduling approaches introduced in the next chapters since they are built on top of it.

Chapter 5

Execution Scheduling

The workload assignment presented in Chapter 4 requires to be given a set of VMs. In other words, it raises a question of selecting resources based on given requirements. That question will be answered by this chapter which presents and discusses **execution scheduling**. This chapter also aims to find a solution for the optimisation problem introduced in Chapter 3. In other words, execution scheduling aims to *determine the execution of all submitted BoT jobs on cloud VMs so that their deadlines are not violated and the total monetary cost is kept as low as possible*. The result of the execution scheduling is an **execution plan** which is a list of workloads, each of which represents an execution of a job on an instance in term of number of tasks, start time, and finish time. Based on the framework discussed in Chapter 2, the research presented in this chapter, and thesis, can be categorised as **scheduling in a heterogeneous cloud cluster** and consists of two activities:

- **Resource selection:** *to determine the number of VMs of different types in a cloud VM clusters*. Resource selection must ensure that there are enough processing capability of execute all tasks within their deadlines. Moreover, it also directly related to the monetary cost that a user has to pay.
- **Workload assignment:** *to allocate tasks of jobs to the VMs rented by users*. There are many constraints that workload assignment need to follow. For instance, all tasks should be executed before the deadline. On the other hand, it is impossible to execute tasks on an instance which is either not created yet or already terminated.

It should be noted that resource selection and workload assignment are interrelated to each other. Workloads can only be assigned to selected resources, i.e. VM that are created and not yet terminated. On the other hand, enough resources must be selected in order to ensure that all all tasks are executed within deadlines.

In Section 5.1, we are going to simplify the Model 3.14 into a *Quadratic Programming (QP)* problem, whose solution is optimal, i.e. an execution plan with minimum monetary cost. Since finding an optimal solution requires performing exhaustive search on the search space, it may take a considerable amount of time. As a result, in Section 5.2, we propose two other approaches which sacrifice the optimality of a solution in order to achieve faster solving time. Section 5.3 concludes this chapter.

5.1 The Exact Approach

In Chapter 3, we have introduced the mathematical representation of the statically scheduling problem. However, the optimisation presented in the Model 3.14 is complicated since it consists of not only many decision variables but also conditional constraints. As a result, even though Constraint Programming (CP) can be applied to find the solution, it can be time consuming due to the complexity of the problem.

In this section, we attempt to simplify the optimisation problem presented in the Model 3.14 into a Quadratic Programming (QP) problem which can be solved faster. This approach is named **exact approach** since the optimal solution can still be found using the proposed approach. We have already presented and discussed a large portion of this Section in the peer reviewed paper [64].

First of all, we make an assumption that *the submitted jobs are arranged in a predefined order and the execution of jobs on instances must follow that order*. Which means, upon submission, if a job j_1 is placed before a job j_2 , then j_1 must be executed before j_2 in all instances. Furthermore, since execution overlap is not supported, the execution of j_2 can start only when the execution of j_1 finishes. In summary, we have the following formula:

$$st_{w_{j_1,v}} \leq fi_{w_{j_1,v}} = st_{w_{j_2,v}} \quad (5.1)$$

Formula 5.1 also states that a workload starts as soon as its preceding one finishes, which means the start time of a workload is equal to the finish time of its preceding workload.

Notably, it is possible for a job not to be executed on an instance while its succeeded one is, e.g. j_1 is not executed on an instance v while j_2 is. The constraint presented in Formula 5.1 can still be satisfied as the finish time of j_1 will be equal to its start time, which results in its execution time to be zero.

There are different ways for determine the order of jobs. For instance, jobs can be ordered based on the urgency represented by their deadlines, e.g. *earliest-deadline-first*. Similarly, each user can have his/her own way to order jobs based on their importances, e.g. assigning

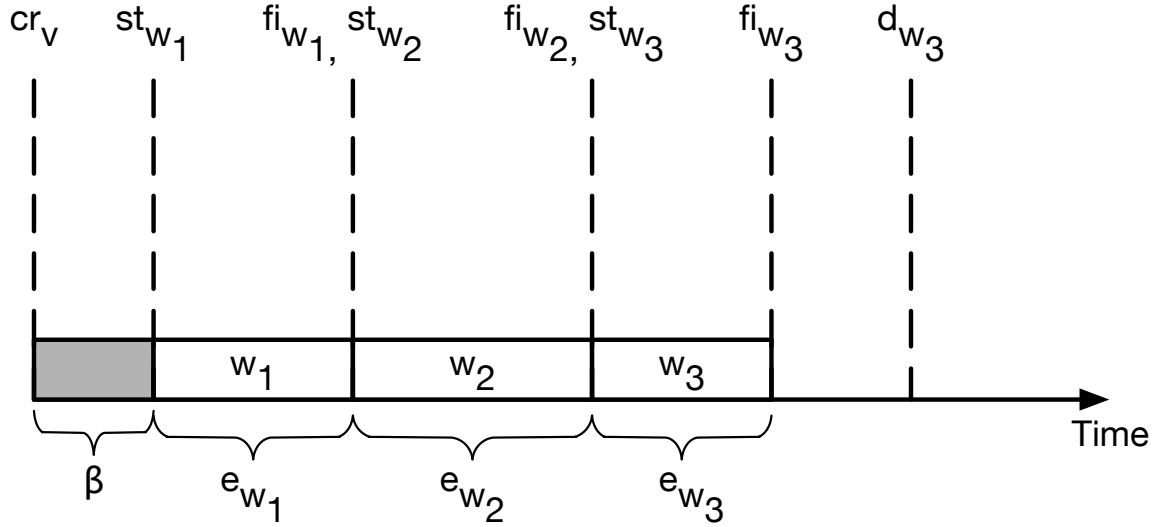


Fig. 5.1 Example of Execution Order in an Instance

a priority value to each job. In this thesis, we adopt the earliest-deadline-first ordering approach.

Figure 5.1 illustrates an example in which three workloads w_1 , w_2 , and w_3 are scheduled to be executed on an instance. A workload w_1 must be executed first, which is immediately when an instance is ready. Hence, its start time is equal to an instance's ready time, i.e. $st_{w_1} = re_v = cr_v + \beta$.

When an instance finishes executing w_1 , it immediately starts executing w_2 . Thus, a finish time of w_1 is equal to a start time of w_2 , i.e. $fi_{w_1} = st_{w_2}$. Similarly, a finish time of w_2 is equal to a start time of w_3 , i.e. $fi_{w_2} = st_{w_3}$. Since there is no more workload after w_3 , an instance does not start executing any workload after finish executing w_3 .

Notably, in order to satisfy the deadline, the finish times of all workloads must be less than or equal to their jobs' deadlines. For instance, as shown in the Figure 5.1 a workload w_3 has to finish before its deadline, i.e.:

$$fi_{w_3} \leq d_{w_3} \quad (5.2)$$

Since an execution time of a workload is the difference between its finish and start times, it is possible to calculate a finish time given an execution time and a start time, e.g. $fi_{w_3} = st_{w_3} + e_{w_3}$. Furthermore, as a workload w_3 starts executing right after its previous one (i.e. w_2) finishes, we have $st_{w_3} = fi_{w_2}$. In other words, a finish time of a workload can be calculated by adding its execution time to a finish time of its preceding one. If a workload has no preceding one (e.g. w_1) its start time is the same as an instance's ready time, assuming

that a workload is executed as soon as an instance is ready. As a result, a finish time of a workload w_3 can be calculated as follow:

$$fi_{w_3} = cr_v + \beta + e_{w_1} + e_{w_2} + e_{w_3} \quad (5.3)$$

Replacing Formula 5.2 to Formula 5.3, we have:

$$cr_v + \beta + e_{w_1} + e_{w_2} + e_{w_3} \leq d_{w_3} \quad (5.4)$$

Or,

$$e_{w_3} \leq d_{w_3} - (e_{w_1} - e_{w_2} + \beta + cr_v) \quad (5.5)$$

Formula 5.5 is a constraint which states that an execution time of a workload w_3 must be less than or equal to its deadline subtracting the sum of execution times of preceding workloads, the instance creation overhead, and an instance creation time. This constraint can be generalised as follow:

$$e_{w_{j,v}} \leq d_j - \left(\sum_{j'=1}^{j-1} e_{w_{j',v}} + \beta + cr_v \right) \quad (5.6)$$

Formula 5.8 states that on the same instance, an execution time of a workload must be less than or equal to a its deadline subtracting the summation of execution times of the preceding workloads, the creation overhead, and an instance's created time.

It should be noted that we do not assume the prior knowledge regarding job submission, i.e. we do not know when jobs are submitted. As a result, we cannot create a VM before a job submission. In other words, a VM can only be create as soon as a job is submitted, i.e.:

$$cr_v = s_j \quad (5.7)$$

Replacing Formulas 5.7 and 3.1, which states that a job's available execution time is the difference between its deadline and its submitted time, to Formula 5.6, we have:

$$e_{w_{j,v}} \leq d_j - s_j - \sum_{j'=1}^{j-1} e_{w_{j',v}} - \beta = e_j - \sum_{j'=1}^{j-1} e_{w_{j',v}} - \beta \quad (5.8)$$

Formula 5.8 states that *an execution time of a workload on an instance must be less than or equal to its job's available execution time (i.e. e_j) subtracting the summation of execution times of the preceding workloads, and the creation overhead*. In other words, an execution time of a workload is limited by not only the creation overhead but also execution times

of exceeding workloads. The formula simplifies the initial Model 3.14 by removing the necessity of finding the start and finish times of each workload on an instance. Moreover, the condition which prevents execution overlap on the same instance, i.e. Formula 3.14i, is also eliminated.

In other to further simplify the problem, we want to remove the remaining conditional constraint, i.e. Formula 3.14d regarding if an instance is created or not. In order to do so, we introduce a set X which consists of binary values, i.e. $x = \{0, 1\}$, indicating if an instance is used or not. Which means:

$$x_v = \begin{cases} 1, & \text{if an instance } v \text{ is used} \\ 0, & \text{otherwise} \end{cases} \quad (5.9)$$

Using Formula 5.9, an execution time of a workload on an instance, which is presented in Formula 5.8, can be redefined as follow:

$$e_{w_{j,v}} \leq x_v \times (e_j - \beta - \sum_{j'=1}^{j-1} e_{w_{j',v}}) \quad (5.10)$$

If an instance v is created, which means $x_v = 1$, the Formula 5.10 is similar to the Formula 5.8. On the other hand, if an instance v is not created, i.e. $x_v = 0$, an execution time of a workload $w_{j,v}$ must be equal to zero. Since the number of assigned tasks is calculated based on a execution time, as presented in Formula 3.11, this also means there is no task of a job j assigned to an instance v .

Similarly, a running time of an instance, which is presented in the conditional Formula 3.13, can be redefined as:

$$r_v = x_v \times (\beta + \sum_{j \in J} e_{w_{j,v}}) \quad (5.11)$$

If an instance v is created, i.e. $x_v = 1$, its running time is positive and its cost (i.e. c_v) can be calculated based on the Formula 3.4. On the other hand, if an instance v is not created, i.e. $x_v = 0$, its running time and cost have to be zero and a user does not have to pay for instances which are not created.

By removing all the conditional constraints, the constraint programming problem presented by Model 3.14 can be simplified into an Integer Quadratic Programming problem as follow:

$$\text{minimise} \quad c = \sum_{v \in V} c_v \quad (5.12a)$$

$$\text{subject to} \quad c_v = atu_v \times p_{it_v} \quad (5.12b)$$

$$atu_v = \lceil \frac{r_v}{3600} \rceil \quad (5.12c)$$

$$r_v = x_v \times (\beta + \sum_{j \in J} e_{w_{j,v}}) \quad (5.12d)$$

$$e_{w_{j,v}} \leq x_v \times (e_j - \beta - \sum_{j'=1}^{j-1} e_{w_{j',v}}) \quad (5.12e)$$

$$x_v = \{0, 1\} \forall v \in V \quad (5.12f)$$

$$n_w = \lfloor \frac{e_w}{e_{a_{j_w, t_{v_w}}}} \rfloor \quad (5.12g)$$

$$\sum_{w \in W_j} n_w = n_j, \forall j \in J \quad (5.12h)$$

Most of the formulas in Model 5.12 are similar to those in Model 3.14. On the other hand, some of them are different

- Formula 5.12d calculates a running time of each instance, taken into account whether an instance is created or not, indicated by the parameter x .
- Formula 5.12e calculates the execution times of workloads assigned to VMs, taken into account whether an instance is created or not, indicated by the parameter x .
- Formula 5.12f defines that the value all items in set X must be either 1, if an instance is created, or 0, if an instance is not created.

In comparison to Model 3.14, Model 5.12 reduces the complexity of finding a scheduling plan by removing the conditional constraints. Moreover, the number of decision variables to be found is reduced as well. More specifically, Model 3.14 aims to find the execution times of workloads on each instance instead of calculating their start and finish times.

5.2 Single Job Scheduling Approach

The Exact approach introduced in Section 5.1 considers all jobs at the same time for scheduling. Moreover, since this approach aims find an optimal solution, it may require a considerable amount of time to find a solution, especially when the number of jobs is high, which results

in the delay in executing all jobs. In this section, we attempt to further simplify the problem by considering one job at a time. In other words, *instead of handling a submission consisting of multiple jobs, the approaches presented in this section will handle multiple submissions each of which consists only one job*. We believe that by doing so, a solving time can be reduced. Moreover, after a job is scheduled, its execution can start immediately instead of waiting for the succeeding jobs. Similarly to the previous section, jobs are scheduled based on the predefined order which is earliest-deadline-first.

5.2.1 The Hybrid Scheduling Approach

In this section, we introduce the **hybrid approach**, which combines **Integer Linear Programming (ILP)** technique to find a scheduling plan for a single job and heuristic algorithm for combining the individual scheduling. This approach guarantees that the scheduling plan for each job is optimal but does not ensure the optimality of the overall scheduling for all jobs. This approach has already been presented in peer reviewed papers [63, 64].

As mentioned earlier, an available execution time of a job is the difference between its deadline and submitted time. However, if a job is executed on a new instance, it has to wait for an instance to be ready. As a result, its available execution time needs to be updated in order to consider the creation overhead time. Notably, since VMs should be created as soon as a job is submitted in order to accommodate its workload, a VM's created time is equal to a job's submitted time. Thus, an available execution time of a job on a new VM is the difference between a job's deadline and a VM's ready time:

$$e_j = d_j - s_j - \beta \quad (5.13)$$

For any given VM of a type $t \in T$, the **capacity of a VM for a given job**, denoted as $cp_{j,t}$, is the maximum number of tasks that can be executed within a job's deadline. This value can be calculated by dividing its available execution time to a task execution time corresponding to a job and an instance type:

$$cp_{j,t} = \lfloor \frac{e_j}{e_{a_j,t}} \rfloor \quad (5.14)$$

In a cloud VM cluster consisting of instances of different type, let Y be a set whose each item is a number of VMs for each type. For instance, there are y_t VMs created for an instance type $t \in T$ in a cluster. Hence, the number of tasks that y_t VMs of type t can execute within j 's deadline, denoted as $n_{j,t}$, is:

$$n_{j,t} = y_t \times cp_{j,t} \quad (5.15)$$

Given a cluster of cloud VMs, in which there are y_t VMs of type t , the capacity of the whole cluster corresponding to a job j , denoted as cp_j , can be calculated as the summation of the capacity of each type:

$$cp_j = \sum_{t \in T} n_{j,t} \quad (5.16)$$

In order to fully executed a job, all of its tasks must be executed, thus, the following constraint is added to ensure that the capacity of a cloud VM cluster must be greater than or equal to a job's number of tasks:

$$cp_j \geq n_j \quad (5.17)$$

Assuming that the workload is evenly distributed among all VMs, which means each of them is used for the similar amount of time, which is less than or equal to a job's available execution time e_j , hence the maximum total number of ATUs used by each instance is:

$$atu = \lceil \frac{e_j}{3600} \rceil \quad (5.18)$$

The total monetary cost for each type can be calculated by multiplying the number of ATUs to the cost for each ATU and the number of created VMs for that type:

$$c_t = atu \times p_t \times y_t \quad (5.19)$$

In a conclusion, the problem of calculating the amount of VMs required to execute a job within its deadline can be represented as the following optimisation model:

$$\begin{aligned}
&\text{minimise} && c = \sum_{t \in T} c_t && (5.20a) \\
&\text{subject to} && c_t = atu \times p_t \times y_t && (5.20b) \\
&&& y_t \geq 0 && (5.20c) \\
&&& atu = \lceil \frac{e_j}{3600} \rceil && (5.20d) \\
&&& e_j = d_j - \beta && (5.20e) \\
&&& cp_{j,t} = \lfloor \frac{e_j}{e_{a,j,t}} \rfloor && (5.20f) \\
&&& n_{j,t} = y_t \times cp_{j,t} && (5.20g) \\
&&& cp_j = \sum_{t \in T} n_{j,t} && (5.20h) \\
&&& cp_j \geq n_j && (5.20i)
\end{aligned}$$

The breakdown of Model 5.20 is as follow:

- Equation 5.20b calculates the total cost for each instance type based on its number of VMs (presented by Equation 5.20c), and the number of ATUs (calculated by Equation 5.20d).
- Equation 5.20e calculates the available execution time for a job.
- Equation 5.20f calculates the maximum number of tasks that an instance of a certain type can execute.
- Equations 5.20g and 5.20h calculate the capacity of a cloud VM cluster, i.e. the number of tasks can be executed by all VMs of all types.
- Equation 5.20i is a constraint which states that the capacity of a cloud VM cluster must be greater than or equal to the number of tasks of a job.

In summary, the Model 5.20 aims to *select the cheapest amount of VMs of different types so that the total capacity is enough to handle a job within its deadline*. After resources are selected, tasks of a job can be assigned to them using a Workload Assignment algorithm which is presented by Algorithm 4.5 in Chapter 4.

5.2.2 Heuristic Single Job Scheduling

In the previous section, we have introduced an Integer Linear Programming approach to schedule an execution of a BoT job on the cloud so that its deadline is ensured and the monetary cost is minimised. However, as the hybrid approach still aims for an optimal solution for each job, it still may have a high solving time. In this section, we propose a **heuristic approach** which aims to reduce a complexity. However, since the proposed mechanism takes a greedy approach, which selects the best options given the current state, it does not guarantee that an optimal solution, i.e. one with cheapest cost, is found. Hence, it presents a trade-off between solving time and solution quality.

The main idea of the proposed heuristic approach is to start with an initial scheduling plan which is quick and simple to find. Then, an iterative process is performed in order to transform an initial plan so that either a monetary cost or a total execution time is reduced. This approach has been investigated by us and proven to be able to increase the performance and reduce the cost of a cloud VM cluster [59, 60, 62]. In this case, the algorithm starts with a heterogeneous cluster, i.e. all VMs are created from the same type, and then transform it to a homogeneous cluster, which results in lower cost as concluded by our previous research [61].

Algorithm 5.1 Create Initial Plan

```

1: function CREATE_INITIAL_PLAN( $j, V$ )
2:    $e_j \leftarrow d_j - s_j - \beta$ 
3:    $atu \leftarrow \lceil \frac{e_j}{3600} \rceil$ 
4:    $c \leftarrow \infty$ 
5:    $\tau \leftarrow NULL$ 
6:    $y_\tau \leftarrow 0$ 
7:   for  $t \in T$  do
8:      $cpv_t \leftarrow \lfloor \frac{e_j}{e_{a_{j,t}}} \rfloor$ 
9:      $y_t = \lceil \frac{n_j}{cpv_{j,t}} \rceil$ 
10:     $c_t \leftarrow y_t \times atu \times p_t$ 
11:    if  $c_t < c$  then
12:       $c \leftarrow c_t$ 
13:       $\tau \leftarrow t$ 
14:       $y_\tau \leftarrow y_t$ 
15:   $V \leftarrow$  create  $y_\tau$  VMs of type  $\tau$ 
16:  return  $V$ 

```

Algorithm 5.1 illustrates how to create an initial plan which consists of VMs of only one type, i.e. homogeneous cloud VM cluster.

Firstly, based on a job's deadline, its available execution time and the corresponding number of ATUs used by each VM are calculated (Lines 2 and 3).

Then, instance types are considered one by one. For each of them, the capacity per VM, the total number of VMs required to execute all tasks and the total cost are calculated (Lines 8, 9, and 10).

An instance type which results in the lowest cost to execute all tasks within a job's deadline is selected (denoted as τ). Finally, a homogeneous cluster of VMs of type τ is created (Line 15). In summary, Algorithm 5.1 tries to *find the cheapest homogeneous cloud VM cluster to execute all tasks of a job within its deadline*.

Algorithm 5.2 Transform Plan

```

1: function TRANSFORM( $V, atu, \tau, y_\tau, \eta$ )
2:   if  $\eta > y_\tau$  or  $y_\tau = 0$  then
3:     return  $V$ 
4:    $f \leftarrow \eta \times p_\tau \times atu$ 
5:    $V' \leftarrow$  remaining VMs after removing  $\eta$  instances of type  $\tau$ 
6:    $n_{V'} \leftarrow$  numbers of tasks executed by the remaining VMs
7:    $n' \leftarrow n_j - n_{V'}$ 
8:    $t' \leftarrow NULL$ 
9:    $y' \leftarrow 0$ 
10:  for  $t \in T$  do
11:    if  $t \neq t^s$  then
12:       $y_t \leftarrow \lfloor \frac{f}{p_t \times atu} \rfloor$ 
13:       $c_t \leftarrow y_t \times p_t \times atu$ 
14:       $n_{j,t} \leftarrow y_t \times cpv_{j,t}$ 
15:      if  $n_{j,t} \geq n'$  and  $c_t < c'$  then
16:         $f \leftarrow c_t$ 
17:         $t' \leftarrow t$ 
18:         $y' \leftarrow y_t$ 
19:  if  $t' \neq NULL$  then
20:     $V \leftarrow$  replace  $\eta$  VMs of type  $\tau$  by  $y'$  VMs of type  $t'$ 
21:    return TRANSFORM( $V, atu, \tau, y_\tau - y', \eta$ )
22:  else
23:    return TRANSFORM( $V, atu, \tau, y_\tau, \eta + 1$ )

```

With homogeneous cloud VM cluster of type τ created by Algorithm 5.1, the next step is to transform it into a heterogeneous cluster in order to reduce the cost while ensuring that all tasks are still executed within a job's deadline. This transformation process is presented in Algorithm 5.2.

In general, the transformation algorithm is a recursive and greedy process, in which each iteration tries to replace a certain number of the selected type τ by VMs of other type so that a cost can be reduced.

At the beginning, based on the number of VMs of the selected type τ to be replaced, which is denoted as η and set to one at the beginning, the algorithm calculates the *available fund* f which is the amount of money paid to η VMs of type τ to be replaced (Line 4). For example, if the algorithm is trying to replace 4 VMs, each of which is estimated to cost 2, the available fund is $4 \times 2 = 8$.

Next, the number of tasks that a new set of replacing VMs is required to execute is calculated. More specifically, this value is the difference between the total number of tasks and the number of tasks executed by the remaining VMs which are not considered to be replaced (Line 7).

Then, Algorithm 5.2 loops through all available instance types different than τ . For each type, it calculates the number of VMs affordable by the available fund f and the number of tasks that those VMs can execute. *The cheapest instance type that can execute all tasks and has the total cost lower than the available fund is selected* (Lines 10 to 18).

VMs of type τ is replaced by a replacement type if one is found (Line 20). And a process continues. On the other hand, if no replace type is found, η , i.e. a number of VMs of type τ to be replace, is increased by one before continuing the process. By increasing the number of VMs to be replaced, the available fund is increased as well. However, this also results in the higher number of tasks that a replacing VMs must be able to handle.

Algorithm 5.2 terminates if either η is greater than the actual number of VMs of type τ or there is no VM of type τ left (Line 2).

5.2.3 Handling Multiple Jobs Using Single Job Scheduling Approaches

The approaches introduced in Sections 5.2.1 and 5.2.2 are able to schedule only one job. Hence, in order to handle an submission consisting of multiple jobs, it is necessary to combine them with the *ASSIGN_WORKLOAD* method presented by Algorithm 4.5 in Section 4.2.

Algorithm 5.3 Single Job Scheduling

```

1: function SINGLE_JOB_SCHEDULING( $J, V_0$ )
2:   for  $j \in J$  do
3:      $n_j \leftarrow \text{ASSIGN\_WORKLOAD}(j, n_j, V_0)$ 
4:     if  $n_j > 0$  then
5:        $V' \leftarrow$  new VMs for the remaining tasks of  $j$ 
6:        $V_0 \leftarrow V_0 \cup V'$ 

```

Algorithm 5.3 presents a process of handling multiple jobs using single job scheduling approaches. The inputs of an algorithm are the list of submitted jobs, and the list of existing instances (denoted as v_0), which can be empty if there is no created VM yet.

Algorithm 5.3 goes through each job and first tries to assign it to the existing VMs (Line 3) using the *WORKLOAD_ASSIGNMENT* method presented by Algorithm 4.5. It takes a job, its number of tasks, and a set of existing instances. It returns the number of remaining tasks, i.e. those that cannot be assigned to the existing instances.

After the assignment, if there are remaining tasks of a job, i.e. the existing VMs are not enough to handle all tasks of the submitted job, new VMs must be created using either the Hybrid approach of the Heuristic approach (Line 5). Then, the new VMs are added to the list of existing ones and will be considered to schedule the succeeding jobs (Line 6).

5.3 Chapter Summary

In this chapter, we have presented different approaches for scheduling the execution of BoT jobs on the cloud. More precisely, our proposed approaches determine the workloads, each of which corresponds to a job and consists of the number of tasks, the start and finish times, assign to each instance. They aim to not only ensure that all tasks are executed within the deadlines but also keep the total monetary cost as low as possible.

There are three scheduled approaches which have been proposed in this chapter:

- **The exact approach** which aims to find an optimal scheduling plan but may require very high solving time.
- **The hybrid approach** which manages to find optimal scheduling plans for each job but does not ensure that the complete scheduling plan for all jobs is optimal. However, it is less complicated and is believed to have a moderate solving time.
- **The heuristic approach** which does not guarantee the optimality. Nevertheless, due to its simplicity, we believe this approach can find a solution in a very short time.

The three proposed approaches also demonstrate the trade-off between solution optimality and solving time. We believe that finding an optimal solution which guarantees minimum cost has high computational complexity, which may result in high solving time. On the contrary, it is possible to have another approach which is less complex, i.e. lower solving time, however, such approach does not guarantee to find an optimal solution. The difference in performance and solving times between the proposed approach will be evaluated in Chapter 8.

Chapter 6

Execution Management

The previous chapter discussed the approaches to schedule an execution of multiple BoT applications with deadlines on the cloud. Those approaches are also called static-scheduling which mean they are performed prior to the actual execution based on given parameters. However, as mentioned earlier, it is possible for the given parameters to vary during runtime, as a result, the results of the static scheduling process can become obsolete and needs to be update.

This chapter presents the **execution management** which is performed during the execution in real-time. The execution management consists of two separated components with different objectives. **Dynamic scheduling** aims to handle unexpected events during runtime and is presented by Section 6.1. **Unknown handling** mechanism estimates unknown variables and is discussed in Section 6.2. This chapter is concluded by Section 6.3

6.1 Dynamic Scheduling

In the previous chapter, we have discussed the methods to schedule the execution of multiple BoT applications with deadlines on the cloud. These methods are performed prior to the actual execution and rely on task execution times to estimate the start and finish times of each workload. However, as we already mentioned, a task execution time is just an average value, which means its actual value can vary during runtime.

As mentioned in Chapter 2, the variation of both the task execution time and creation time overhead can affect an estimation of workload execution and result in inaccurate scheduling plan during an execution. In other to prevent potential violations during runtime, in this section, we present the **execution management** process which i) monitors execution during runtime, ii) detects potential violations, and iii) resolves them. The research in this section has been discussed and presented in peer reviewed papers [63, 64].

This section is structured as follows: Section 6.1.1 briefly presents the monitoring process to retrieve execution progresses of all VMs. Section 6.1.2 describes a process to detect potential violation by categorising VMs into groups based on their execution progresses. Section 6.1.3 presents an algorithm to reassign workloads between VMs in order to reduce deadline violation without risking creating new ones.

6.1.1 Progress Monitoring

During execution, the **progress monitoring process** is performed periodically in order to retrieve the current execution progress of all VMs. For each instance, its execution progress consists of the list of finished tasks of a current workload, i.e. the one that is being executed by an instance, and their actual execution times. The actual execution times are recorded in order to be used in the later section.

6.1.2 Progress Categorisation

Given the progresses of all running VMs, the next step is to categorise them into different groups, each of which is handled differently. This process, which is named **progress categorisation**, will be discussed in this section.

Based on a current execution progress, a VM can be put into one of the following categories:

- *IDLE*: instance that does not have any workload to execute.
- *CAN_RECEIVE*: instance that has remaining workloads to execute but can receive extra ones.
- *NOT_RECEIVE*: instance that has finished executing current workloads and must start executing the next ones immediately, so it cannot receive an extra workload.
- *NOT_GIVE*: instance that is executing the last task of its current workload so it cannot give any workload.
- *VIOLATING*: instance that is predicted to violate the deadlines of its current workload.
- *CAN_GIVE*: instance that is executing its current workload and is predicted to not violate a deadline but can still give workload if possible.

In terms of reassignment, instances in categories *IDLE* and *CAN_RECEIVE* can receive extra tasks, i.e. new workloads from those of category *VIOLATING* and *CAN_GIVE*.

Algorithm 6.1 presents the logic to categorise an instance based on its current progress. The inputs of the algorithm are an instance, its current workload and the number of finished tasks (denoted as n_w^f). The output of the algorithm is a category of an instance. Moreover, Algorithm 6.1 also calculate the amount of time that an instance either can use to receive extra workload or need to give away in order to avoid deadline violation.

First of all, an instance needs to be checked if it has finished its current workload. This can be done by comparing the number of finished tasks to the total number of tasks assigned to a workload (Line 4). In other words, an instance has finished executing its current workload if the number of finished tasks is equal to the number of tasks assigned to a workload. On the other hand, if the number of finished tasks is less than the number of assigned tasks, a VM has not finish its execution yet. Instances are handled differently based on the fact that they finish their current workloads or not.

Categorise Finished Instances

For VMs which have finished its current execution, the next step is to check if there are any unfinished workloads assigned to them (Line 6). An instance that has no remaining workload is called **idle instance** and its label is set to *IDLE* (Line 7). Since each VM is charged by hour, it is possible to keep an idle instance running until the end of the nearest ATU without paying any additional cost. Hence, an idle instance is free to receive extra workload as long as it does not exceed a termination time. In other words, its available time to receive extra workload is the difference between termination time and the current time (Line 9). The idle workload is illustrated by Figure 6.1a.

On the other hand, if a VM has remaining workload(s) left, it is necessary to check the expected start time of the immediately next workload. If an expected start time is in the future, i.e. there is gap between current time and an expected start time of the next workload, it is possible for an instance to receive extra workload and its label is set to *CAN_RECEIVE* (Lines 16 and 17). Its available time to receive extra workload is the difference between a start time of the next workload and the current time (Line 18), as illustrated by Figure 6.1b.

However, if an expected start time is not in the future, i.e. an instance finishes its execution either on time or later than estimated, a VM, whose label is *START_NEW*, cannot receive additional workload and has to start executing the next workload immediately (Lines 13 and 14).

Algorithm 6.1 Progress Categorisation

```

1: function CATEGORISE( $v, w, n_w^f$ )
2:    $LABEL \leftarrow NULL$ 
3:    $e \leftarrow 0$ 
4:   if  $n_w^f = n_w$  then
5:     An instance has finished its current execution
6:     if  $v$  has no remaining workload then
7:        $v$  is idle
8:        $LABEL \leftarrow IDLE$ 
9:        $e \leftarrow te_v - NOW$ 
10:    else
11:       $w^n \leftarrow$  next workload
12:      if  $st_{w^n} \leq NOW$  then
13:         $v$  cannot receive additional tasks
14:         $LABEL \leftarrow START\_NEW$ 
15:      else
16:         $v$  can receive additional tasks
17:         $LABEL \leftarrow CAN\_RECEIVE$ 
18:         $e \leftarrow st_{w^n} - NOW$ 
19:    else
20:       $n_w^r \leftarrow n_w - n_w^f - 1$ 
21:      if  $n_w^r = 0$  then
22:         $v$  is executing the last task
23:         $LABEL \leftarrow NOT\_GIVE$ 
24:      else
25:         $e_w^r = n_w^r \times e_{j_w, v_w}$ 
26:         $fi_w^u = NOW + e_w^r$ 
27:        if  $fi_w^u > d_{j_w}$  then
28:          Possible deadline violating detected
29:           $LABEL \leftarrow VIOLATING$ 
30:           $e \leftarrow fi_w^u - d_{j_w}$ 
31:        else
32:           $LABEL \leftarrow CAN\_GIVE$ 
33:           $e \leftarrow \frac{fi_w^u - NOW}{2}$ 
34:    return ( $LABEL, e$ )

```

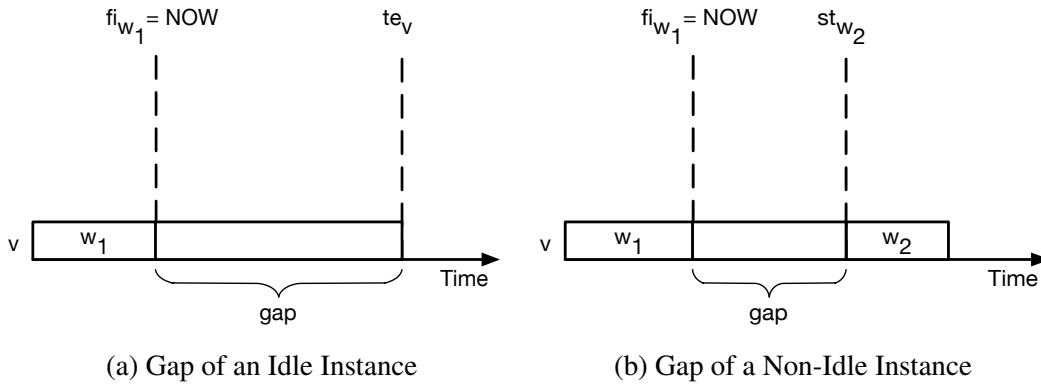


Fig. 6.1 Gap for Receiving Extra Workload

Categorise Unfinished Instances

For instances which have not finished executing the current workloads yet, the first step is to calculate the number of remaining tasks, which is denoted as n_w^r . The number of remaining tasks can be calculated by subtracting the current workload's total number of tasks to the number of finished tasks. Notably, since there is one task currently in execution, the result is them subtracted to one (Line 20).

If the number of remaining tasks is equal to zero, which mean an instance is executing the last task of a workload and has no task left be reassigned (Lines 22 and 23).

On the other hand, if there are tasks left to be reassigned, i.e. $n_w^r > 0$, the next step is to calculate the remaining execution time by multiplying the number of remaining tasks to the task execution time (Line 25). Then, an updated expected finish time, denoted as fi_w^u , can be calculated by adding a remaining execution time to the current time (Line 26).

A potential deadline violation is detected if an instance is estimated to finish executing its current workload after a deadline, i.e. an updated finish time is greater than a workload's job's deadline (Lines 28 and 29). An instance's label is set to *VIOLATING*. In order to resolve a violation, the goal is to reduce an execution time of a violating workload an amount of time which is equal to the difference between an updated finish time and a deadline (Line 30).

On the other hand, an instance which is not predicted to be deadline violating if an updated finish time is less than or equal to a deadline (Line 32), its label is set to *CAN_GIVE*. Notably, it is possible to reassign tasks from a non-violating VM in order to improve performance. For instance, given an instance v_1 has 10 tasks left and is predicted to not violate a deadline. However, there is another VM v_2 which can receive 5 tasks from v_1 . In this case, it is advisable to send 5 tasks from v_1 to v_2 so that a workload can be finished earlier than

estimated, thus improve an overall performance. Hence, it is beneficial to reassign at most half of the remaining execution time of this workload, if it is possible (Line 33).

6.1.3 Dynamic Reassignment

After categorising VMs into different groups, the next step is to perform **dynamic reassignment**. This is a process in which tasks are moved between VMs in order to reduce the risk of deadline violation. More specifically, it aims to move tasks from VMs that need to give (i.e. those of groups *VIOLATING* and *CAN_GIVE*, called **task-giving instances**) to VMs that can receive tasks (i.e. those of groups *IDLE* and *CAN_RECEIVE*, called **task-receiving instances**).

Algorithm 6.2 Dynamic Reassignment

```

1: function REASSIGNMENT(IDLE, CAN_RECEIVE, VIOLATING, CAN_GIVE)
2:   IDLE  $\leftarrow$  sort IDLE by receiving times in descending order
3:   CAN_RECEIVE  $\leftarrow$  sort CAN_RECEIVE by receiving times in descending order
4:    $V^r \leftarrow IDLE \cup CAN\_RECEIVE$ 
5:   VIOLATING  $\leftarrow$  sort VIOLATING by giving times in descending order
6:   CAN_GIVE  $\leftarrow$  sort CAN_GIVE by giving times in descending order
7:    $V^g \leftarrow VIOLATING \cup CAN\_GIVE$ 
8:   for  $v^g \in V^g$  do
9:      $w \leftarrow$  current workload of  $v^g$ 
10:     $e^g \leftarrow$  amount of time that  $v^g$  needs to give
11:     $n^g \leftarrow \lceil \frac{e^g}{e_{a_{j_w, t_{v^g}}}} \rceil$ 
12:     $n^g \leftarrow \min(n^g, \text{number of remaining tasks of } w)$ 
13:     $n' \leftarrow ASSIGN(j_w, n^g, V^r)$ 
14:    if  $n' < n^g$  then
15:      Remove  $n'$  tasks from  $w$  of  $v^g$ 
16:      Remove all VMs that receive extra workloads from  $V^r$ 
17:    if  $V^r = \emptyset$  then
18:      Break

```

Algorithm 6.2 describes the dynamic reassignment process. First of all, the lists of task-receiving and -giving instances are constructed. The list of task receiving VMs consists of all instances in the *IDLE* and *CAN_RECEIVE* groups, each of which is sorted based on their receiving times in descending order (Lines 2 and 3). Which means a VM that can receive a higher workload is considered first. Notably, in the list of task-receiving instances, *IDLE* VMs are placed in front of *CAN_RECEIVE* ones (Line 4). In other words, we prefer to assign tasks to idle VMs since they do not have to execute any workloads. On the other

hand, if workload is assigned to a VM with remaining tasks, a delay in an extra workload can cascade to other ones.

Similarly, the list of task-giving instances consists of VMs from *VIOLATING* group is placed in front of *CAN_GIVE* group, both of them are sorted by their giving times in descending order (Lines 5, 6, and 7). Which means that violating instances are considered for reassignment before *CAN_GIVE* ones.

Next, all task-giving instances are considered one by one (Line 8). For each task-giving instance, the number of tasks needed to be reassigned is calculated by dividing the giving time to the task execution time of a workload on a VM (Line 11). Notably, in order to avoid over-reassigning tasks, the result must then be compared with the actual number of remaining tasks, and the smaller value is the actual number of tasks to be given (Line 12).

After that, tasks from the giving instance are assigned to the receiving one using Algorithm 4.5 proposed in Chapter 4. The algorithm returns the number of remaining tasks, if this value is less than the number of tasks to give, that means reassignment is performed (Lines 14 and 15). Finally, all task-receiving VMs that receive extra workloads are removed, i.e. they will not be considered to receive more extra workloads (Line 16). This is done in order to avoid aggressively reassign tasks from many task-giving VMs onto one task-receiving instance. If an instance is able to receive more than one extra workloads, it needs to finish the first one before being considered to receive another.

The Algorithm 6.2 terminates if either all task-giving instances are considered or all task-receiving instances are removed. Notably, the algorithm does not guarantee to reassign all tasks from $|V^g|$ task-giving instances to $|V^r|$ task-receiving VMs. In other words, it is possible that there are not enough task-receiving VMs to handle all workload given by task-giving VMs, even the violating ones. As a result, *Algorithm 6.2 does not guarantee to resolve all potential violations*. One of the ways to resolve potential violations is to add extra VM, and suffer extra cost. However, this approach is not considered in this thesis. Moreover, we argue that potential violation may be resolved automatically, since it is possible for the remaining tasks to be executed faster than estimated due to the performance variation.

6.2 Handling Unknown Applications

In Chapter 5, task execution times are required in order to perform execution scheduling and management. These values can be known for recurring applications. However, when a new application is executed for the first time, its task execution times corresponding to all instance types are not available. In this section, we present an approach for handling **unknown applications**, whose task execution times are not yet available. In our proposed

approach, instead of performing scheduling as a job is submitted, we perform a **sampling phase** in which a small portion of a job is executed on VMs of all available types. After this phase is over, the task execution times are estimated based on the actual execution times. Within the newly estimated task execution times, a job can be scheduled using any of the scheduling approaches proposed in the Chapter 5. This mechanism has already been published in a peer reviewed paper [63].

6.2.1 Determine the Sampling Duration

When a job of an unknown application is submitted, a sampling phase starts. In this phase, some tasks of a job are assigned to an instance of each type for execution. The actual execution time of an task is retrieved and later used to estimate the average task execution of a job's application on each instance type.

The most important factor to consider before running a sampling phase is the **sampling duration**, i.e. how amount of time that a sampling phase lasts. If a duration is too short, only few, or even none, of the sampling tasks are executed, hence the retrieved data is not be sufficient to estimate the average task execution time. On the other hand, if a duration is too long, the full execution phase is delayed further, which results in a short available time to execute tasks of a job. Notably, while calculating this duration, the instance creation overhead must be taken in to account since it may require to create new instances in both the sampling duration, when an instance type has no available instance to receive sampling tasks, and the full execution phase, when new instances are required to execute a job within its deadline.

In our approach, a duration of a sampling phase is calculated as a fraction of a job's available execution time, the amount of time from when a job it submitted to its deadline. For instance, a sampling phase can take 5% or 10% of a job's available execution time. As a result, we assume that when a job of an unknown application is submitted, its available execution time must be large enough to have a sufficient amount of time for a duration of a sampling phase.

Formally, given a job j with available execution time e_j , if the sampling duration takes 10% of a job's available deadline then $e_s = 10\% \times e_j$.

6.2.2 Schedule the Sampling Phase

In a sampling phase, a VM of each instance type is selected in order to execute the sample tasks. Ideally, if it is possible to select only existing instances, there will be no additional cost. However, if there is a type that either has no existing VMs or has existing VMs but

their capacities are not enough to handle sampling tasks, a new instance of that type needs to be created, thus resulting in additional cost. Obviously, this will result in additional cost, however, we argue that it is worthwhile to estimate the task execution times, regardless the overhead. This claim will be verified by the experiments in Chapter 8.

Select VMs for the Sampling Phase

In order to estimate the task execution time between an unknown application and **all** available instance types, tasks must be executed on VMs of all types. In other words, it is necessary to select VMs of each type. Existing VMs are considered to be selected first since they are already paid for, i.e. running a sampling phase on them does not result in additional cost. However, a new instance must be created for a type that does not have any VM available for executing task since either all VMs of that type do not have enough capacity or there is no existing VM of that type.

Algorithm 6.3 Select Sampling VMs

```

1: function ASSIGN_SAMPLING( $e_s$ )
2:    $V_s \leftarrow \emptyset$ 
3:   for  $t \in T$  do
4:      $V_t \leftarrow$  all existing VMs of type  $t$ 
5:      $e_t \leftarrow e_s$ 
6:      $v_s \leftarrow NULL$ 
7:     for  $v \in V_t$  do
8:        $W_v \leftarrow$  all unfinished workloads in  $v$ 
9:        $delay \leftarrow CALCULATE\_DELAY(v, W_v)$ 
10:      if  $delay \geq e_t$  then
11:         $e_t \leftarrow delay$ 
12:         $v_s \leftarrow v$ 
13:      if  $v_s = NULL$  then
14:         $v_s \leftarrow$  new instance of type  $t$ 
15:      Add  $v_s$  to  $V_s$ 
16:   return  $V_s$ 

```

The Algorithm 6.3 presents a logic for selecting VMs for a sampling phase. Its input is a duration of a sampling phase, i.e. e_s . The algorithm loops through each instance type and its existing VMs. In each VMs, the permissible delay of all unfinished workloads, including the current one, is calculated using the *CALCULATE_DELAY* algorithm presented by Algorithm 4.2 in Chapter 4. Only VMs whose the delays are greater than equal to a duration of a sampling phase are able to execute sample tasks. Among those VMs, the one with the highest permissible delay is selected (From Line 7 to Line 12).

On the other hand, if no VM of a given type is selected, a new instance is created (Line 14). Which means a user has to pay for the incurred cost.

Finally, the Algorithm returns the list of VMs that can be used for the sampling phase.

Schedule the Sampling Workloads

After selecting a set of VMs to perform in a sampling phase, the next step is to assign tasks to them. This process is presented in the Algorithm 6.4 whose inputs are the list of VMs used in a sampling phase (V_s), a job (j), and a duration of a sampling phase (e_s).

Algorithm 6.4 Schedule Sampling Phase

```

1: function SCHEDULE_SAMPLING( $V_s, j, e_s$ )
2:    $n' \leftarrow \lfloor \frac{n_j}{|V_s|} \rfloor$ 
3:   for  $v \in V_s$  do
4:      $w \leftarrow$  new workload with  $n'$  tasks of  $j$ 
5:     if  $v$  is new instance then
6:        $st_w \leftarrow NOW + \beta$ 
7:     else
8:        $PREEMPTION(v)$ 
9:        $SHIFT\_WORKLOADS(w_v, e_s)$ 
10:       $st_w \leftarrow NOW$ 
11:       $fi_w \leftarrow NOW + e_s$ 
12:      Assign workload  $w$  to an instance  $v$ 

```

First, the number of tasks assigned for each VM is calculated. Since the task execution times are unknown, tasks are evenly distributed to each VM (Line 2). It should be noted that each instance does not need to execute all the tasks assigned to it. Instead, they just need to execute as many tasks as possible within a sampling phase.

For each instance, a new workload is created (Line 4). If an instance is just created, its ready time, which is calculated by adding the creation overhead to the current time, is a start time of a new workload (Line 6). On the other hand, for an existing instance, a start of a new workload is the current time (Line 6). Notably, pre-emption must be performed on an existing instance in order to stop an execution of the current workload (Line 8). Then, all remaining workloads on an instance is shifted backward in order to create space for a new workload (Line 9).

Next, a finish time of a workload is calculated by adding the current time to its duration (Line 11). Finally, a workload is added to an instance to be executed either immediately (if an instance is already existing) or when it is ready.

Terminate a Sampling Phase and Estimate Task Execution Times

After assigning and start executing sampling tasks on instances, the execution is monitored in order to retrieve the actual execution of each task on an instance.

The sampling phase is completed when either all sampling tasks are executed or the sampling phase times out, i.e. the end of a sampling phase' duration is reached. After that, all the actual execution times of each tasks are used to estimate the task execution time of a job's application on all instance types.

The task execution time of a job's application on an instance type is calculated as the average value of the actual execution times of all sampling tasks assigned to an instance of that type. For example, if an instance is able to execute 3 tasks of a jobs and the actual execution times are 13, 14, and 15 seconds, then the task execution time of an application corresponding to an instance type is $\frac{13+14+15}{3} = 14$ seconds.

However, if an instance is not able to execute even a single task, a very large value is used as the task execution time so that an instance type will not be considered.

After the termination, the task execution time of a new application on all available types are estimated. This value can be used to schedule the remaining tasks of a job using any scheduling approaches presented in Chapter 5.

6.3 Chapter Summary

In this chapter, we have presented two different execution management mechanisms. Dynamic scheduling process detects any potential deadline violations during runtime and performs task reassignment between VMs in order to prevent such violations. Unknown handling mechanism estimates task execution times of an unknown application by performing sampling phase in which some tasks of an application are executed on VMs of all available instance types.

Chapter 7

Design and Implementation

This chapter is going to present how the scheduling approaches and execution management mechanisms introduced in Chapters 3 and 6 are materialised and implemented into a complete software framework. We will discuss the design and implementation of the software framework which is developed in Scala. This system is able to handle the submission of one or more BoT jobs at any time, schedule and manage their execution on cloud VMs. Its objective is to minimise the monetary cost incurred by using cloud resources while ensuring that all jobs are executed by their deadlines.

The main design goal that we are trying to accomplish is **loose coupling**. In other words, we want to make the system consist of many independent components or services, i.e. each service has no knowledge and is not affected by the internal structure of others. The communication between services is done using the Actor Model, supported by the Akka framework, a toolkit for building a concurrent and distributed system [13]. Moreover, this design goal also helps to avoid code duplication which can be caused by reusing mechanisms, for instance the assignment mechanism presented in Chapter 4 is used by both execution scheduling and managing mechanisms.

This chapter is structured as follow. Section 7.1 presents the Data Transfer Objects (DTOs) which represent the domain objects and are used to carry information between components. Section 7.2 presents the general design of our framework and its components. Section 7.3 explains the features supported by the framework in term of the interaction between different components. This chapter is concluded by Section 7.4.

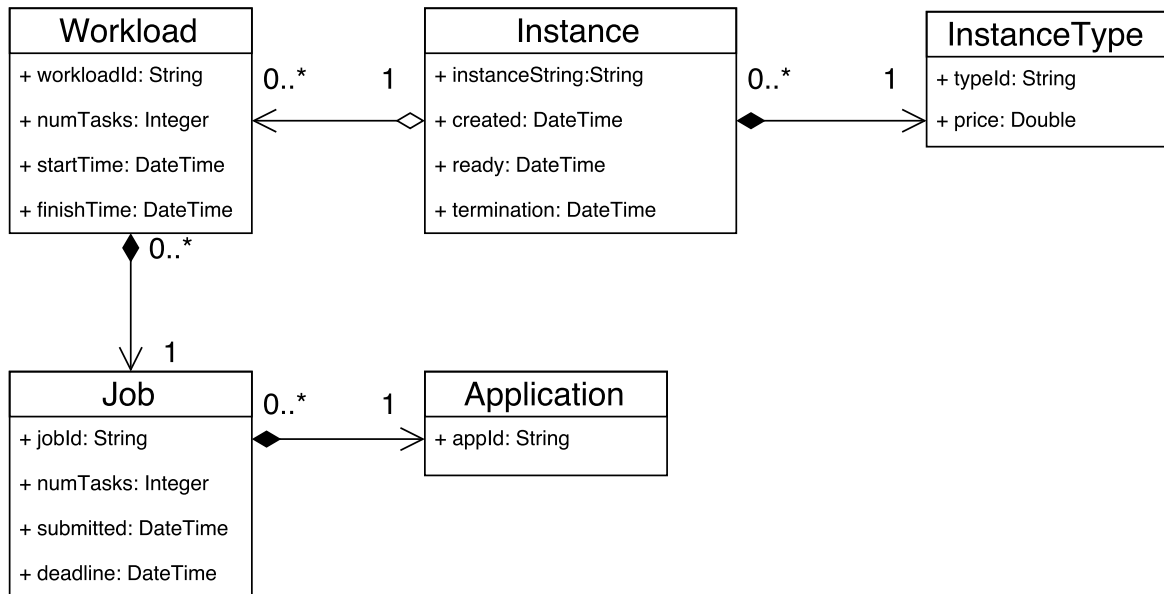


Fig. 7.1 Data Transfer Objects

7.1 Data Transfer Object (DTO)

Data and information are represented by DTOs in order to be stored and transferred between different components. Moreover, the DTOs also represent the domain objects that exist in our framework.

As illustrated by the Figure 7.1, there are total 5 DTOs that are related to each other.

7.1.1 Application

The `Application` class represents BoT application whose jobs are submitted to be executed on the cloud. It only has one attribute, named `appId`, which is unique and used to represent each application.

7.1.2 Job

The `Job` class represents a BoT job. Each `Job` object corresponds to one and only one `Application` object. On the other hand, there can be many jobs of the same application. Besides an unique identifier `jobId`, other attributes of a `Job` are its number of tasks, the submitted time, and a deadline. Notably, the submitted time can represent the time in the future, which can be used to support scheduled jobs.

7.1.3 InstanceType

The `InstanceType` class represents an available instance type offered by the cloud provider. Each `InstanceType` object has a unique identifier `typeId` and a cost per ATU value, denoted as `price`.

7.1.4 Instance

The `Instance` class represents a cloud VM which are created to execute tasks of BoT jobs. Each instance is tied to an `InstanceType` object. Moreover, it also stores the creation, read, and termination time of an instance.

It should be noted that the ready and termination time can be either estimated or actual values. For instance, when an instance is created but not ready yet, its ready time is estimated using the creation overhead β . Later, an instance's ready time is replaced by its actual ready time. Similarly, if an instance is still executing workload at its termination time, the termination time will be delayed, i.e. updated.

7.1.5 Workload

The `Workload` class represents a workload between a job and an instance. As a result, each `Workload` object has to be tied to exactly one `Job` and one `Instance` objects. Similarly, a `Job` object also has a list of its workloads, i.e. W_j . On the other hand, in an `Instance` object, its workloads, i.e. W_v , are divided into two queues containing finished and unfinished workloads. Both queues are sorted based on order of execution, thus, the current workload of an instance is the one in the front of its unfinished workloads queue.

The other attributes of a `Workload` object are its number of tasks, start and finish times. It should be noted that the start and finish times can either be estimated or actual values. For instance, when a workload is scheduled but not yet executed on an instance, its start time is estimated. But since it is possible for a workload to be executed before or after its estimated start time, this value must be updated when an execution starts. Similarly, a workload's finish time can be updated during its execution.

7.2 Components

This section introduces the components, or services, each of which serves a particular role in the system. The supported features are performed based on the interaction between the services and will be discussed later in the next section. The Figure 7.2 illustrates the

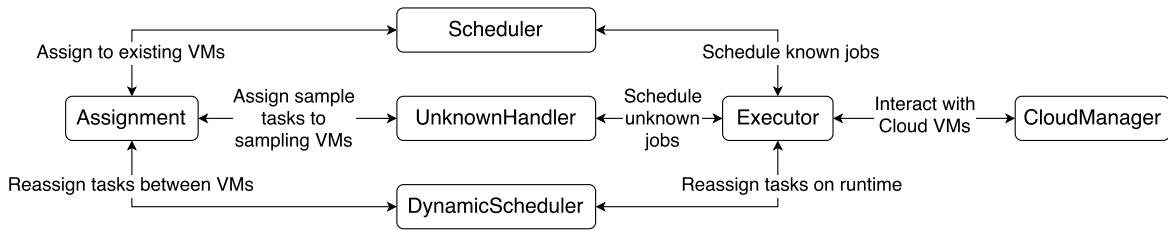


Fig. 7.2 The Components of the Software Framework

components in the software framework and their interaction. There are totally 6 components which are going to be presented and discussed in the following sections.

7.2.1 Assignment Service

The **Assignment Service** component implements all the logics presented in Chapter 4 and are used by other components. More precisely, this component supports assigning tasks of jobs to the set of existing VMs while ensuring that there is no deadline violation.

Assigning tasks from jobs to VMs is performed using Algorithm 4.5. However, for convenience, we break it into two processes and add an extra one, those three processes are presented by three methods as follow:

- `assignJobToVm`: assigning tasks of one job to one VM. More precisely, this method creates a new workload of a given job and assigns it to an instance as long as it does not result in any violation.
- `assignJobToVms`: assigning task of one job to a set of VMs. Given a job, this method loops through each VM and invokes the above `assignJobToVm` method. This feature is used in a hybrid and heuristic scheduling approaches which require to schedule a job to existing VMs before calculating the amount of new instances to rent.
- `assignJobsToVms`: assigning tasks of multiple jobs to a set of VMs. This method loops through each job and invokes the above `assignJobToVms` method. This feature is used in a exact scheduling approach which requires to schedule all jobs to existing VMs before calculating the amount of new instances to rent.

Furthermore, the component also consists of the implementations of the utility methods presented in Section 4.1, which are:

- `findPrecedingAndSucceedingWorkloads`: to find preceding and succeeding workloads in a VM in order to put a new workload in between them (Algorithm 4.1 of Section 4.1.1).

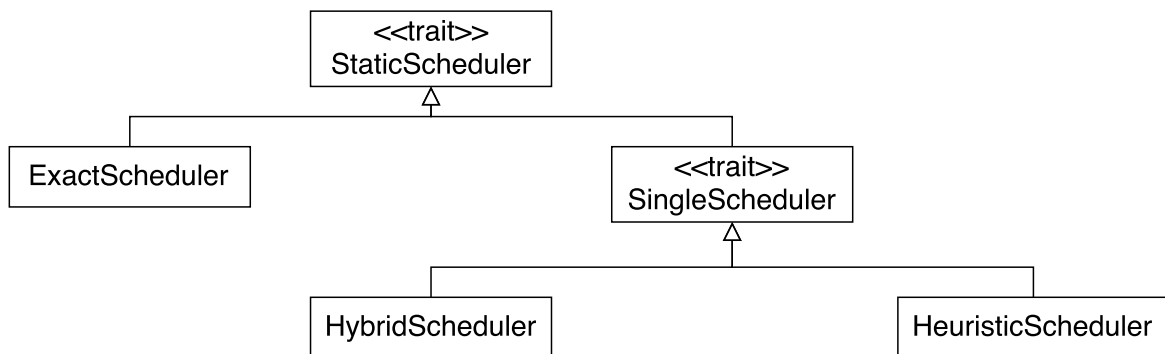


Fig. 7.3 Static Scheduler Hierarchy Structure

- `calculatePermissibleDelay`: to calculate permissible delay (Algorithm 4.2 of Section 4.1.2).
- `shiftWorkloads`: to shift workloads in a VM (Algorithm 4.3 of Section 4.1.3).
- `preempt`: to perform execution pre-emption (Algorithm 4.4 of Section 4.1.4).

7.2.2 Scheduler

The **Scheduler** component implements all the approaches presented in Chapter 5. As we have proposed three different approaches, which are exact, hybrid, and heuristics, the Scheduler component also consists of three different scheduling services. The actual approach is selected based on the configuration parameter which is set when the system starts.

The component's hierarchy is illustrated by the Figure 7.3. `StaticScheduler` is a **trait**, which is a Scala concept similar to Java's interface, which defines the supported behaviours to schedule a set of BoT jobs with deadlines. The signature of the `scheduleJobs` method is presented in the Listing 7.1.

```

def scheduleJobs(jobs: List[Job],
  existingInstances: List[Instance],
  instanceTypes: List[InstanceType],
  taskExecTimes: Map[Application, Map[InstanceType, Double]])
  : Map[Instance, List[Workload]]
  
```

Listing 7.1 `scheduleJobs` Method

The method takes the inputs as i) the list of jobs and their tasks, ii) the list of existing instances, iii) the list of available instance types and their prices, and iv) the task execution times, which are represented as the map from jobs' applications to instance type and then to

the actual values. The output of the method is the map from cloud VMs (both existing and new ones) to the list of new workloads.

The `scheduleJobs` method is implemented by the children of `StaticScheduler`, namely `ExactScheduler` and `SingleScheduler`.

ExactScheduler

The `ExactScheduler` object implements the exact scheduling approach presented in Section 5.1. More specifically, in order to handle a submission of jobs, it first assign them to the existing VMs. Then, the remaining jobs are then scheduled to new VMs based on the Model 5.12.

In order to solve the optimisation problem presented in the Model 5.12, we use Gurobi [12], the optimisation solver. Gurobi provides a Java API to construct an optimisation model as the combination of decision variables, expressions, and constraints. The model is then solved, or optimised, in order to find an optimal solution.

SingleScheduler

The `SingleScheduler` trait is implemented based on the logic presented in Section 5.2. More specifically, each submitted job is first assigned to existing instances before being scheduled on new VMs, which results in additional monetary cost. The process which assigns tasks to existing VMs are implemented within the `SingleScheduler` trait, while the selecting new resources feature will be implemented by its children. This method is named `selectResources` and has the signature as follow:

```
def selectResources(job: Job,
  numTasks: Int,
  instanceTypes: List[InstanceType],
  taskExecutionTimes: Map[InstanceType, Double])
  : List[Instance]
```

Listing 7.2 `selectResources` Method

As shown by Listing 7.2, the inputs of the `selectResources` method are: i) a submitted jobs, ii) its number of *remaining* tasks after assigning to existing VMs, iii) list of available instance types and their prices, and iv) the task execution times of a job's application to all available instance types. The method returns the list of new instances.

The method `selectResources` of the `SingleScheduler` trait is implemented by two objects `HybridScheduler` and `HeuristicScheduler`, corresponding to the hybrid and heuristic scheduling approaches respectively.


```
samplingDuration: Int,
allInstanceTypes: List[InstanceType])
:(List[Task], Map[Instance, List[Workload]])
```

Listing 7.4 `scheduleUnknownJob` Method

The method will perform two steps. The first step is to select a VM of each available type as a sampling VM. As presented in the Algorithm 6.3, the method tries to select existing VM and only creates new VM if necessary. The second step is to schedule a sampling execution on all sampling VMs, as presented by the Algorithm 6.4. This step also sets a timeout for a sampling phase to end.

7.2.5 Cloud Manager

The **Cloud Manager** component provides the features to interact with the cloud. This component is in charge of managing not only the cloud VMs but also the workload execution within each VM. Moreover, the Cloud Manager component provides the interfaces which need to be implemented for each cloud platform.

Manage Cloud VMs

In order to create new VMs, the Cloud Manager supports a `createInstance` method which receives an instance type as input. Its output is a running instance, as shown by the Listing 7.5.

```
def createInstance(instanceType: InstanceType) : Instance
```

Listing 7.5 `createInstance` Method

The termination feature requires an instance and does not return anything, as shown by the Listing 7.6. Notably, there is a flag, i.e. Boolean value, named `forcedTermination` which indicates whether a termination must be done. If this value is set to `false`, a termination may not be performed if an instance is still executing workload. Instead, an instance's termination is delayed for another ATU which results in additional cost.

```
def terminateInstance(instance: Instance, forcedTermination: Boolean)
```

Listing 7.6 `terminateInstance` Method

It should be noted that when an instance is created, it is also scheduled to be terminated at the end of its ATU. In other words, an VM's termination is triggered automatically when time out. Since we do not allow assigning workload whose finish time exceeds an instance's terminated time, an instance's termination should not be changed. However, if an instance is

not yet idle, i.e. is still executing tasks, at its termination time, its termination is delayed for another ATU, which results in unavoidable additional cost.

Manage Workload Execution

In order to start a new workload on an instance, the Cloud Manager provides a `startWorkload` method whose input is an instance. An instance will automatically start a workload at a top of the unfinished workloads queue, which is sorted by their deadlines.

```
def startWorkload(instance: Instance)
```

Listing 7.7 `startWorkload` Method

In order to move tasks between workloads during execution, it is necessary to support removing tasks from a workload. This method, named `removeTasks` is presented by the Listing 7.8. Its input is the Instance object, the Workload, and the list of tasks to be removed.

```
def removeTasks(instance: Instance, workload: Workload, removedTasks:
    List[Task])
```

Listing 7.8 `removeTasks` Method

In order to support execution management during runtime, it is necessary to support a monitoring functionality. The Cloud Manager component provides the `monitorInstance` method whose signature is presented by the Listing 7.9.

```
def monitorInstance(instance: Instance): (String, List[Int])
```

Listing 7.9 `monitorInstance` Method

Taking an instance as a input, the `monitorInstance` returns a pair whose first value is a identifier of the current workload. The second value of a returned pair is a list whose items are the actual times that tasks of a current workload are executed. This list serves two purposes: i) the number of its items is the number of executed tasks, ii) the actual execution times can be used to estimate task execution time while handling unknown application.

7.2.6 Executor

The **Executor** component is a centralised manager which not only supports but also connects other components. It has many features that will be presented in the next few sections.

Receiving Submission

Before job submissions are sent to the Scheduler component, they must be received by the Executor. In this framework, we assume that all the required library and data are already available in the cloud, e.g. they can be stored in cloud storing service such as AWS S3 [14] or Azure Storage [15]. Hence, a job submission can be simply expressed by stating a job id, its application id, the number of tasks, the submitted time and deadline. In our framework, we use JSON to represent job submission.

```
{
  "jobs": [
    {
      "jobId": "Job_1",
      "appId": "Application_1",
      "numTasks": 30,
      "expectedStartTime": "now",
      "deadline": 1200
    },
    {
      "jobId": "Job_2",
      "appId": "Application_2",
      "numTasks": 50,
      "expectedStartTime": "now" + 1200,
      "deadline": 1500
    }
  ]
}
```

Listing 7.10 Job Submission JSON File Example

The Listing 7.10 presents an example of a JSON file that consist of two jobs. The first one named `Job_1` belongs to an application `Application_1` and has 30 tasks. A job needs to be scheduled and executed immediately and its deadline is 1200 seconds from now. The second one named `Job_2` belongs to an application `Application_2` and has 50 tasks. A job needs to be scheduled and executed 1200 from now, thus `"now" + 1200`. Its deadline is 1500 seconds after its start time, which is $1200 + 1500 = 2700$ seconds from now.

Using the JSON file, we are able to not only submit jobs to be executed immediately but also schedule job execution in the future, e.g. recurring jobs.

Caching Data

Since all other components are stateless, i.e. they do not store data, the Executor store all the data in the series of caches. For instance, there are five caches corresponding to

five DTOs classes, namely `ApplicationCache`, `JobCache`, `InstanceTypeCache`, `InstanceCache`, and `WorkloadCache`.

Since we assume that the available instance types and their prices are available prior to the execution and unchanged, the `InstanceTypeCache` is populated when the framework started. The data is stored in a JSON file whose format is illustrated by the Listing 7.11.

```
{
  "instanceTypes": [
    {
      "instanceTypeName": "m3.medium",
      "price": 0.073
    },
    {
      "instanceTypeName": "m3.large",
      "price": 0.146
    },
    {
      "instanceTypeName": "m3.xlarge",
      "price": 0.293
    }
  ]
}
```

Listing 7.11 A JSON File Which Stores Instance Types and Their Prices

Similarly, the `ApplicationCache` can be populated based on predefined data in form of a JSON file, as presented by the Listing 7.12. However, this cache can also be updated at runtime when unknown jobs are submitted.

```
{
  "applications": [
    {
      "applicationName": "COMPRESS"
    },
    {
      "applicationName": "MOLECULAR"
    },
    {
      "applicationName": "SVM"
    }
  ]
}
```

Listing 7.12 A JSON File Which Stores Applications

Jobs are added to the `JobCache` when they are submitted to the system. Similarly, instances and workloads are added to the `InstanceCache` and `WorkloadCache` when they are created in order to execute jobs' tasks and are updated during runtime.

Moreover, the `Executor` also maintains the `TaskExecutionTimes` cache which stores a task execution times between applications and instance types. This cache can be populated using pre-defined data or during runtime by the unknown application handling process. The Listing 7.13 presents a JSON file which store task execution times data. For example, a task execution time for a *COMPRESS* application on VM of type *m3.medium* is 44.5 seconds.

```
{
  "taskExecTimes": [
    {
      "instanceTypeName": "m3.medium",
      "applicationName": "COMPRESS",
      "taskExecTime": 44.5
    },
    {
      "instanceTypeName": "m3.medium",
      "applicationName": "MOLECULAR",
      "taskExecTime": 60.0
    },
    {
      "instanceTypeName": "m3.medium",
      "applicationName": "SVM",
      "taskExecTime": 22.65
    },
    {
      "instanceTypeName": "m3.large",
      "applicationName": "COMPRESS",
      "taskExecTime": 21.35
    },
    {
      "instanceTypeName": "m3.large",
      "applicationName": "MOLECULAR",
      "taskExecTime": 20.0
    },
    {
      "instanceTypeName": "m3.large",
      "applicationName": "SVM",
      "taskExecTime": 13.35
    },
    {
      "instanceTypeName": "m3.xlarge",
```

```
        "applicationName": "COMPRESS",
        "taskExecTime": 16.0
    },
    {
        "instanceTypeName": "m3.xlarge",
        "applicationName": "MOLECULAR",
        "taskExecTime": 10
    },
    {
        "instanceTypeName": "m3.xlarge",
        "applicationName": "SVM",
        "taskExecTime": 13.3
    }
]
}
```

Listing 7.13 A JSON File Which Stores Task Execution Times between Instance Types and Applications

Finally, in order to support handling unknown applications, there is a cache named `SamplingCache` which stores the actual execution times of unknown applications on all available instance types.

Connecting Other Components

The final features of the Executor Component is to be a middleman connecting other components. As presented in the previous sections, other components are loosely coupled and have different objective, hence, it is necessary to have the Executor to coordinate the execution by invoking services based on the predefined order.

7.3 Supported Features

In this section, we are going to present and discuss all the features supported by the software framework. The supported features are presented in form of processes, each of which involves the interaction between components introduced in the Section 7.2.

7.3.1 Submission Handling

Figure 7.4 is the sequence diagram illustrating the job submission handling process. There are four components that involve in this process: the Executor, Scheduler, CloudManager, and UnknownHandler.

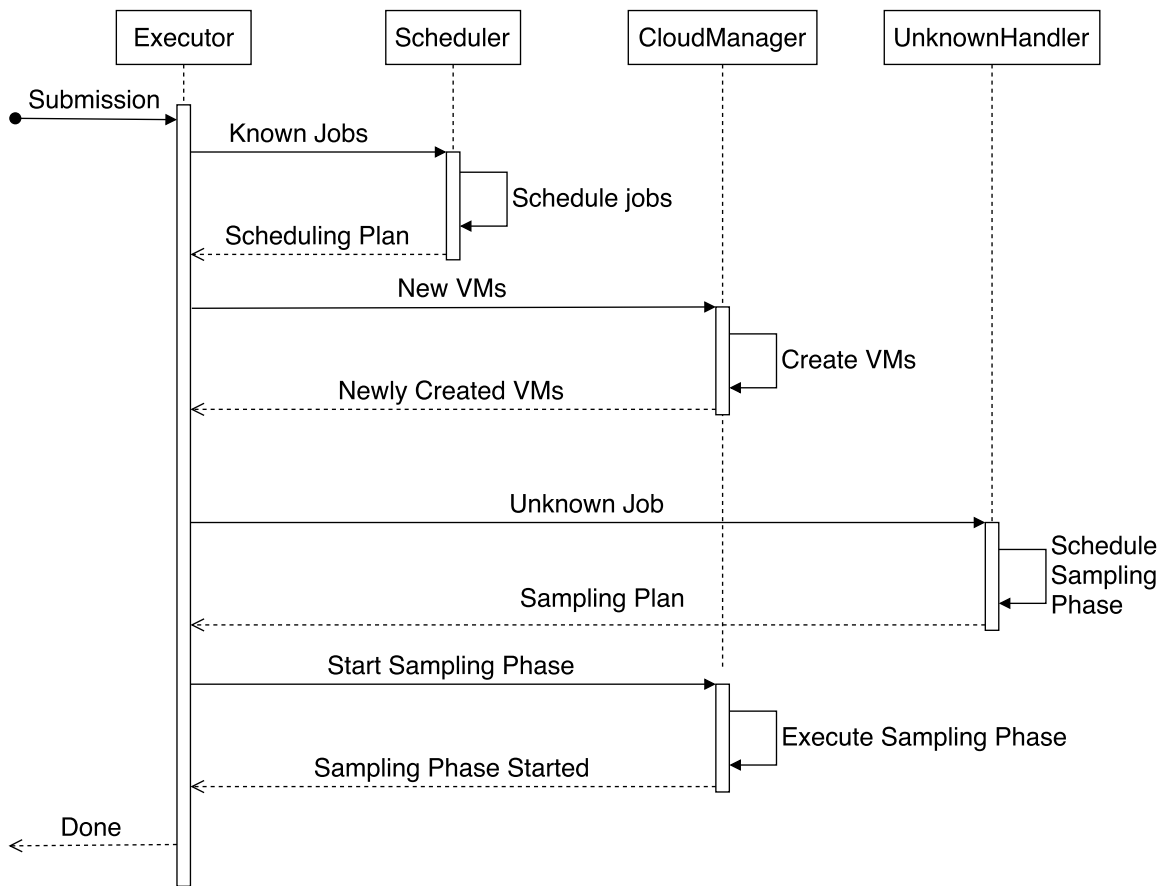


Fig. 7.4 Job Submission Handling Process

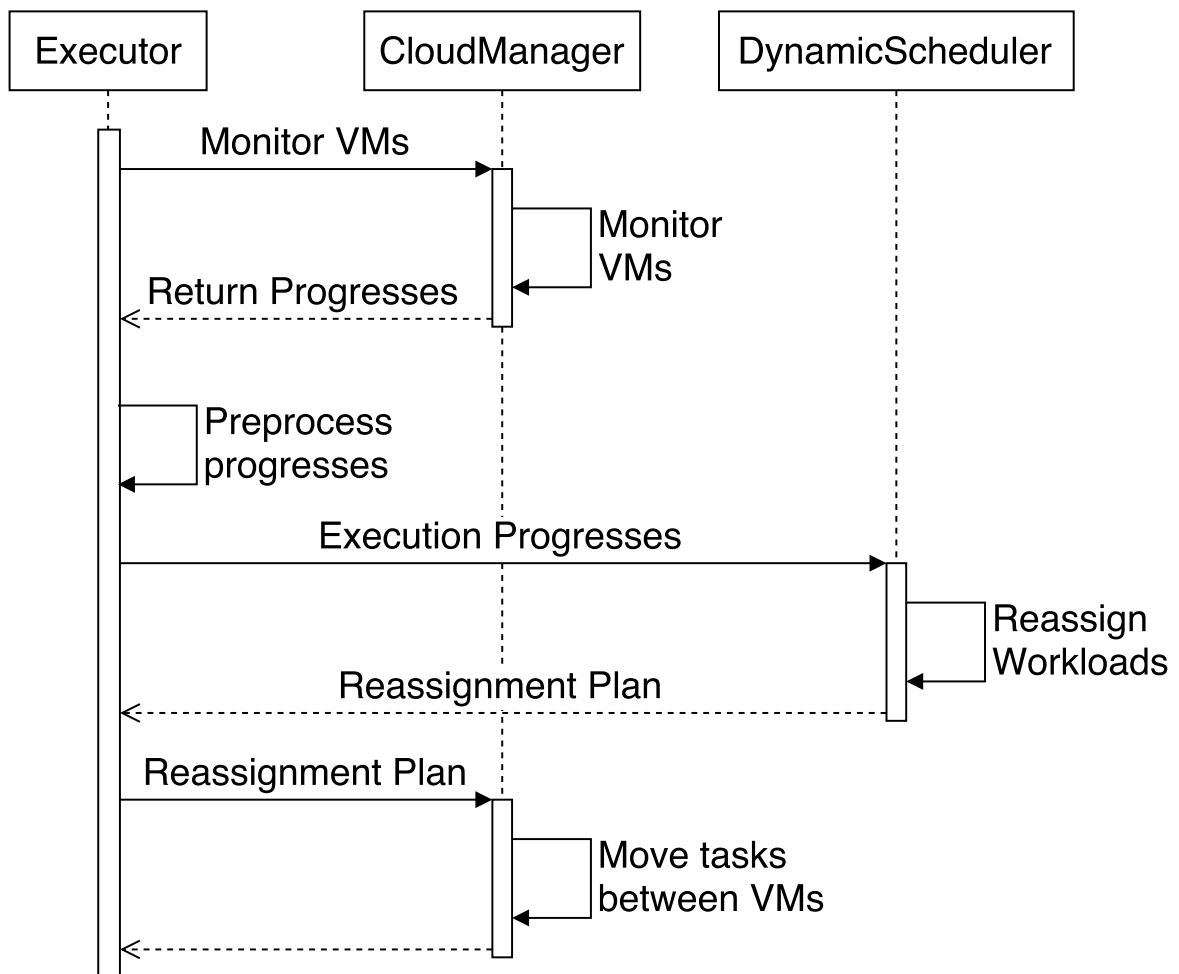


Fig. 7.5 Execution Management Process

The process starts when one or more jobs are submitted to the system via the Executor component.

First, all the known jobs, whose tasks execution times are known, are sent to the Scheduler component to be assigned to existing VMs and/or scheduled to new ones. The result of the scheduling process, i.e. the scheduling plan, is returned to the Executor.

After that, the scheduling plan is sent to the CloudManager to either create new VMs if necessary or start executing tasks on existing ones.

After all the known jobs are scheduled, the unknown one is sent to the UnknownHandling component which schedules a sampling phase. The result, i.e. the sampling plan, is sent back to the Executor which again forwards to the CloudManager to start a sampling phase.

7.3.2 Execution Monitoring and Management

Figure 7.5 presents the sequence diagram illustrating the execution management process. It starts when the Executor asks the CloudManager for execution progresses, which can be triggered periodically.

After receiving all the VM progresses from the CloudManager, the Executor pre-processes them, e.g. performing progress categorisation, before sending the result to the DynamicScheduler module. This module performs dynamic reassignment by moving tasks from violating VMs to those that can receive extra workloads. The result, which is called reassignment plan, is sent back to the Executor which forwards to the CloudManager.

7.4 Chapter Summary

In this chapter, we have presented the design and implementation details of the software system which incorporates all the research presented in the previous chapters in order to schedule the execution of BoT jobs on the cloud.

The proposed system consists of many loosely coupled services which interacting with each other in order to perform supported features. The interaction is performed using the Actor model supported by the Akka framework. Data are stored and transferred using DTOs. We employ Gurobi, the off-the-shelf mathematical solver, in order to solve optimisation problems and produce exact solutions.

The main drawback of our framework is its centralised nature, more specifically, all the communication must go through the Executor component which becomes the single point of failure. Investigating the way to decentralise the Executor component will be investigated in the future work.

Chapter 8

Evaluation

8.1 Introduction

In this chapter, our proposed research and framework are thoroughly evaluated via the set of different experiments. First, we perform simulation experiment to evaluate different mechanisms presented in Chapters 5 and 6 individually (Sections 8.2, 8.3, and 8.4). It should be noted that by running our experiments in the simulated environment, our results were exposed to a certain degree of bias due to the fact that the simulated environment could not completely replicate the behaviours of the real cloud environment. As a result in Section 8.5, we evaluated our proposed research against AWS cloud environment in order to demonstrate its applicability in the real world.

A majority of this chapter has been presented in the peer reviewed papers [63, 64].

8.2 Comparing the Scheduling Approaches

In Chapter 5 we presented three different scheduling approaches, namely Exact Approach, Hybrid Approach, and Heuristic Approach.

The Exact Approach aims to find a global optimal solution. More specifically, it considers all submitted jobs at the same time and produces a fine-grained scheduling plan which assigns the certain number of tasks of each job to each cloud VMs. As a result, the Exact Approach can achieve the lowest cost possible. However, it can take a considerable amount of time in order to find a scheduling plan.

The Hybrid Approach aims to find an optimal solution for each job and combines them using a heuristic algorithm. In other words, this approach considers one job at a time and applies Integer Linear Programming techniques to calculate the amount of VMs of each

instance type that can execute a job with the lowest cost. Hence, even though this approach is able to achieve the optimal solution for each job, it does not guarantee the scheduling plan for all jobs are optimal. On the other hand, we believe it can be solved faster compared to the Exact Approach.

The Heuristic Approach considers one job at a time, similar to the Hybrid Approach. However, it uses the greedy and best-effort algorithm in order to find the scheduling plan with lowest cost possible. In other words, the optimality of the solution is not guaranteed. We believe this approach will have the lowest solving time, however, the resulted monetary cost is highest compared to the other approaches.

The three approaches present the trade-off between the solving time, the time it takes to find a scheduling plan, and the optimality of the solutions. This trade-off will be thoroughly investigated by the experiments performed in this section. The objective of this section is to show the actual difference in solving time and solution optimality between the three approaches. Furthermore, we also aim to investigate the influence of different parameters on the scheduling problem.

8.2.1 Environment Set-up

Programmatically speaking, this section solely focuses on testing the Scheduling service presented in Section 7.2.2. More specifically, the experiment is performed by invoking the `scheduleJobs` methods of three services: `ExactScheduler`, `HybridScheduler`, and `HeuristicScheduler`. In order to compare the three approaches, we use the following two metrics: i) the time it takes to produce a scheduling plan, i.e. solving time, and ii) the cost of the returned scheduling plan, i.e. solution optimality.

Dependent and Independent Variables

In our experiment, there are two sets of variables. The first one is related to the input, i.e. cloud environment and problem, and are called **independent variables**. The independent variables used in our experiments are i) number of jobs, ii) number of tasks, iii) number of available instance types, and iv) the job deadline. In our experiment, we assume that the number of applications is equal to the number of jobs. The other set of variables is called **dependent variables** and consists of i) solving time, and ii) the cost of a scheduling plan. Any changes in the independent variables will affect the dependent variables.

Each independent variable is assigned a default value, which is used as a baseline. In each experiment, we change only one independent variable while keeping the remaining variables unchanged. As a result, we are able to not only compare the solving time and

Table 8.1 Summary of the Independent and Dependent Variable

Name	Type	Default	Description
Number of Jobs	Independent	5	The number of jobs
Number of Tasks	Independent	100	The number of tasks per job
Number of Type	Independent	4	The number of available instance types
Deadlines	Independent	600	The amount of time in which a job must be fully executed
Solving Time	Dependent	N/A	The time it takes to find a scheduling plan for a given problem
Total cost	Dependent	N/A	The cost of a returned scheduling plan

solution optimality between three approaches but also investigate the affect of the independent variables on them.

Table 8.1 summarises all the independent and dependent variables used in this section. It also shows the default values of the independent variables. In other words, when one independent variable is changed, the others are set to their default values.

Input Synthesiser

In order to evaluate and compare the scheduling approaches, we need to generate the input which consists of available instance types, application, jobs (and their tasks), and the task execution times between instance types and applications. Automatically generate instance types, applications, and jobs based on the set of given independent variables is simple, however, generating task execution times is more complicated.

We develop the task execution generator, which automatically generates task execution times of given application and instance types. The main idea is to calculate a task execution time between an application and an instance type based on the task execution time of the same application on the nearest cheaper instance type. The logic is presented by Algorithm 8.1 which generate the task execution time between an application an all instance types.

Algorithm 8.1 Generate Task Execution Times

- 1: **function** GENERATE_TASK_EXEC_TIMES(a, T)
 - 2: Sort T by their prices in ascending order
 - 3: $t_0 \leftarrow$ first type in T
 - 4: $e_{a,t_0} \leftarrow$ a random value
 - 5: **for** $i \in \{1 \dots |T| - 1\}$ **do**
 - 6: $\delta \leftarrow$ random number between 0.1 and $\frac{p_{t_i}}{p_{t_{i-1}}}$
 - 7: $e_{a,t_i} \leftarrow \delta \times (e_{a,t_{i-1}} \times \frac{p_{t_{i-1}}}{p_{t_i}})$
-

First, the instance types are sorted based on their prices in ascending order (Line 8.1), which means the first type on the list, denoted as t_0 , has the cheapest price. The execution time of an application on the cheapest type is assigned to a random positive value (Line 4).

Then, the algorithm loops through the remaining instance type. For each type, its task execution time is calculated based on the *performance improvement factor* δ which *represents the relationship between the task execution times and prices of two instance types* t_1 and t_2 , assuming $p_{t_1} < p_{t_2}$, as follow:

$$\frac{e_{t_2}}{e_{t_1}} = \delta \times \frac{p_{t_1}}{p_{t_2}} \quad (8.1)$$

Hence, it is possible to calculate a task execution time of t_2 given a task execution time of t_1 and their prices:

$$e_{t_2} = \delta \times e_{t_1} \times \frac{p_{t_1}}{p_{t_2}} \quad (8.2)$$

Based on Formula 8.2, it is possible to make the following remarks:

- If $\delta = 1$, then $e_{t_2} = e_{t_1} \times \frac{p_{t_1}}{p_{t_2}}$, which means the decrease in task execution time is equal to the increase in price. For example, if $p_{t_2} = 2 \times p_{t_1}$ then $e_{t_2} = e_{t_1}/2$. In this case, it makes almost no difference selecting two VMs of t_1 or one VM of t_2 .
- If $\delta < 1$, then $e_{t_2} < e_{t_1} \times \frac{p_{t_1}}{p_{t_2}}$, which means the decrease in task execution time is greater than the increase in price. For example, if $p_{t_2} = 2 \times p_{t_1}$ then $e_{t_2} < e_{t_1}/2$. In this case, selecting one VMs of t_2 is more cost effective than selecting two VMs of t_1 .
- If $\delta > 1$, then $e_{t_2} > e_{t_1} \times \frac{p_{t_1}}{p_{t_2}}$, which means the decrease in task execution time is less than the increase in price. For example, if $p_{t_2} = 2 \times p_{t_1}$ then $e_{t_2} > e_{t_1}/2$. In this case, selecting two VMs of t_1 is more cost effective than selecting one VM of t_2 .
- If $\delta = \frac{p_{t_2}}{p_{t_1}}$, then $e_{t_2} = e_{t_1}$, which means a task execution time remains the same between two instance types. For example, even if $p_{t_2} = 2 \times p_{t_1}$, e_{t_2} and e_{t_1} are identical. In this case, selecting two VMs of t_1 is more cost effective than selecting one VM of t_2 .

In other words, δ represents the performance/cost trade-off between two instance types. For each instance type, δ takes a random value between 0.1 and $\frac{p_{t_i}}{p_{t_{i-1}}}$, i.e. the ratio between its price and the previous type's (Line 6). It is then used to calculate a task execution time of an instance type (Line 7).

Algorithm 8.1 is applied for each application. The final result is the task execution times between all given applications and all available instance types.

8.2.2 Experiment Results and Discussion

We performed four sets of experiment. In each set, one independent variable varied while the rest were set to the default values presented in Table 8.1. The experiment was performed five times for each value of the selected independent variable. The dependent variables, i.e. solving time and total cost, were recorded after each run.

Varied Numbers of Jobs

In this set of experiment, we selected 10 different values corresponding to the different numbers of job which varied from 5 to 14. The results are illustrated by Figure 8.1.

Figure 8.1a presents the solving times of three approaches when the number of jobs varies. The x-axis shows the number of jobs which goes from 5 to 14. The y-axis shows the solving time in milliseconds. Notably, log-10 scale is used for the y-axis.

It is evident that the solving times of the Exact approach are higher than the Hybrid approach's which are higher than the solving times of the Heuristic approach. However, the difference between the Exact and Hybrid approaches is substantial compared to the difference between the Hybrid and Heuristic approaches. For instance, when there are 5 jobs, the solving time of the Exact approach (804.2 milliseconds) is nearly 14 times higher the solving time of the Hybrid approach (58.4 milliseconds), which is only 1.4 times higher than the solving time of the Heuristic approach (41 milliseconds). Furthermore, when there are 14 jobs, the solving time of the Exact approach (545362.8 milliseconds) is about 7400 times higher the solving time of the Hybrid approach (73 milliseconds), which is only 1.2 times higher than the solving time of the Heuristic approach (62.4 milliseconds).

The Exact approach's solving time not only are higher than other approaches but also suffers substantial growth. As the number of jobs increases from 5 to 14, the solving time of the Exact approach grows 67714%, from 804.2 milliseconds to more than 540000 milliseconds, i.e. more than 9 minutes. On the other hand, the solving times of the Hybrid and Heuristic approaches only increase 25% and 52% respectively.

Figure 8.1b presents the total cost of the scheduling plans produced by the three approaches when the number of jobs varies. The x-axis shows the number of jobs while the y-axis shows the total cost. It can be seen that as the number of jobs increases, the cost increases as well.

The Exact approach always manages to find the scheduling plan with the cheapest cost. Interestingly, the costs of Hybrid and Heuristic approaches are nearly identical. In average, the cost of using the Exact approach is 3.5% and 4.9% cheaper compared to the Hybrid and

Heuristic approach respectively. Using the Hybrid approach results in the cost which is 1.1% cheaper compared to the Heuristic approach.

Varied Numbers of Tasks

In this set of experiment, we selected 10 different values corresponding to the different numbers of tasks which varied from 100 to 1000. Since there are 5 jobs by default, the total number of tasks varies from 500 to 5000 tasks. The results are illustrated by Figure 8.2.

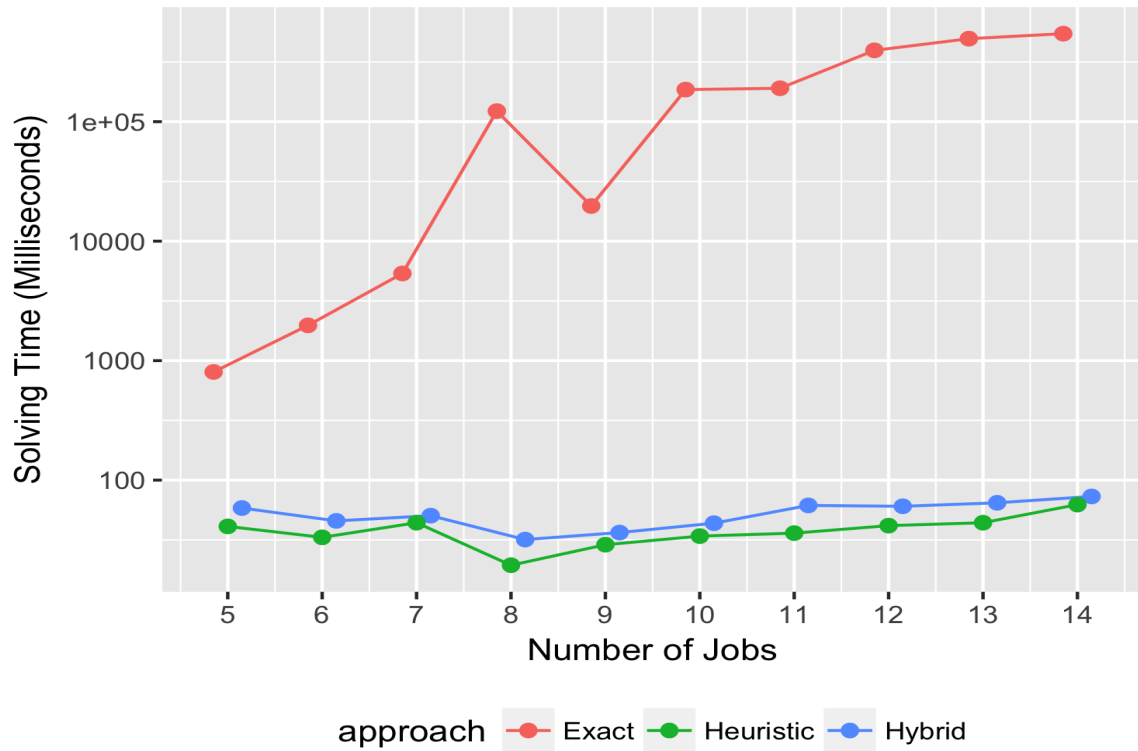
Figure 8.2a presents the solving times of three approaches when the number of jobs varies. The x-axis shows the number of tasks which goes from 500 to 5000. The y-axis shows the solving time in milliseconds. Notably, log-10 scale is used for the y-axis.

It is evident that the solving times of the Exact approach are higher than the Hybrid approach's which are higher than the solving times of the Heuristic approach. However, the difference between the Exact and Hybrid approaches is substantial compared to the difference between the Hybrid and Heuristic approaches. For instance, when there are 500 tasks, the solving time of the Exact approach (927 milliseconds) is more than 15 times higher the solving time of the Hybrid approach (58.8 milliseconds), which is only 1.6 times higher than the solving time of the Heuristic approach (37.4 milliseconds). Furthermore, when there are 5000 tasks, the solving time of the Exact approach (259813.2 milliseconds) is nearly 3000 times higher the solving time of the Hybrid approach (86.4 milliseconds), which is only 1.05 times higher than the solving time of the Heuristic approach (82.6 milliseconds).

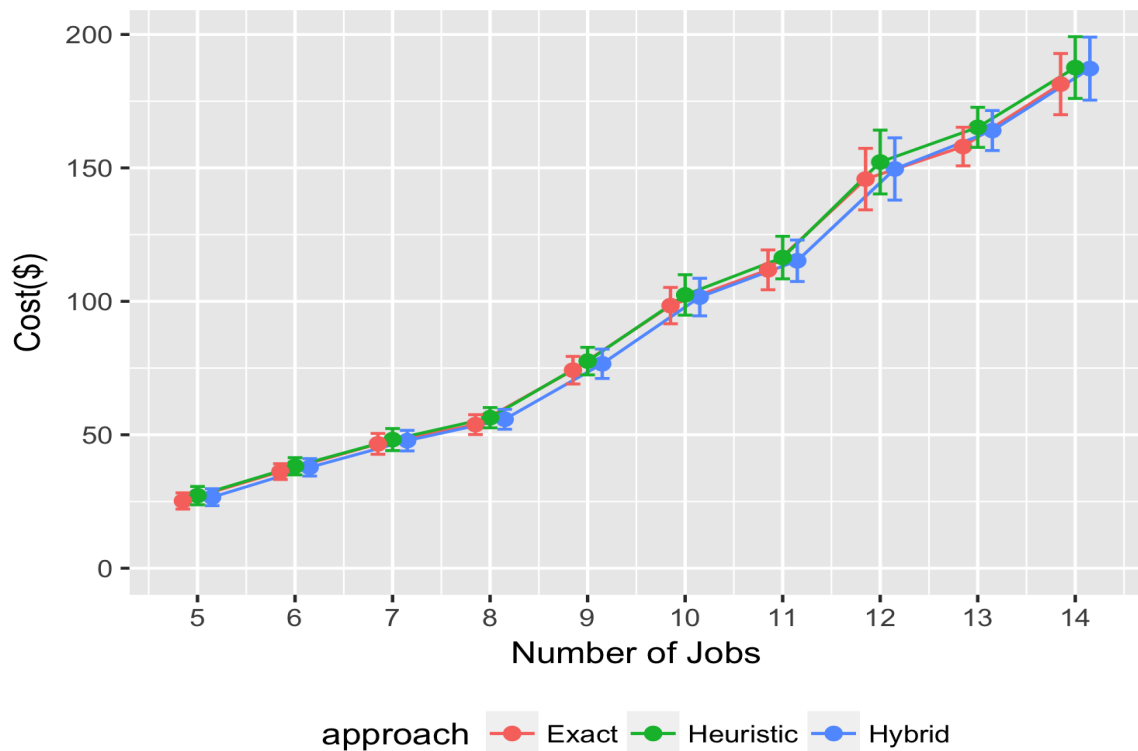
The Exact approach's solving time not only are higher than other approaches but also suffers substantial growth. As the number of tasks increases from 5 to 14, the solving time of the Exact approach grows 27900%, from 927 milliseconds to more than 250000 milliseconds, i.e. more than 4 minutes. On the other hand, the solving times of the Hybrid and Heuristic approaches only increase 47% and 120% respectively.

Figure 8.2b presents the total cost of the scheduling plans produced by the three approaches when the number of tasks varies. The x-axis shows the number of tasks while the y-axis shows the total cost. It can be seen that as the number of tasks increases, the cost increases as well.

Figure 8.2b shows that the total costs of three approaches are almost identical. In average, the cost of using the Exact approach is 1.8% and 2.5% cheaper compared to the Hybrid and Heuristic approach respectively. Using the Hybrid approach results in the cost which is 0.7% cheaper compared to the Heuristic approach.

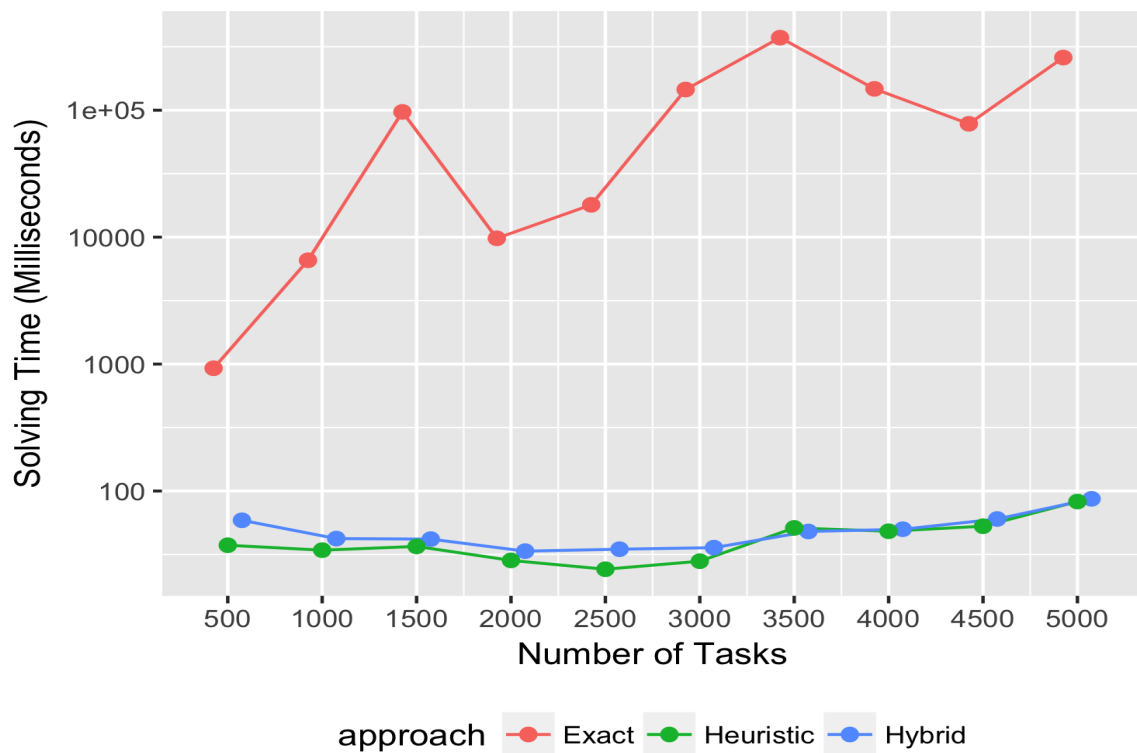


(a) Average solving times of three scheduling approaches when the number of jobs varies. The x-axis presents the number of jobs, the y-axis presents the solving time in log scale.

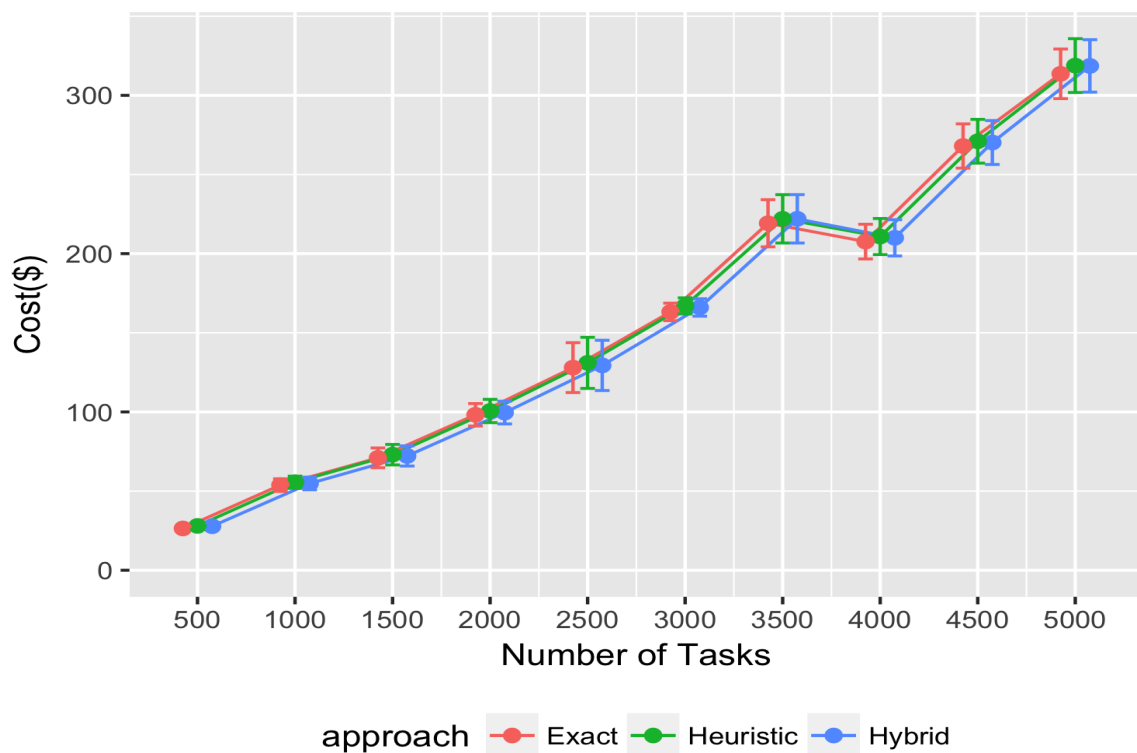


(b) Average cost of three scheduling approaches when the number of jobs varies. The x-axis presents the number of jobs, the y-axis presents the total cost of a cloud cluster constructed to execute all the jobs. The error bars illustrate the standard errors.

Fig. 8.1 Experiment Result of Three Scheduling Approaches When the Number of Jobs Varies



(a) Average solving times of three scheduling approaches when the number of tasks varies. The x-axis presents the number of tasks, the y-axis presents the solving time in log scale.



(b) Average cost of three scheduling approaches when the number of tasks varies. The x-axis presents the number of tasks, the y-axis presents the total cost of a cloud cluster constructed to execute all the jobs. The error bars illustrate the standard errors.

Fig. 8.2 Experiment Result of Three Scheduling Approaches When the Number of Tasks Varies

Varied Numbers of Instance Type

In this set of experiment, we selected 10 different values corresponding to the different numbers of instance types which varied from 5 to 14. The results are illustrated by Figure 8.3.

Figure 8.3a presents the solving times of three approaches when the number of jobs varies. The x-axis shows the number of types which goes from 5 to 14. The y-axis shows the solving time in milliseconds. Notably, log-10 scale is used for the y-axis.

It is evident that the solving times of the Exact approach are higher than the Hybrid approach's which are higher than the solving times of the Heuristic approach. However, the difference between the Exact and Hybrid approaches is substantial compared to the difference between the Hybrid and Heuristic approaches. For instance, when there are 5 instance type, the solving time of the Exact approach (1611.6 milliseconds) is more than 26 times higher the solving time of the Hybrid approach (60.8 milliseconds), which is only 1.3 times higher than the solving time of the Heuristic approach (44.6 milliseconds). Furthermore, when there are 14 instance type, the solving time of the Exact approach (1494 milliseconds) is more than 37 times higher the solving time of the Hybrid approach (39.8 milliseconds), which is only 1.7 times higher than the solving time of the Heuristic approach (23.2 milliseconds).

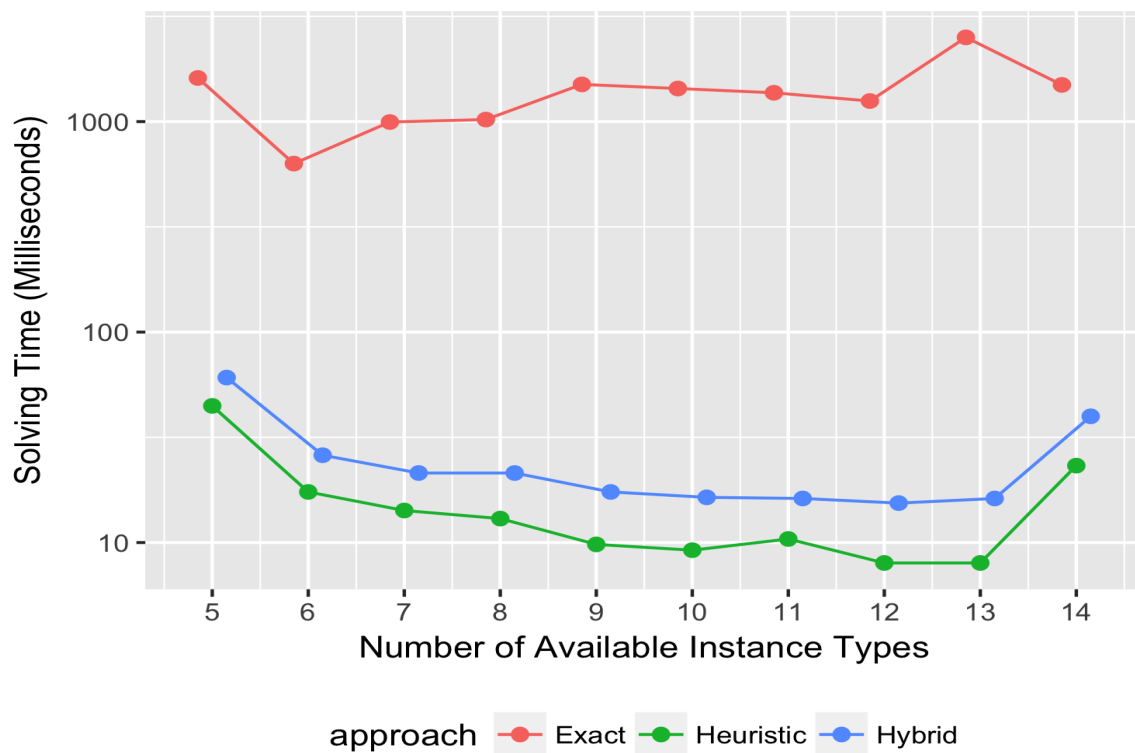
Figure 8.3a also shows that increasing the number of available instance types does not significantly increase the solving time. In other words, the number of instance type does not affect the solving time as much as the number of jobs and the number of tasks do.

Figure 8.3b presents the total cost of the scheduling plans produced by the three approaches when the number of tasks varies. The x-axis shows the number of instance types while the y-axis shows the total cost. Similar to the solving time, the total cost seems to be not affected by the increase in the number of instance types. Notably, the plot seems to vary due to the random process that creates applications, jobs, and task execution times.

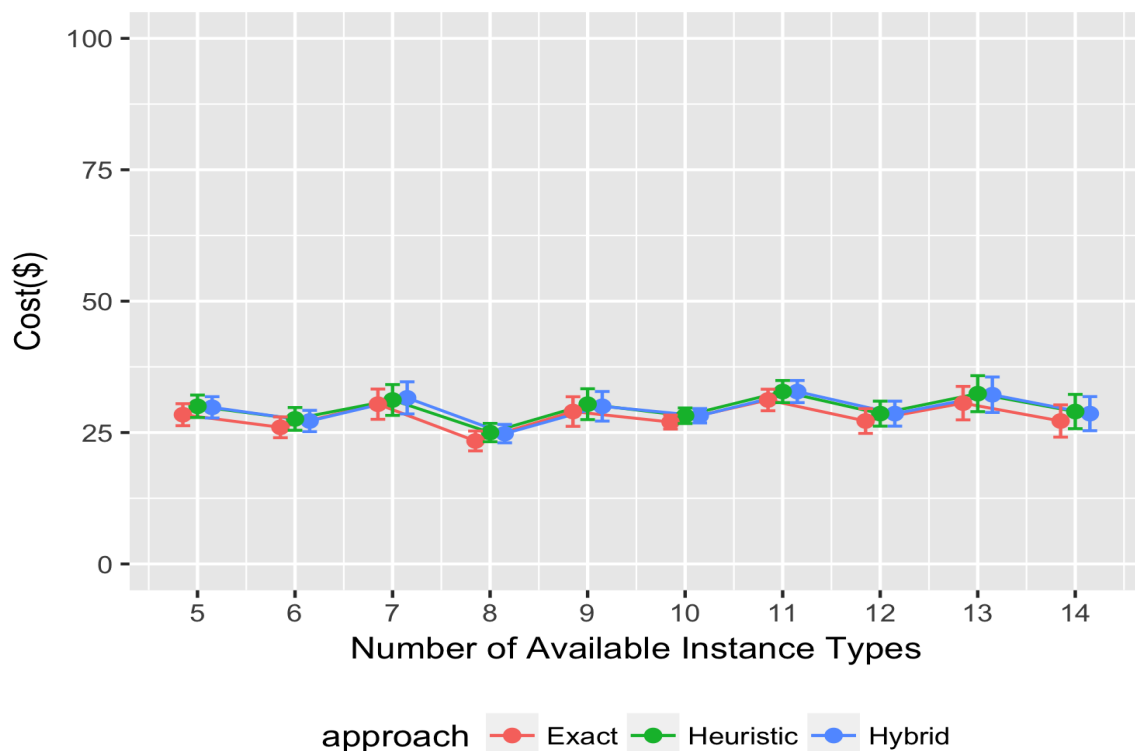
The figure shows that the Exact approach always has the lowest cost in comparison to the other approaches, whose costs are almost identical. In average, the cost of using the Exact approach is 4.8% and 5.3% cheaper compared to the Hybrid and Heuristic approach respectively. Using the Hybrid approach results in the cost which is 0.5% cheaper compared to the Heuristic approach.

Varied Deadlines

In this set of experiment, we selected 10 different values corresponding to the different deadlines which varied from from 600 to 1500 seconds. More specifically, each value



(a) Average solving times of three scheduling approaches when the number of instance types varies. The x-axis presents the number of instance types, the y-axis presents the solving time in log scale.



(b) Average cost of three scheduling approaches when the number of instance types varies. The x-axis presents the number of instance types, the y-axis presents the total cost of a cloud cluster constructed to execute all the jobs. The error bars illustrate the standard errors.

Fig. 8.3 Experiment Result of Three Scheduling Approaches When the Number of Instance Types Varies

denoted the amount of time within which all tasks of a job must be fully executed. The results are illustrated by Figure 8.4.

Figure 8.4a presents the solving times of three approaches when the number of jobs varies. The x-axis shows the deadline which goes from 600 to 1500. The y-axis shows the solving time in milliseconds. Notably, log-10 scale is used for the y-axis.

It is evident that the solving times of the Exact approach are higher than the Hybrid approach's which are higher than the solving times of the Heuristic approach. However, the difference between the Exact and Hybrid approaches is substantial compared to the difference between the Hybrid and Heuristic approaches. For instance, when the deadline is 600 seconds, the solving time of the Exact approach (2122.6 milliseconds) is more than 41 times higher the solving time of the Hybrid approach (51.6 milliseconds), which is only 1.1 times higher than the solving time of the Heuristic approach (46.6 milliseconds). Furthermore, when the deadline is 1500 seconds, the solving time of the Exact approach (197.6 milliseconds) is more than 19 times higher the solving time of the Hybrid approach (10.2 milliseconds), which is only 2 times higher than the solving time of the Heuristic approach (2.04 milliseconds).

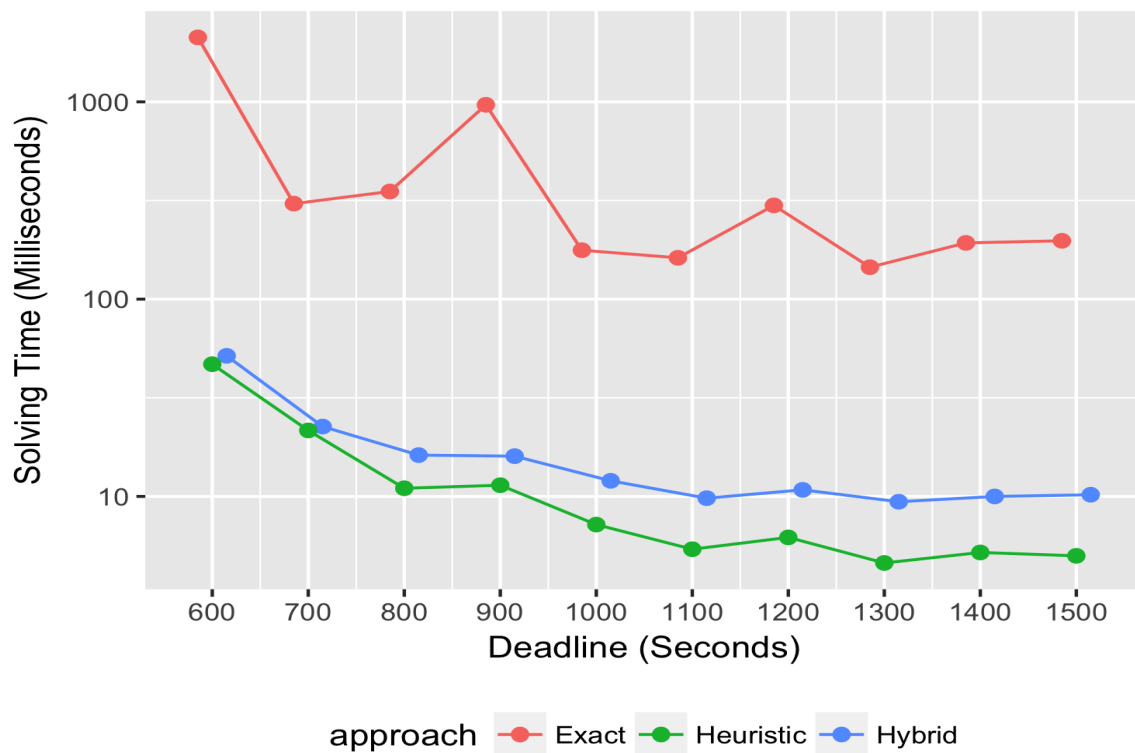
Figure 8.4a also shows that the solving time decreases as the deadline increases. In average, the solving times of the Exact, Hybrid, and Heuristic approaches decreases 90.7%, 80.2%, and 89% respectively. This can be explained as follow: since the deadline increases, it is possible to assign more tasks to be executed on each instance. This results in the lower number of instances required to execute all tasks. As a result, the scheduling approaches need to consider the lower number of instances, which reduces the solving time.

Figure 8.4b presents the total cost of the scheduling plans produced by the three approaches when the deadline varies. The x-axis shows the deadline while the y-axis shows the total cost. Similar to the solving time, the total cost decreases as the deadline increases with the same reason, i.e. lower number of VMs is required to execute all tasks, which results in the lower total cost.

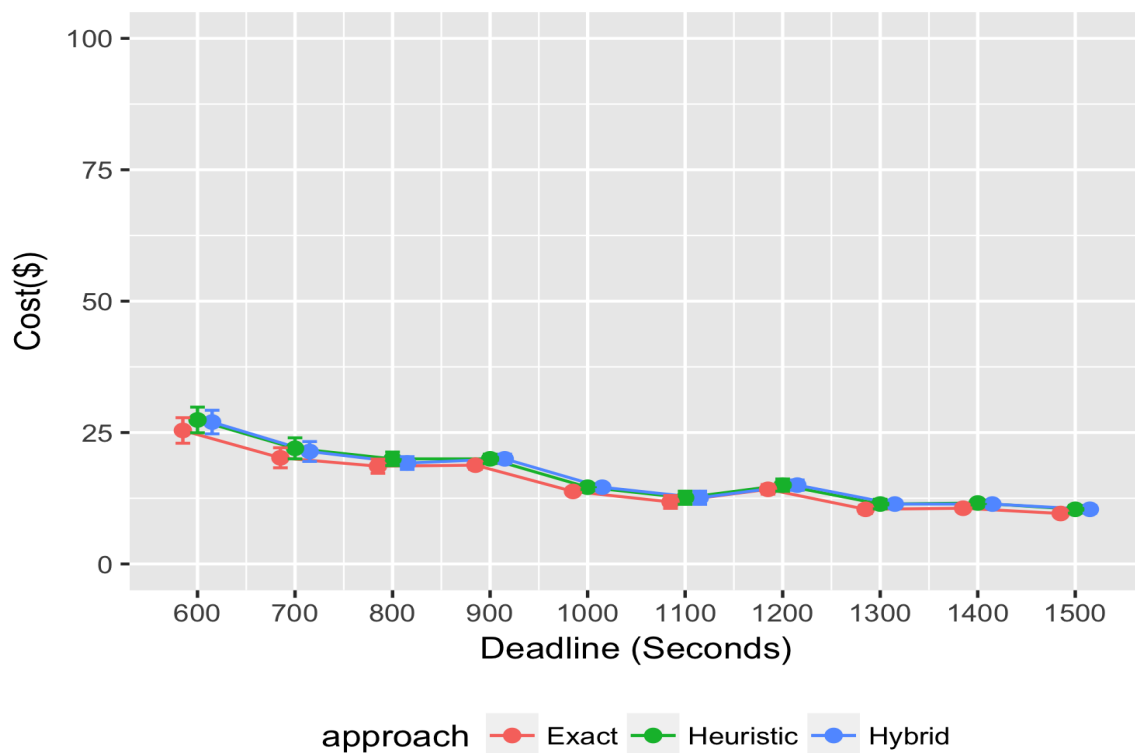
The figure shows that the Exact approach always has the lowest cost in comparison to the other approaches, whose costs are almost identical. In average, the cost of using the Exact approach is 6.6% and 7.6% cheaper compared to the Hybrid and Heuristic approach respectively. Using the Hybrid approach results in the cost which is 1% cheaper compared to the Heuristic approach.

8.2.3 Discussion

In the previous section, we have presented the experiment results comparing three proposed scheduling approaches. Based on the results, this section presents the remarks and conclusion, some of which are previously hypothesised.



(a) Average solving times of three scheduling approaches when the deadline varies. The x-axis presents the length of deadlines, the y-axis presents the solving time in log scale.



(b) Average cost of three scheduling approaches when the length of deadlines varies. The x-axis presents the length of deadlines, the y-axis presents the total cost of a cloud cluster constructed to execute all the jobs. The error bars illustrate the standard errors.

Fig. 8.4 Experiment Result of Three Scheduling Approaches When the Deadline Varies

The total cost is affected by the intensity of the submission. As shown by Figures 8.1b and 8.2b, intensive workloads, i.e. higher number of tasks/jobs or shorter deadlines, also results in higher costs, since it requires more instances. On the contrary, high deadline, i.e. low urgency, can reduce the number of instances and result in lower cost. as shown by Figure 8.4b

The solving time is affected by the intensity of the submission. As shown by Figures 8.1a and 8.2a, the solving times of all three approaches increase when the number of jobs or tasks increases. Moreover, as shown by Figure 8.4a, the solving time can decrease if the deadline decrease, i.e. the urgency decreases.

The number of instance types does not affect the solving time and cost. As shown in Figure 8.3, there is no correlation between the number of instance types and the solving time. This can be explained as all approaches are able to quickly focus on the certain instance types which discarding the rest. Similarly, the total cost is not affected by the number of instance types.

The Exact approach is the most cost efficient. In all of our experiment, the Exact approach is able to achieve the lower cost than the other two. It can reduce cost as much as 6.6%.

The Exact approach is not scalable. As shown by Figures 8.1a and 8.2a, the solving time of the Exact approach grow significantly. It can take few minutes to find a solution. This is not ideal for a real time system which should schedule job execution as soon as possible.

Both the Hybrid and Heuristic approaches are scalable. In our experiments, both approaches are able to produce the scheduling plan within 100 milliseconds at any type of workload intensity. In other words, they both scale well and are suitable for a real time system which requires quick scheduling decision making.

The Hybrid approach is the best of both world. Even though the Hybrid approach is able to find a cheaper scheduling plan compared with the Heuristic approach, its cost saving is not significant since it always around 1%. However, taking into account that the solving time of the hybrid approach is still very fast (less than a second), there is no disadvantage of using this approach.

As a conclusion, we believe the both the Hybrid and Heuristic approach are suitable for a real time system which not only needs to handle intensive workload but also is required to make quick scheduling decision. On the other hand, even though the Exact approach can achieve cost saving up to 6.6%, its solving time is too high to be suitable highly intensive system.

8.3 Evaluating the Unknown Handling Mechanism

In Section 6.2, we presented a mechanism to estimate the task execution time of an unknown application on all available instance types by performing a sampling phase. This section presents thorough experiments in order to test the effectiveness of the Unknown Handling mechanism. More specifically, we are going to evaluate whether the proposed approach can achieve cost saving in comparison with others which do not rely on task execution time for scheduling job execution.

8.3.1 Environment Set-up

In order to shorten the length of each experiment run, we define each ATU to be 10 minutes (600 seconds). The overhead to create new instance, i.e. β , is set to 15 seconds. We also set the length of a sampling phase to 10% of a job's available execution time.

Simulation Framework

The experiment in this section is performed on the simulated environment. In other words, instead of interacting with the cloud, we have developed a simulation framework which supports creating offline cloud VMs, executing tasks on instances, and monitoring the execution progresses.

The simulation framework is implemented based on the methods defined by the Cloud Manager service presented in Section 7.2.5. The framework creates a simulated cloud environment in a single machine. Each VM is represented by a thread.

The task execution on each VM is simulated using Scala's `Thead.sleep` method which temporarily ceases an execution of a thread, i.e. VM, for a given amount of time. The amount of sleeping time is determined based on the predefined task execution times data. In order to simulate the variation of task execution times, we create a normal distribution in which a task execution time is a mean of the distribution.

For progress monitoring, each thread, i.e. VM, keeps track of its own execution process and has a method which returns the list of actual execution times for all finished tasks of a current workload.

Simulated Instance Types

For the experiment, we simulate three general purpose instance types provided by the AWS cloud. Their hardware specifications and prices are shown in Table 8.2.

Table 8.2 AWS Instance Types

Name	vCPU	ECU	Mem (GiB)	Storage (GB)	Price per Hour (\$)
m3.medium	1	3	3.75	4	0.073
m3.large	2	6.5	7.5	32	0.146
m3.xlarge	4	13	15	80	0.293

Simulated Applications

In the simulated experiment, we use the traces of three real world applications.

The first is a Molecular Dynamics Simulation (MDS) of a 250 particle system in which the trajectory of the particles and the forces they exert are solved using a system of differential equations [37]. MDS is embarrassingly parallel and CPU intensive. Which means that all the computation is performed on CPU. Moreover, there is no communication between processes running on all CPU cores. The second one uses *SVM^{light}*¹ to classify data sets provided as input files ranging from 100MB to 500MB. This application only uses one CPU core on a machine. The third one uses *lbzip2*², a parallel compression utility, to compress files ranging from 500MB to 1GB. This application can run on multiple CPU cores, there is communication between processes running on CPU cores.

Prior to the experiment, a sampling process to generate the average task execution time of all applications on the instance types was performed as shown in Figure 8.5.

According to Figure 8.5a, MDS application enjoys significant performance improvement. More specifically, doubling the number of cores reduces the task execution time by half. This is because MDS is an embarrassingly parallel application, which means the workload can be evenly distributed and executed independently on all cores. Moreover, it can be seen that the task execution times of MDS suffer no variation. We believe that since all the computation are performed in the CPU, whose performance remains stable all the time.

According to Figure 8.5b, the performance of *lbzip2* improves when moving to more expensive and better performing instance type. However, the performance improvement is not as high as MDS's. This is due to the communication overhead between processes which increases as more cores are used. Furthermore, the task execution times of *lbzip2* have high variation since it requires IO operations, i.e. reading and writing files, which have been reported to suffer unstable performance [40].

According to Figure 8.5c, *SVM^{light}* experience little improvement in performance between different instance type. This is not surprising since the application requires only one core, which mean the number of cores does not affect the overall performance. However,

¹<http://svmlight.joachims.org/>

²<http://lbzip2.org/>

Table 8.3 Job Specification

Job	Application	Submission Time	Number of Tasks	Deadline
j_1	<i>MDS</i>	0	50	600
j_2	<i>SVM^{light}</i>	300	50	900
j_3	<i>lbzip2</i>	600	50	1200
j_4	<i>MDS</i>	900	100	1500
j_5	<i>SVM^{light}</i>	1200	100	1800
j_6	<i>lbzip2</i>	1500	100	2100

there is still a performance improvement between m3.large and m3.xlarge compared to m3.medium. We believe this is because these two instance types are equipped with better CPU than m3.medium, i.e. each single core is better performing. Finally, since *SVM^{light}* has to read data files from local drive, its task execution times varied, although not as much as *lbzip2*.

Job Submission Pattern

Table 8.3 presents the job submission pattern used in our experiment. Overall, there are six jobs with two jobs for each application. Each job is submitted 300 second apart from each other. The first three jobs have 50 tasks each while the last three have 100 tasks each. All jobs have the same deadline, which is 600 seconds from their submission.

Evaluating Approaches

In order to evaluate our proposed approach, which schedules unknown BoT jobs with deadlines on the Cloud and is denoted as *unknown*, we also perform experiment using other settings. The first one is the ideal setting which has full knowledge regarding executing times of applications on all instance types and is denoted as *known*.

Our approach is also compared with other approaches which do not require task execution time. More specifically, they use the fixed amount of resources at all time and apply the round-robin method to assign tasks to any available idle VMs. We use totally six different configurations which are selected since their costs per ATU are quite similar:

- *medium.8*: 8 instances of m3.medium
- *medium.10*: 10 instances of m3.medium
- *large.4*: 4 instance of m3.large
- *large.5*: 5 instances of m3.large

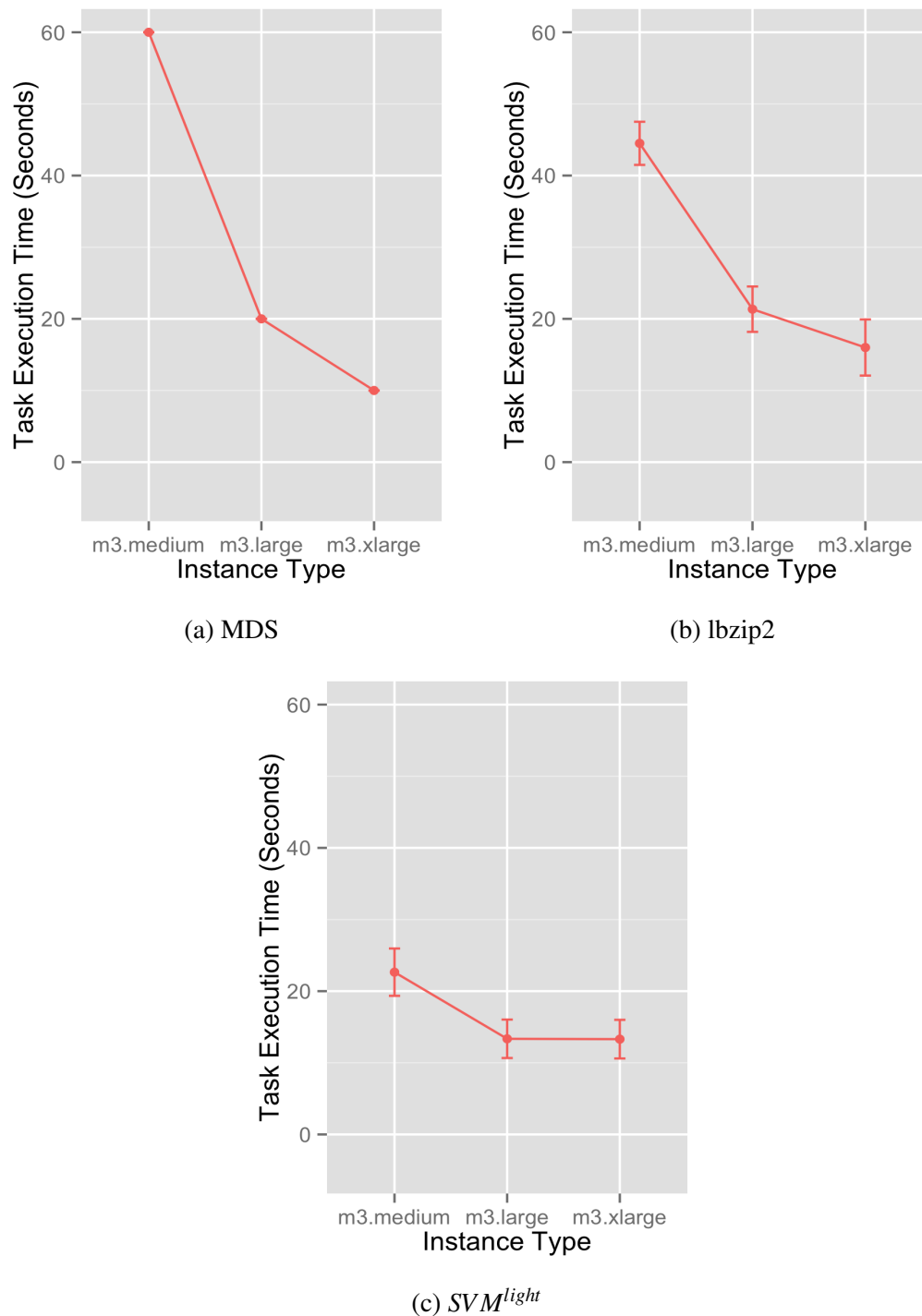


Fig. 8.5 The Task Execution Times of the Applications Retrieved by Running Sampling Execution. The error bars illustrate the standard errors.

- *xlarge.2*: 2 instances of m3.xlarge

- *xlarge.3*: 3 instances of m3.xlarge

8.3.2 Experiment Results

In order to evaluate the unknown handling mechanism and compare it with others settings, we use three metrics: i) the total cost, ii) the number of tasks that finish after deadline, and iii) the total amount of time by which deadlines are exceeded. The results are presented by Table 8.4 and illustrated by Figure 8.6.

In term of monetary cost, Figure 8.6a shows that in comparison to the ideal *known* setting when the tasks execution times are already known, the monetary cost of the *unknown* approach is 10% higher. This is due to the fact that VMs of all types need to be created in order to perform the sampling phases. Moreover, the monetary cost of the *unknown* is comparable to those of the *medium.8*, *medium.10*, and *large.5* settings. Interestingly, the *large.4* setting has the lowest cost compared to others setting, including the ideal *known* one. Finally, the *xlarge.2* and *xlarge.3* settings have higher cost than any other settings.

In term of violation, it is evident that the our unknown handling approach is able to reduce the violation as low as the ideal *known* setting, as illustrated by Figures 8.6b and 8.6c. More specifically, in the *unknown* setting, by average there are 2.8 late tasks in comparison to 2.4 late tasks when the task execution times are known priori. However, the average late time of the *unknown* setting is 34.6 seconds and is even lower than 46 seconds, which is the average late time of the *known* setting. We believe that is due to the fact that the total amount resources of the *unknown* setting is higher than the *known*'s, since it is more expensive.

Compared to other approaches in which resources are fixed, the unknown handling mechanism is able to reduce the number of late tasks from 100% to more than 12785%. Similarly, by perform the sampling phase to retrieve tasks execution times, which are then used to perform scheduling, the amount of exceed time is reduced from nearly 150% to almost 111430%.

As mentioned earlier, using 4 instances of type m3.large results in the lowest cost, even lower than the *known* setting, which has full knowledge regarding task execution times. However, this setting also has a high degree of violation. On an average, the number of late tasks is 27 times higher than the proposed *unknown* approaches. Its late time is 66 times higher as well.

Using either 2 or 3 m3.xlarge instances performs poorly in comparison with other approaches. They not only are the most expensive options but also have the highest numbers of late tasks. This can be explained using Figure 8.5 which presents the task execution times of all applications on all instance types. It can be seen that in comparison to a m3.medium (or m3.large) instance, a m3.xlarge one does not have any significant speed-

up. Moreover, a SVM^{light} application has nearly identical performance on both of them. On the other hand, because m3.xlarge is two and four times more expensive compared to m3.large and m3.medium, respectively, with the same amount of money, the number of m3.xlarge instances is always less than the number of instances of other types, which results in lower execution parallelism, since each instance can only execute one task at a time. In summary, using instances of m3.xlarge has insignificant performance speed-up per instance but significantly reduces the number of instances to execute tasks in parallel. As a result, its overall performance is lower in comparison to other settings.

Between all fixed resources settings, *large.5* is the best one overall. Its cost is almost identical to the cost of the unknown handling approach. It also manages to achieve the least violation in comparison to other fixed resource setting as well. However, its number of late tasks and amount of exceed time are still more than twice of the *unknown* setting's.

Table 8.4 Experiment Results

Approach	Mean Cost	Mean Number of Late Tasks	Mean Amount of Late Time
known	2.62	2.4	46.0
unknown	2.9	2.8	34.6
medium.8	2.91	126.0	6174.2
medium.10	2.92	19.4	865.6
large.4	2.42	76.6	2292.0
large.5	2.92	5.6	85.8
xlarge.2	3.51	360.8	38812.4
xlarge.3	4.04	203.4	5582.6

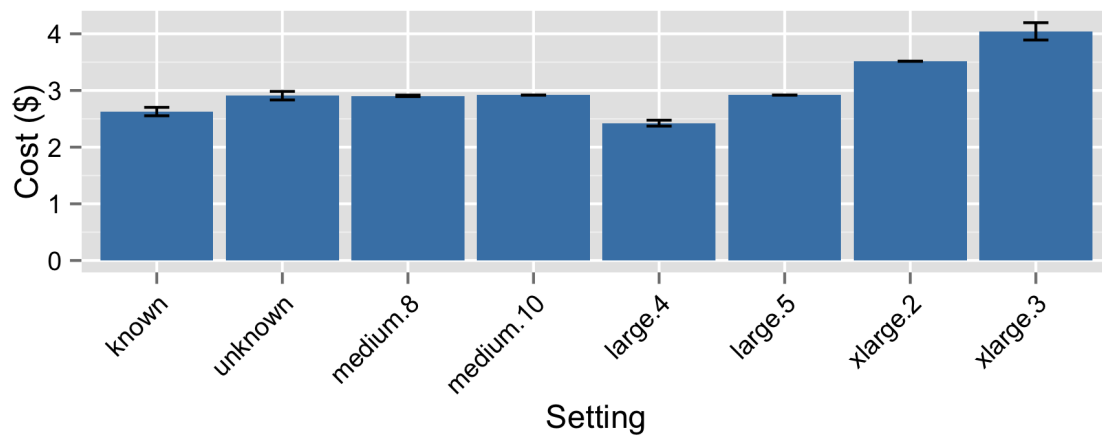
Violation Cost

In order to provide a better comparison between all approaches, we introduce the **violation cost** metric which represents the violation of a execution in term of monetary cost. First of all, the cost of each non-violated tasks can be calculated by dividing the number of non-violated tasks to the total cost:

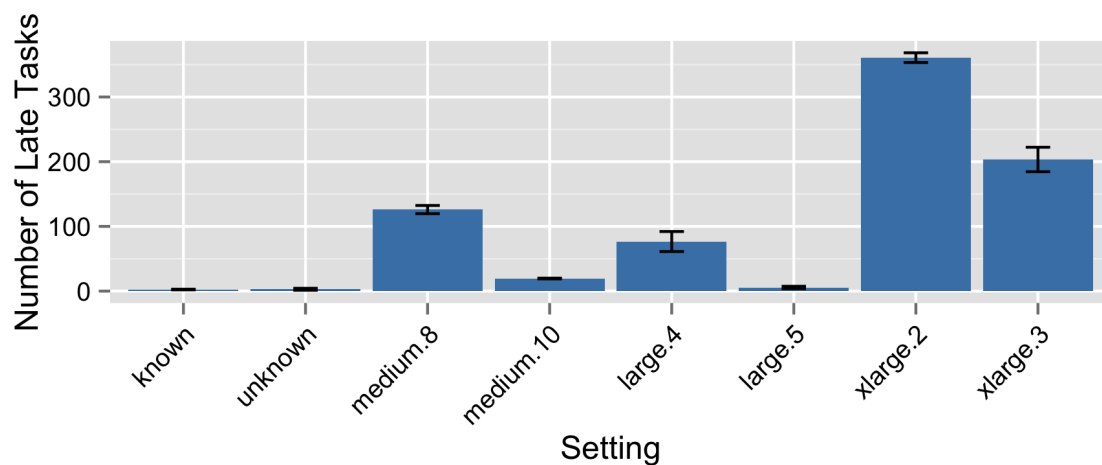
$$cost_per_tasks = \frac{total_cost}{number_of_non_violating_tasks} \quad (8.3)$$

Then, the violation cost is calculated as:

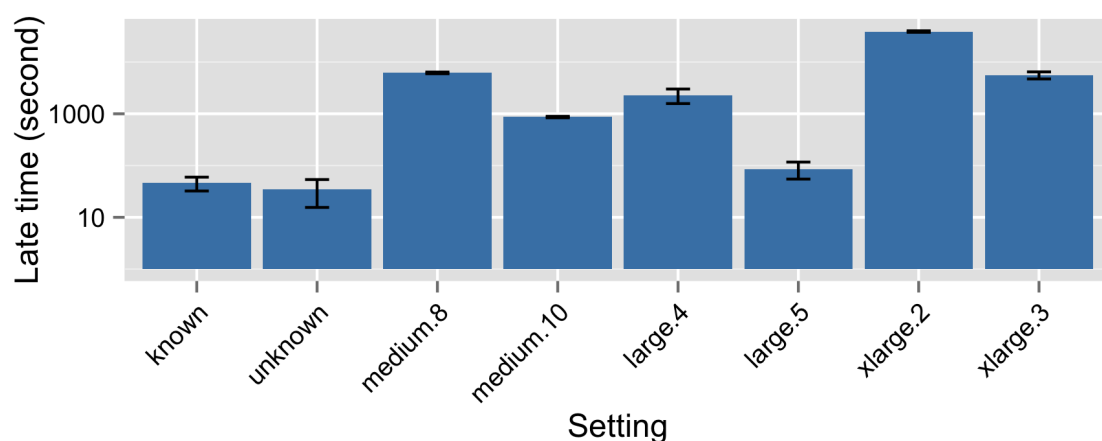
$$violation_cost = cost_per_tasks \times number_of_late_tasks \quad (8.4)$$



(a) Total costs



(b) Number of late tasks



(c) Amount of Late Time in Seconds

Fig. 8.6 The total cost and violation of different settings: in the **known** setting, the task execution times are already available. In the **unknown** setting, the task execution times are not available but going to be estimated during an execution. The **medium.8**, **medium.10**, **large.4**, **large.5**, **xlarge.2**, and **xlarge.3** settings have a fixed number of VMs of a given instance type to execute tasks, thus do not rely on the knowledge regarding the task execution times. The error bars illustrate the standard errors.

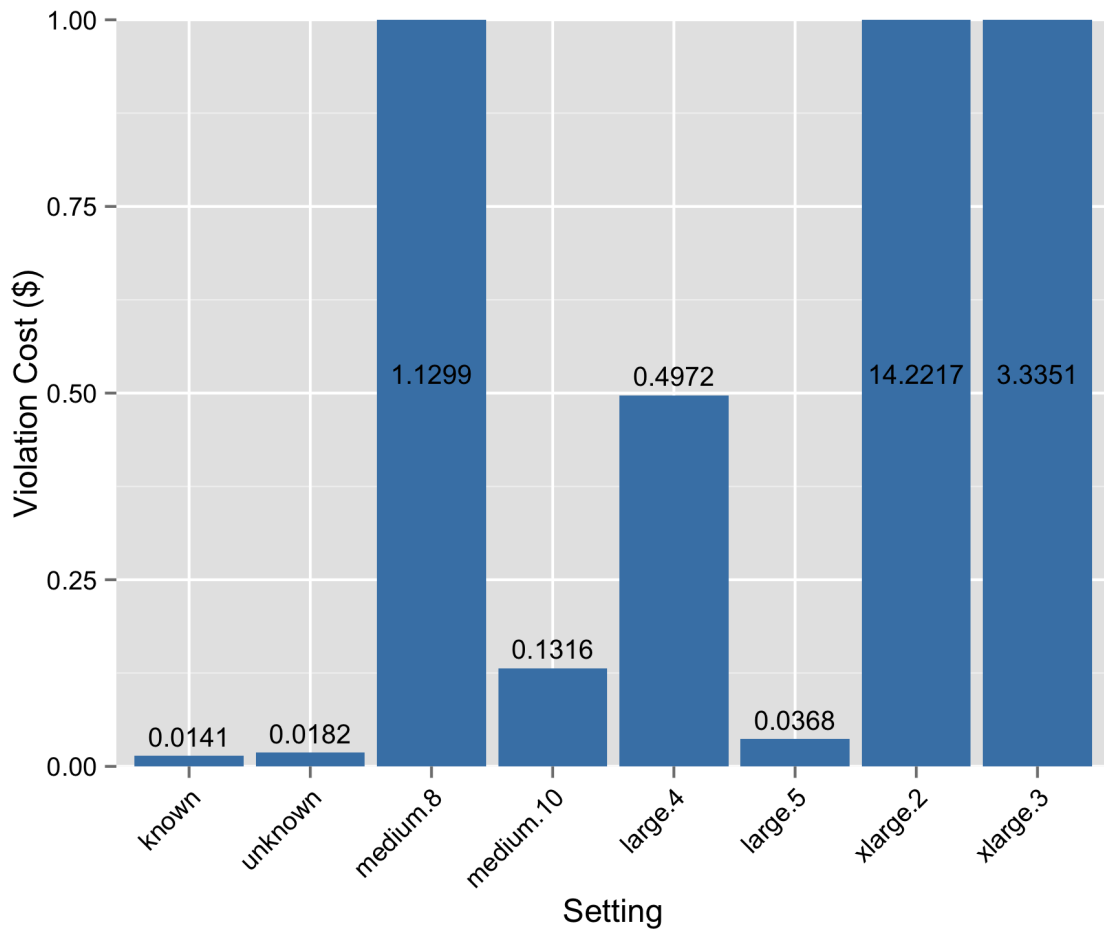


Fig. 8.7 The violation costs of different settings: in the **known** setting, the task execution times are already available. In the **unknown** setting, the task execution times are not available but going to be estimated during an execution. The **medium.8**, **medium.10**, **large.4**, **large.5**, **xlarge.2**, and **xlarge.3** settings have a fixed number of VMs of a given instance type to execute tasks, thus do not rely on the knowledge regarding the task execution times.

The violation cost of an approach can also be described as *the additional cost required to finish all tasks within their deadline*. Hence, it is desirable to achieve as low violation cost as possible.

As shown by Figure 8.7, the violation costs of the *unknown* approach is 29% higher than the *known* one. This is understandable since our approach not only costs more but also has the higher number of late tasks. On the other hand, the proposed approach still outperforms the rest, whose violation costs are from 2 (e.g. *large.5*) to more than 700 times higher (e.g. *xlarge.3*).

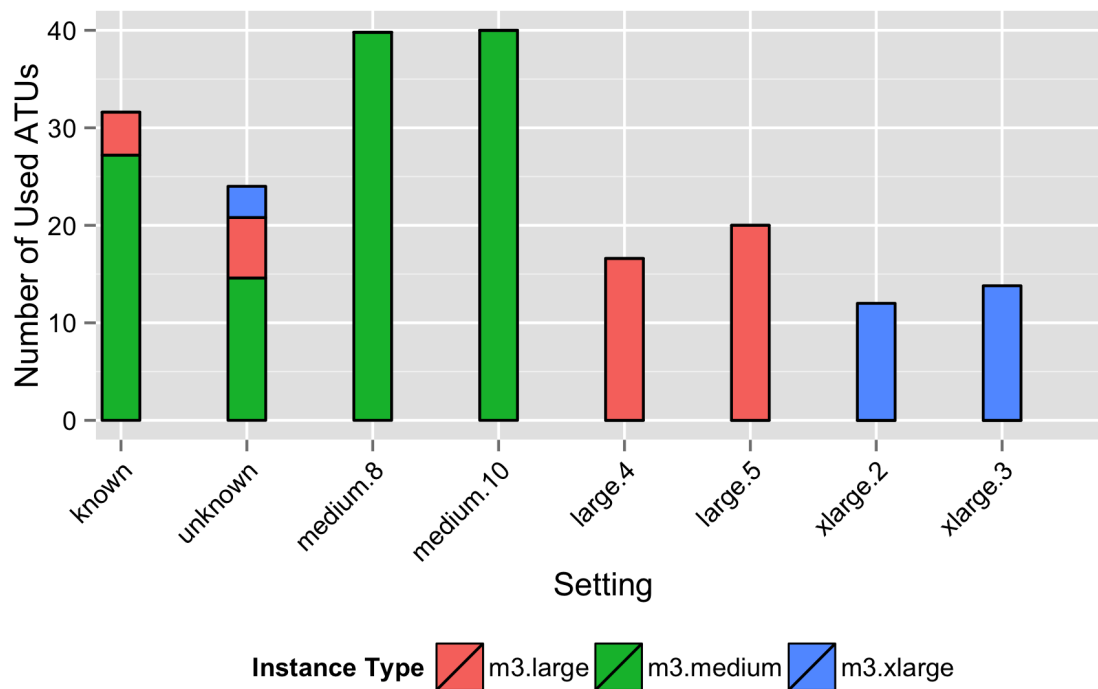


Fig. 8.8 The number of used ATUs per instance Type of each setting: in the **known** setting, the task execution times are already available. In the **unknown** setting, the task execution times are not available but going to be estimated during an execution. The **medium.8**, **medium.10**, **large.4**, **large.5**, **xlarge.2**, and **xlarge.3** settings have a fixed number of VMs of a given instance type to execute tasks, thus do not rely on the knowledge regarding the task execution times.

Numbers of ATU

In order to give a better understanding why the *unknown*, and *known*, approaches outperform other ones which use the fixed amount of VMs of the same type, Figure 8.8 illustrates the overall number of used ATUs corresponding to each instance types for each approaches.

It can be seen that, apart from the settings which use the same instance types, the *known* and *unknown* approaches use the combination of different ones. In other words, those two approaches are able to build a heterogeneous cluster of cloud VMs in order to execute given workload while ensuring the quality of service, i.e. deadline, and minimising the cost.

Notably, the *known* approach does not use any instance of type m3.xlarge which is expensive without any significant speed-up, as explained earlier. On the other hand, since the sampling phase requires tasks to be executed on instances of all types, the *unknown* approach has to create some instances of m3.xlarge. Due to the cost-inefficiency of this type which

results in lower execution parallelism without significant performance improvement, our proposed approach is outperformed by the ideal one.

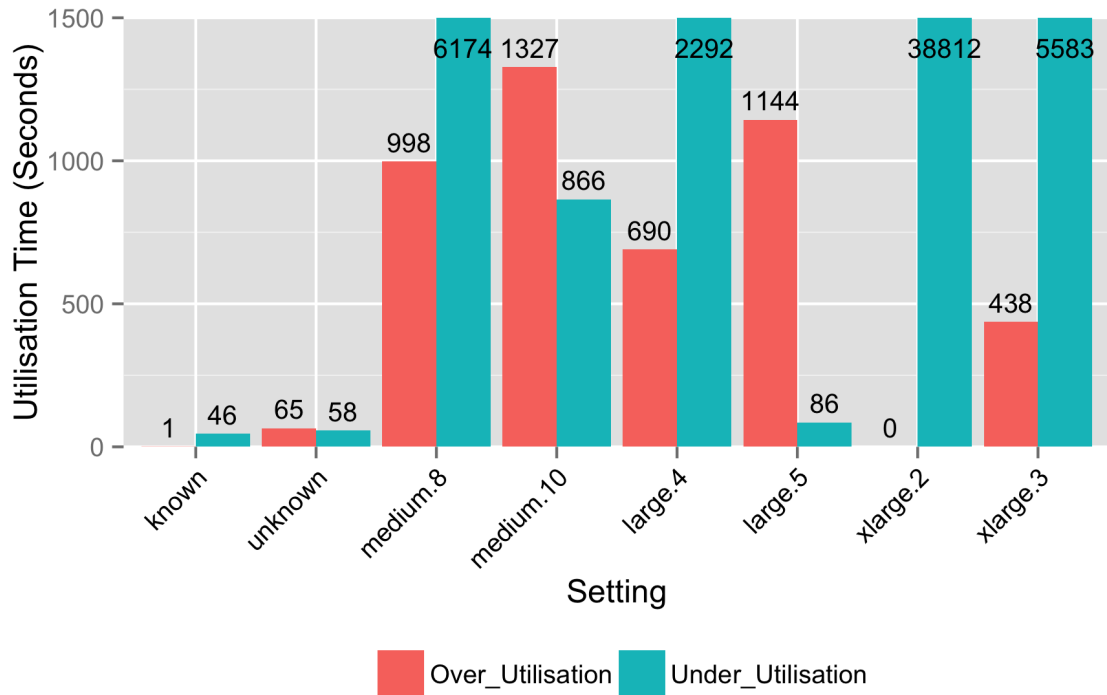


Fig. 8.9 Resource Utilisation of each setting: in the **known** setting, the task execution times are already available. In the **unknown** setting, the task execution times are not available but going to be estimated during an execution. The **medium.8**, **medium.10**, **large.4**, **large.5**, **xlarge.2**, and **xlarge.3** settings have a fixed number of VMs of a given instance type to execute tasks, thus do not rely on the knowledge regarding the task execution times.

Resource Utilisation

Finally, we evaluate the **resource utilisation** of each approach. There are two types of mis-utilisation: **resource over-utilisation** is when the amount of allocated resources is more than necessary and results in idle VMs. The over-utilisation is calculated as the total idle times of all instances. Notably, jobs are submitted in the way so that their execution overlap with each other. Hence, ideally, there should not be any idle instances.

On the other hand, **resource under-utilisation** happens when the amount of resources is not enough to execute all tasks within their deadlines. As a result, resource under-utilisation is calculated as the total violation time.

Figure 8.9 illustrates the average resource utilisation of each approach. It can be seen that both the *known* and *unknown* manage to achieve very low resource over- and under-utilisation in comparison with other ones.

On the other hand, most of the fixed resources setting have high under-utilisation due to the fact that they have enough resources to finish some jobs very early before the submission of the next one. However, occasionally, they do not have enough resources to execute some other jobs within deadlines, hence the over-utilisation is also very high.

Finally, the *xlarge.2* setting has no idle time but very high late time. In other words, the allocated resources of this setting is always less than required.

In summary, the *known* and *unknown* approaches are able to not only achieve low cost but also minimise violation by flexibly adjust the amount of allocated resources based on current workload.

8.3.3 Discussion and Summary

In this section, we have presented and discussed the experiment results evaluating the unknown handling approach. It is shown the the proposed approach, which performs a sampling phase in order to retrieve required knowledge, is able to deliver the scheduling decision within 10% cost and 16% violation compared to the ideal setting which has full knowledge from the beginning. It also outperforms other approaches that use a fixed amount of resources by reducing the monetary cost of violation by at least two times.

8.4 Dynamic Reassignment

This section presents thorough experiments in order to evaluate the benefits of the dynamic reassignment feature.

8.4.1 Experiment Set-up

In this section, we use the same simulation set-up as mentioned in Section 8.3. Which means that the instance types, applications, task execution times, and job submission pattern used in this experiment is similar to those used in evaluating the unknown handling mechanism. On the other hand, since the focus of the experiments performed is the dynamic reassignment feature, i.e. to show the difference in execution when this feature is turned on and off, we will not use any naive approaches. In other words, the experiments are performed on three proposed scheduling approaches, i.e. Exact, Hybrid, and Heuristic approaches.

Moreover, in order to investigate the affect of performance variation, it is simulated using **coefficient of variation**, which is the ratio between standard variation and mean. As mentioned in the previous section, the actual task execution time is generated using its mean and standard deviation time. My manipulating the standard deviation, we can manipulate the performance variation. Given a coefficient of variation CV and a mean task execution time e , the standard deviation σ can be calculated as: $\sigma = CV \times e$.

In our experiments, we use 5 different values of CV : 0, 0.25, 0.5, 0.75, and 1. When CV is equal to 0, then $\sigma = 0$, which means there is no performance variation. On the other hand, when $CV = 1$, then $\sigma = e$, which means the actual execution time can reach as much as twice the mean value.

Notably, in the cloud einvironment, the performance variation is random instead of following a statistical model. However, since the scheduling framework is not aware of the statistical model used to model the performance variation, we believe that this does not greatly affect the outcome of the experiment. Moreover, we decided not to consider sudden termination of VMs due to the rarity of such event. For instance, AWS, Google Cloud Platform, and Microsoft Azure guarantee the availability of their cloud systems to be more than 99%.

8.4.2 Experiment Results and Discussion

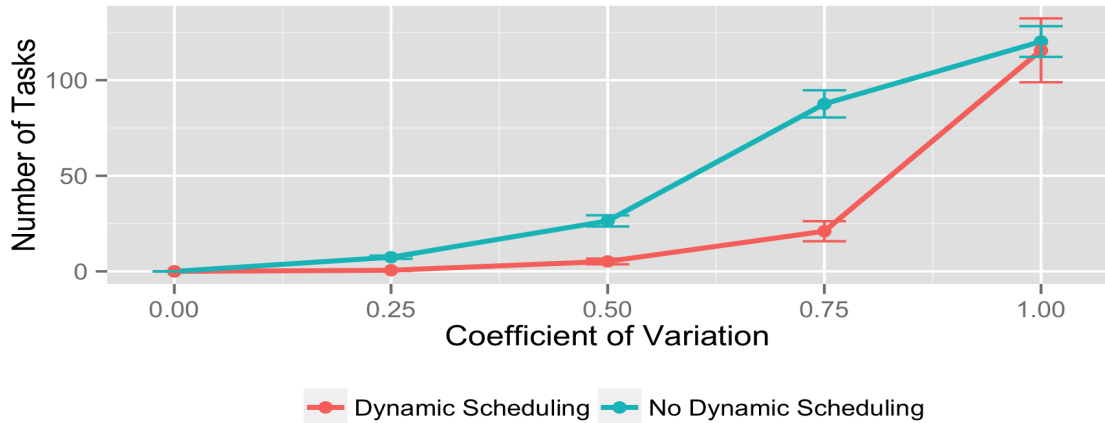
Each experiment is performed 5 times. We use the number of tasks finished after deadlines and the amount of exceeded time in order to evaluate the dynamic reassignment feature. Figures 8.10 and 8.11 illustrate the experiment results.

First of all, it is evident that increasing the performance variation, i.e. coefficient of variation, affects the violation of the execution. When there is no performance variation, i.e. $CV = 0$, deadline violation does not occur at all. As the performance variation increases, both the number of violated tasks and exceed time increase as well.

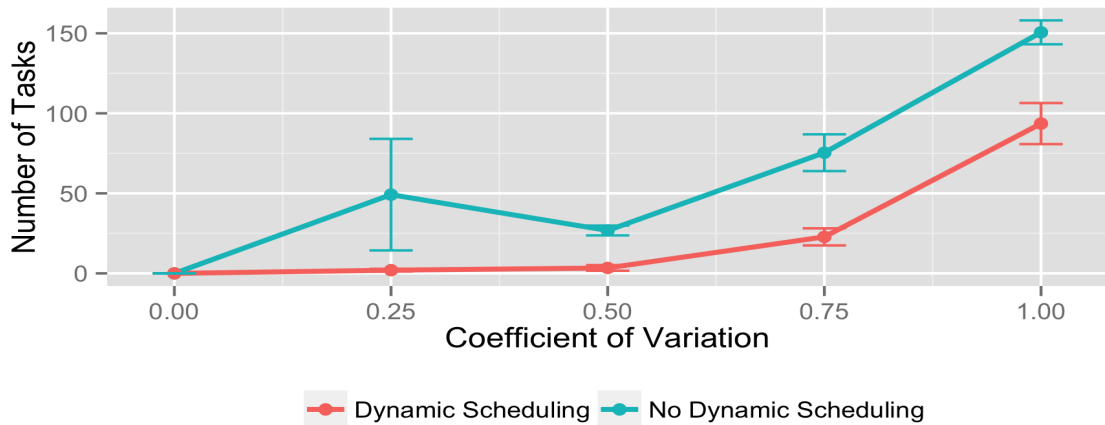
Secondly, using the dynamic reassignment feature lessens the effect of performance variation and reduces the deadline violation. As illustrated by Figures 8.10 and 8.11, when performance variation occurs (i.e. $CV > 0$), using dynamic reassignment always results in lower number of violated tasks and exceeded time. More specifically, the number of violated tasks was reduced from 3.8% up to 99.5%. Similarly, the amount of violation time was reduced from 17% to 93.7%.

Interestingly, using the dynamic reassignment feature also results in lower costs as illustrated by Figure 8.12. The total cost while the dynamic reassignment feature is on is always less than or equal to the cost when it is off. We believe this is caused as some VMs are behind schedule, they are not available to receive extra tasks from a newly submitted job.

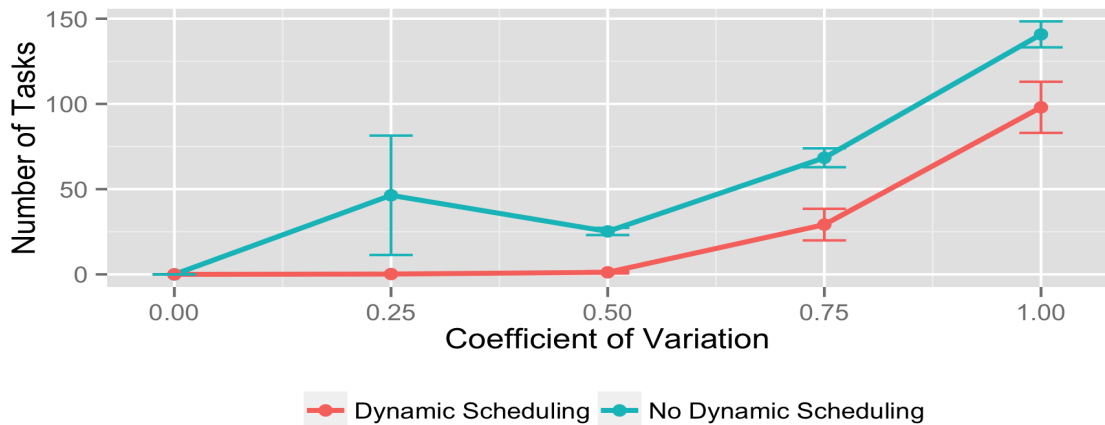
As a result, more VMs are required to handle new workload. Which means the total cost is higher as well.



(a) Average Number of Violated Tasks of the Exact Approach

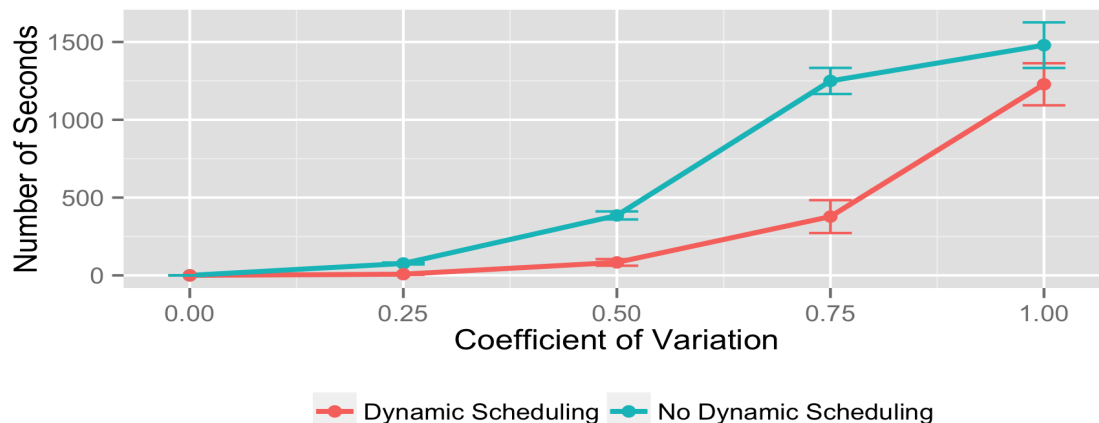


(b) Average Number of Violated Tasks of the Hybrid Approach

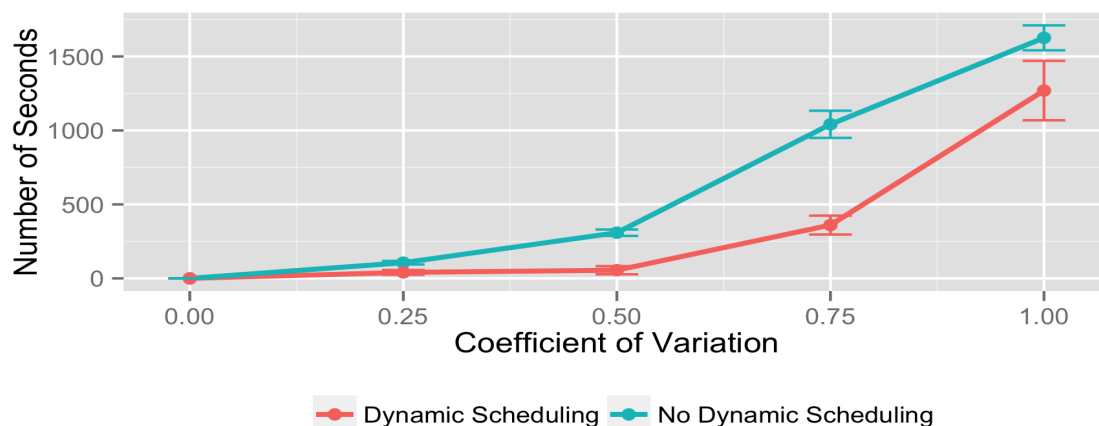


(c) Average Number of Violated Tasks of the Heuristic Approach

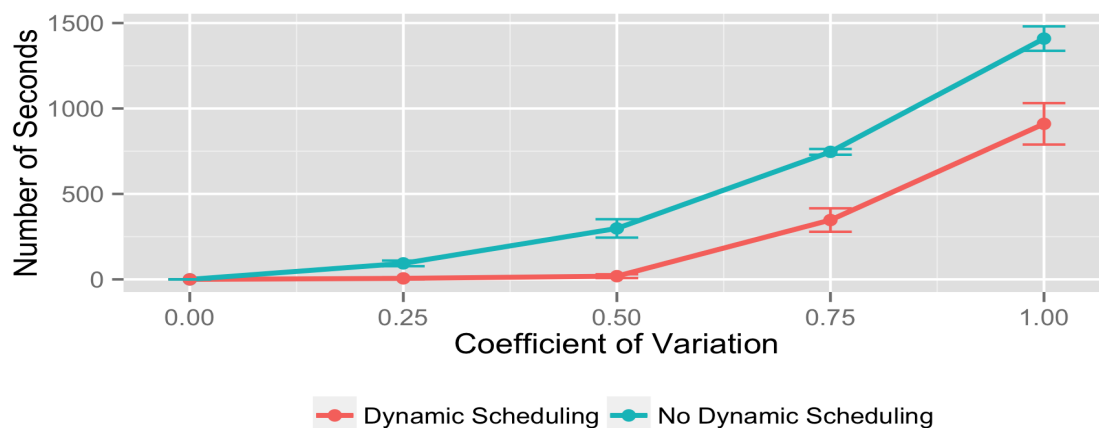
Fig. 8.10 Average Number of Violated Tasks for Each Approach When Dynamic Scheduling Is Turned On/Off. The error bars illustrate the standard errors.



(a) Average Amount of Violated Time of the Exact Approach

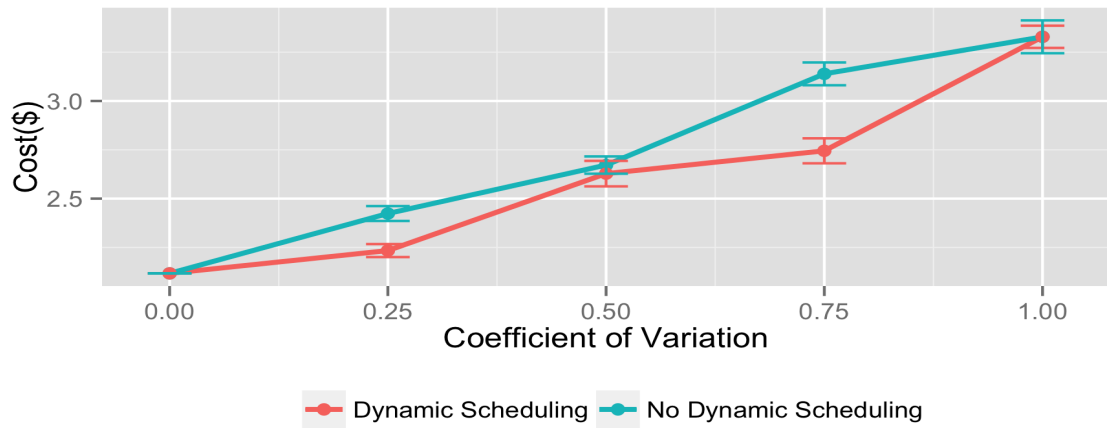


(b) Average Amount of Violated Time of the Hybrid Approach

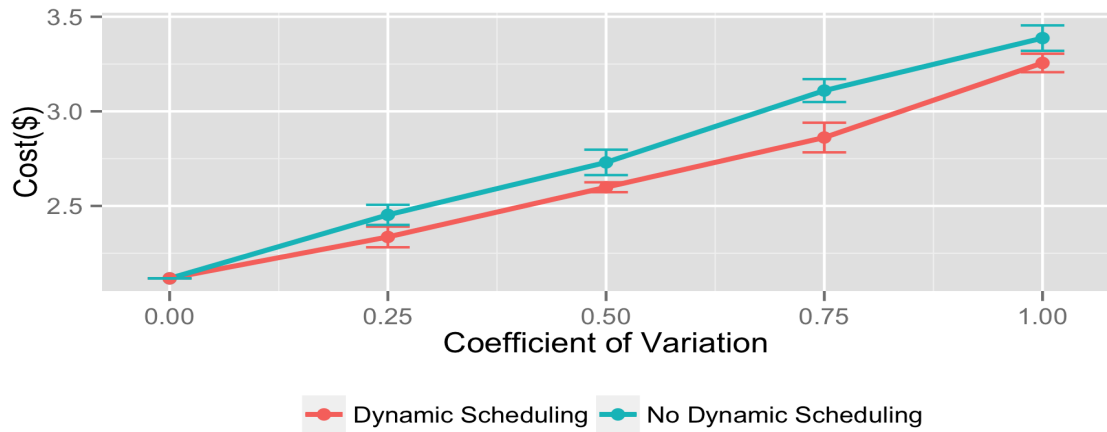


(c) Average Amount of Violated Time of the Heuristic Approach

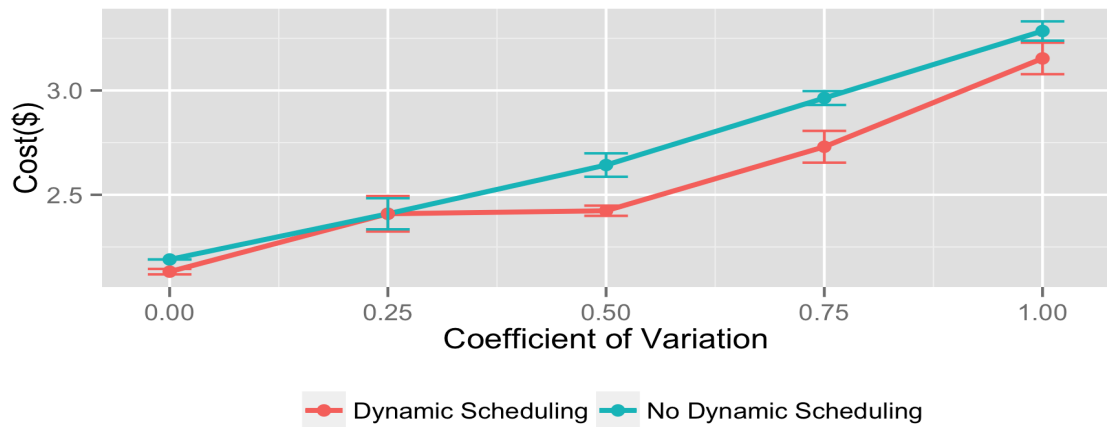
Fig. 8.11 Average Amount of Violated Time for Each Approach When Dynamic Scheduling Is Turned On/Off. The error bars illustrate the standard errors.



(a) Average Cost of the Exact Approach



(b) Average Cost of the Hybrid Approach



(c) Average Cost of the Heuristic Approach

Fig. 8.12 Average Cost of Each Approach When Dynamic Scheduling Is Turned On/Off. The error bars illustrate the standard errors.

8.5 Cloud Experiments

In the previous sections, we have evaluated the proposed research in an simulated environment. This section presents the set of experiments which were performed in real world in order to provide the proof of concept that our research can be applied to real world cloud.

8.5.1 Environment Set-up

The instance types, applications, and task execution times used in the experiment are similar to the simulation experiment. On the other hand, the job submission pattern is changed as follow: the jobs are submitted in two batch. The first one is submitted when an experiment started and consists of three jobs, corresponding to each application, with 100 tasks each. Their deadlines are 1200 seconds. The second batch is submitted 300 seconds later. It also contains 3 jobs, but each of them has 150 tasks. The deadlines are also 1200 seconds. Notably, all tasks execution times are known.

Experiment Framework

In order to interact with AWS cloud, we use the library **AWScala**³. The library allows us to create and terminate AWS VMs programmatically.

In order to perform and monitor execution within each VM, we develop a RESTful python web service using **cherrypy**⁴. The web service is pre-deployed inside a Amazon Machine Image (AMI). In other words, when a new VM is created based on the AMI, the python service is already running. Similarly, all the applications used in the experiment are also pre-deployed in an AMI in term of scripts file. The execution can be performed by executing those scripts. And required data is also stored in the cloud and all VMs can access them directly. The service provides the following functionalities which can be invoked remotely using a HTTP call:

- `StartWorkload`: this functionality takes a name of a workload, its jobs, and tasks in order to execute one of the applications mentioned above by executing the batch scripts.
- `RemoveTasks`: this functionality takes a number of tasks to be removed from the currently executed workload.

³<https://github.com/seratch/AWScala>

⁴<http://cherrypy.org>

- `CheckProgress`: this functionality returns the list of integer values, each of which is a number of second it takes to execute a task.

We use the library `scalaj-http`⁵ which allows us to make HTTP call the the RESTful web service. Finally, `play-json`⁶ is used for parsing the response which is in JSON format.

Evaluated Approaches

Three propose approaches are evaluated in the real world cloud environment: Exact, Hybrid, and Heuristic approaches. For each of time, the dynamic reassignment feature is turned on and off.

The proposed approaches are compared to commonly used approaches that only use one instance type to create a homogeneous cloud VM cluster. Those approaches use task execution time to calculate the number of VMs required to execute all tasks within deadlines. As mentioned earlier, $n_{j,t}$ is the number of tasks of job j that one instance of type t can execute before a deadline. Hence, the number of VM of t required to execute all tasks of j is $\frac{n_j}{n_{j,t}}$. It should be noted that those homogeneous approach can use existing instances to execute tasks before renting new ones.

There is one approach for each available instance type that we use in this experiment. In other words, there are Medium, Large, and XLarge approaches which only use `m3.medium`, `m3.large`, and `m3.xlarge` instance types respectively. The dynamic reassignment feature is also turned on and off.

8.5.2 Experiment Results

Cost Evaluation

Figure 8.13 illustrates the cost of each approach. It also presents the ratio of costs spent on each instance type.

Firstly, it is evident that the Exact approach has the lowest cost compared to other ones. Its cost is 4% lower than other heterogeneous approaches. Compared to the heterogeneous approaches, using the Exact approach can lower the cost from 9% to 46%. The costs of Hybrid and Heuristic approaches are identical and are lower than the cost of other homogeneous approaches. They are able to lower the cost from 4% to 40%. Between the remaining homogeneous approaches, using only `m3.large` instance results in the lowest cost while the cluster with only `m3.xlarge` VMs has the highest monetary cost.

⁵<https://github.com/scalaj/scalaj-http>

⁶<https://www.playframework.com/documentation/2.5.x/ScalaJson>

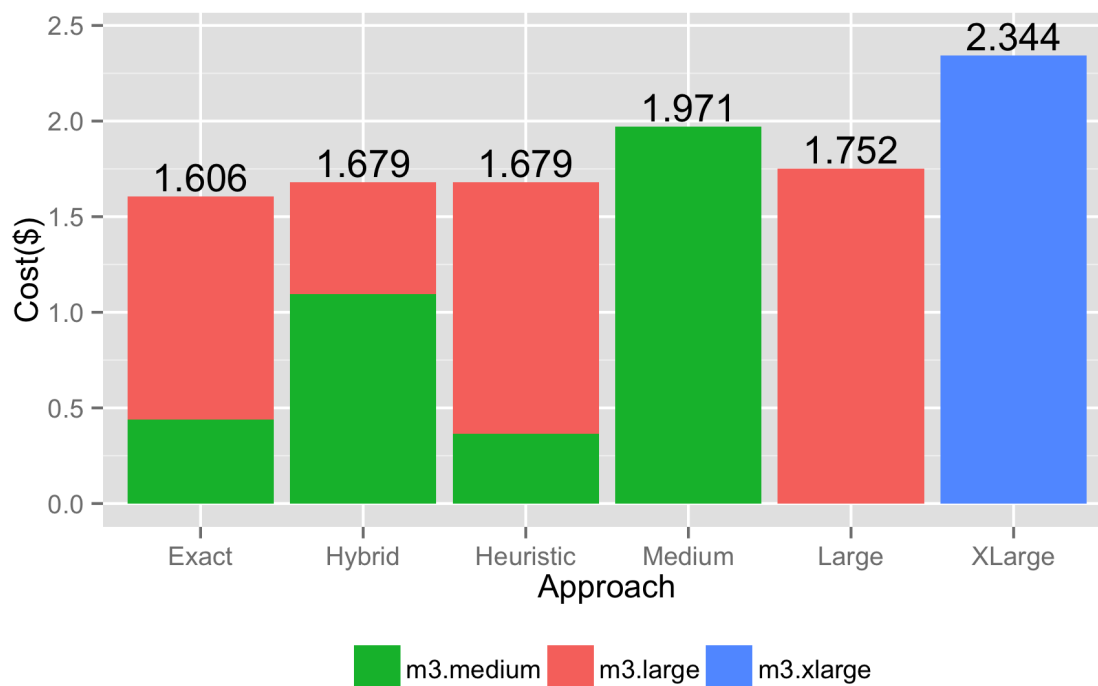
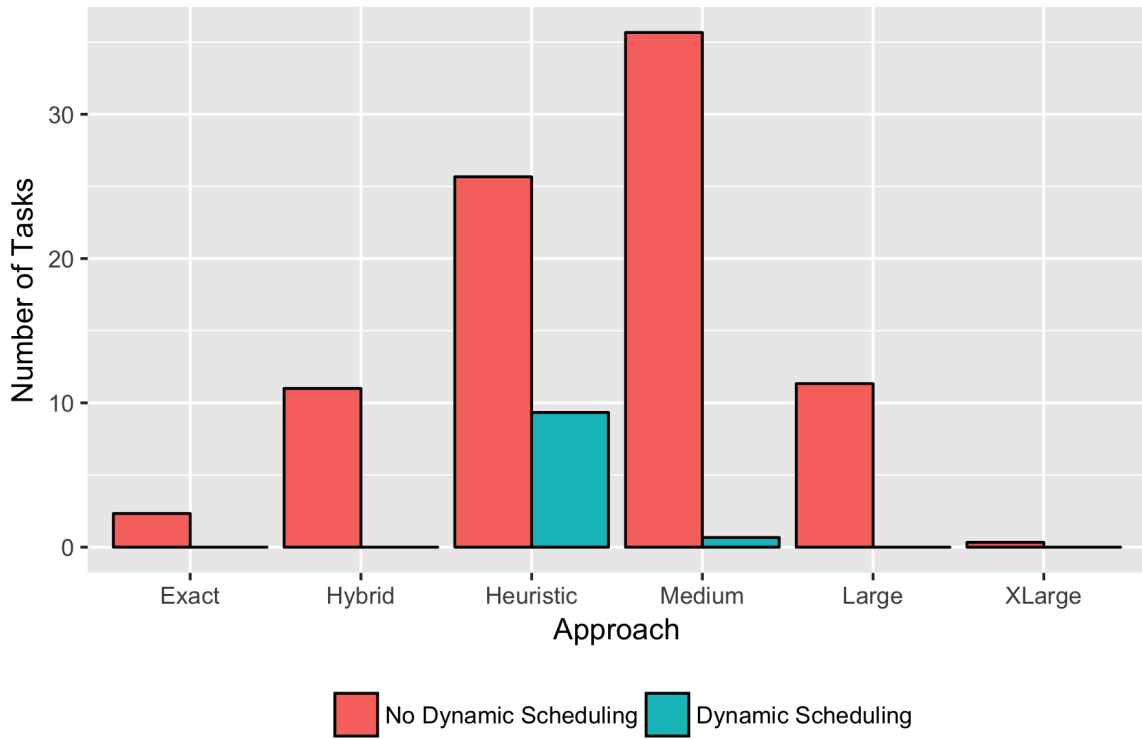


Fig. 8.13 Total costs of each approach. The first three bars represent the total cost of the exact, hybrid, and heuristic approaches which aim to build a heterogeneous cloud clusters. The last three plots represent the total costs of the homogeneous cloud clusters which consist of VMs of only one instance type.

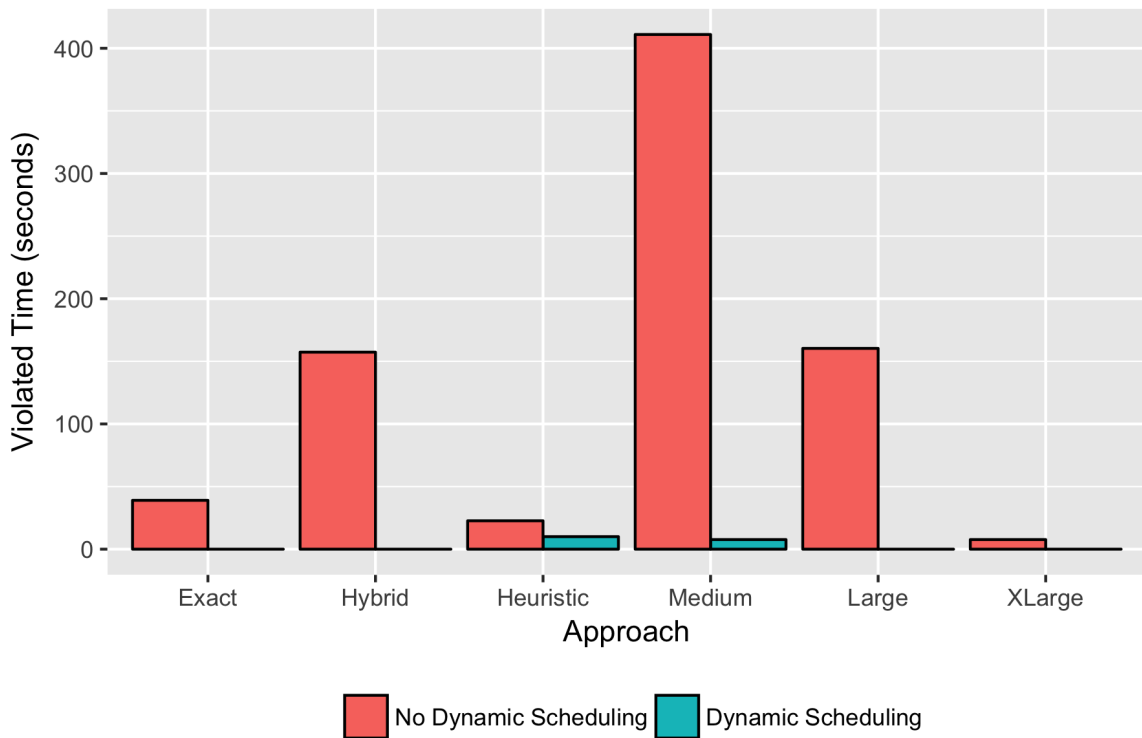
Secondly, all the heterogeneous approaches do not employ instances of m3.xlarge type. This can be explained since this type does not have significant performance improvement compared to the other two types. For instance, even a VM of type m3.xlarge costs twice as much as a VM of type m3.large, their performance corresponding to the application SVM_{light} is almost identical.

Violation Evaluation

Figure 8.14 represents the number of violated tasks and exceeded time for each approach. Once again, it is evident that the dynamic reassignment feature greatly reduces the violation in all approaches. More specifically, there are cases when deadline violation is completely avoided.



(a) Average Number of Violated Tasks of Each Approaches



(b) Average Amount of Violated Time of Each Approaches

Fig. 8.14 Violation of Each Approach. The error bars illustrate the standard errors.

8.5.3 Discussion

In this section, we have performed the experiment on the AWS cloud in order to demonstrate the feasibility of our research in real life scenario. The results have confirm the finding of the simulated experiments presented the the previous sections:

- Constructing heterogeneous cloud cluster using one of the three proposed scheduling approach reduced the monetary cost while satisfying the deadlines.
- Dynamic scheduling mechanism was able to reduce or even completely prevent deadline violation caused by performance variation during runtime.

However, we believe that more experiments with higher degree of complexity are required in order to confidently evaluate the proposed research. In other words, our evaluation is limited in a certain aspects due to financial constraint:

- Each kind of experiment was performed 5 times. If more experiments had been perform, we would be able to confirm our hypotheses with higher confidence.
- The total workload of each experiment was rather small, for example, the difference in cost between two approaches was normally less than \$1. Hence, it was difficult to clearly demonstrate the cost effectiveness of the proposed research.
- Finally, each experiment lasted for less than an hour. Hence, we was not able to evaluate our approaches for long running cluster.

As a conclusion, the experiments presented in this section are just the first step to evaluate the feasibility and applicability of our proposed research in the real world. Further experiment will be left for the future work.

8.6 Chapter Summary

In this chapter, we have evaluated our research in both simulated and real-world environment.

First of all, we compare three scheduling approaches and conclude that the Heuristic approach is the best of both world. It is able to find a solution a lot quicker than the Exact approach. Moreover, the cost of its solution is lower compared to the Heuristic approach.

Our experiments also demonstrate the benefit of the Unknown Handling approach which estimates the task execution times of a new application. With the 10% overhead in monetary cost, the proposed mechanism was able to keep the deadline violation as low as when the task

execution times were fully known prior to the execution. Compared with other configurations which executed jobs without estimating the task execution times, our approach was able to lower the violation rate from 2 to more than 700 times.

The dynamic scheduling mechanism was also evaluated. It was shown that this mechanism was able to reduce deadline violation, with the presence of performance variation, up to 99%. Furthermore, using the dynamic scheduling also lowered the total cost up to 12.6%.

Finally, our proposed research was evaluated in the real world environment. The results showed that it was still able to achieve cheaper cost in comparison to other homogeneous approaches. Once again, the benefit of dynamic scheduling was demonstrated as it greatly reduced and even occasionally completely prevent any deadline violation. However, in order to confidently claim the benefit of the proposed approaches and mechanism, extensive experiment and evaluation are required.

Chapter 9

Conclusion

Scheduling is an important part of the execution of any application in any environment. Its goal is to ensure that the desired performance is guaranteed with the reasonable amount of resources. The role of scheduling has become more and more important in a distributed environment in which multiple applications have to share the same resource pool. This importance is further emphasised, and complicated, when cloud computing is adopted for several reasons. Firstly, the size of a user's resource pool is not fixed. Instead it can grow (by adding more machines) or shrink (by removing machines) almost instantaneously. Secondly, there is the monetary cost that also almost instantaneously added to a user's bill. Without a proper scheduling mechanism, a user may end up with a massive bill with no guarantee that the desired performance is met.

In this thesis, we have set out to address the problem of scheduling multiple applications on the cloud based on a user's requirements, which are described in term of satisfying execution deadlines and minimising the monetary cost. As the result of this journey, we have proposed three different scheduling approaches, each of which has different degrees of optimality and complexity. We have also developed mechanisms to manage the execution at runtime to handle unexpected events and estimate important metrics required for the scheduling process. All the research have materialised into a complete software framework that we also use to perform a thorough evaluation.

9.1 Thesis Summary and Contributions

- Chapter 1 discussed the background and presented motivation of our research. It also introduced the definition and characteristics of cloud computing and BoT applications.

- Chapter 2 surveyed the existing work in scheduling BoT jobs on the cloud. Based on the survey, relevant concepts and themes were identified. Then, they were used to construct a taxonomy reflecting different aspects of the research in workload scheduling. Finally, we presented a set of requirements which were later addressed by the research of this thesis.
- Chapter 3 presented a mathematical model of the problem of scheduling the execution of BoT jobs on the cloud. It also introduced the basic concepts such as task execution time or workload. The result of this chapter was a complete optimisation problem whose solution was a scheduling plan for executing multiple BoT jobs with different deadlines on the cloud so that the monetary cost could be minimised.
- Chapter 4 presented a heuristic algorithm which assign tasks of different BoT jobs to existing running VMs. The objectives were to assign as much workload as possible while preventing not only potential deadline violation but also additional monetary cost. We took the bottom-up approach and built this algorithm upon the smaller ones which were also presented in the chapter.
- Chapter 5 presented different approaches for scheduling an execution of BoT jobs on the cloud. There are three different approaches, namely Exact, Hybrid, and Heuristic approaches. All of them aim to not only select the cloud resources but also allocate workloads to them so that a user's requirements are satisfied. Each of them had a different degree of complexity and solution optimality. One of the goals of this chapter was to present the trade-off between the complexity of a scheduling approaches and their corresponding solution optimality.
- Chapter 6 consisted of two sections each of which addressed the different aspects of managing an execution in runtime.
 - Section 6.1 introduced an algorithm to handle performance variation at runtime. The proposed algorithm was able to detect possible deadline violation which could be caused by performance variation. By reassigning tasks between VMs, the algorithm aimed to effectively reduce the possible deadline violation.
 - Section 6.2 proposed a mechanism to deal with jobs of unknown applications, i.e. when the information required to perform scheduling was not available. By running a sampling phase in which a portion of a job were executed on VMs of all available instance type, we were able to get the actual task execution times and used them to estimate the mean task execution times of an application on all

VM type. The result was then used to schedule an execution of the remainder of a job.

- Chapter 7 contained the design and implementation of the framework which incorporated all the research presented in the previous chapter. Our framework consisted of different loosely coupled services which communicated with each other using the actor model. We also described the supported features which were the results of the interactions between different components.
- Chapter 8 presented the detailed evaluation of the research presented in this thesis. Simulated experiments were performed in order to evaluate static scheduling, dynamic scheduling, and unknown handling mechanism. We also performed a real world experiment on AWS cloud in order to prove the applicability of our research. From the experiment result, we are able to conclude that our research is suitable for scheduling the execution of BoT jobs with deadlines on the cloud.

9.2 Lessons Learned

By employing the scheduling approaches, the cost of executing BoT applications is minimised while maintaining the desired performance. As shown in by the experiments, the heterogeneous cloud clusters constructed by the proposed scheduling approaches were able to satisfy the application deadlines with the lower costs compared to homogeneous cloud clusters.

The Exact approach for static scheduling is not scalable. Out of three proposed scheduling approaches, the Exact approach aims to find the optimal solution in which the monetary cost is minimised. The optimality of this approach was demonstrated by our experiment which showed that its cost was always lower than those of the other approaches. However, its complexity results in the high solving time. As our experiments have shown, the solving time of an Exact approach could be up to minutes while other approaches only needed less than a second. Hence, we can conclude that it is not suitable for a real time system which has high submitted workload and requires decisions to be made as soon as possible.

The Hybrid approach is the best of both worlds. Overall, the cost of the hybrid approach is lower compared to the heuristic approach. Its solving time is only tens of milliseconds higher than the heuristic approach. As a result, we can conclude that the Hybrid approach can achieve both lower cost and fast solving time, which makes it a good candidate for any real time system.

The solving time of all approaches depends on the intensity of the workload. More precisely, the number of jobs and/or the number of tasks per job directly affect the solving time. The deadline also affects the solving time since the shorter deadline requires a higher number of instances.

The number of available instance types does not have a significant effect on the solving time of all three approaches. We believe the main reason for this is that all approaches are able to quickly identify a handful of instance types that are cost-effective, so it is not necessary to consider all instance types.

Task execution time is important for effectively scheduling job execution on the cloud. It is evident that scheduling based on task execution time is more cost effective than the naive approaches which employ fixed amount of resources and round-robin task allocation mechanism. As a result, even when task execution times are not available prior to the execution, employing the unknown application handling mechanism is still beneficial.

Deadline violation can happen due to performance variation. As shown in our experiments, deadline violation is directly correlated with performance variation. In other words, the higher the performance variation is, the more serious deadline violation is.

Deadline violation can be reduced or even completely prevented using dynamic scheduling. Our experiment has shown that with the same set-up, dynamic scheduling is able to greatly reduce the effect of deadline violation. It is also able to reduce the incurred monetary cost as fewer VMs are required in order to handle the extra late tasks.

Our proposed research is applicable in real world experiment. By performing experiment on AWS, we are able to show that our proposed research is able to achieve lower cost in real world setting in comparison to other naive approaches. Moreover, the dynamic scheduling approach can reduce and even completely prevent deadline violation caused by performance variation.

As a conclusion, this thesis has been successful in addressing the research hypotheses presented in Chapter 1.

- **All of the proposed scheduling approach have been able to achieve cost saving while maintaining the desired quality of services.** Compared to other existing approaches, our methods were always be able to construct cloud clusters with not only lower cost but also lower deadline violation
- **By employing the execution management mechanisms, potential violations caused by either performance variation or parameter unavailability can be reduced.** More specifically, the dynamic scheduling mechanism could reduce, or even completely prevent, deadline violation caused by performance variation. On the other hand,

the parameter estimation mechanism was able estimate the task execution times of unknown applications, thus resulted in a lower cost compared to scheduling without task execution times.

9.3 Future Work

Scheduling job executions in a distributed environment, especially cloud computing, in order to achieve a desired performance is a challenging problem. In this thesis, we have set out to address one aspect of that problem. However, our research is far from completed and there are some additional work that can be done.

First and foremost, **more experiments with higher degree of complexity and workload** are required in order to thoroughly evaluate our proposed research, especially in the cloud environment. This requires the experiment to be repeated many times to increase the confidence in the result. Moreover, the higher amount of workload, i.e. number of jobs and tasks, is needed to show if our research can significantly reduce to monetary cost.

Update parameter dynamically during runtime. At the moment, the dynamic scheduling mechanism is only able to reassign tasks between VMs. However, we believe that it will be beneficial to update the parameters, e.g. task execution time, during runtime as well in order to have a better data for scheduling.

Handle multiple unknown jobs concurrently. Currently, the unknown application handling is able to handle only one unknown job at a time. In order to make our research suitable to be used by multiple parties, it is necessary to update the mechanism so that multiple unknown application can be handled at the same time.

Better approaches for estimating task execution time. Our unknown application handling mechanism estimates task execution times. However, this method implicitly assumes that the tasks execution times of an application follow a normal distribution. In other to support application whose performance is non-deterministic, we should investigate other approaches to estimate an application's task execution times.

Supporting different types of application. In this thesis, we only focus on one type of application, namely Bag-of-Tasks. There are many other types of application such as workflow, MapReduce, user-facing, etc. Each of them has different characteristics and requirements. However, they all require an effective scheduling mechanism in order to achieve the desired performance with minimum cost while being executed on the cloud.

Apply our research in existing framework. In order to present our research to wider audience, it is necessary to make it a pluggable component to existing resource management and/or execution scheduling frameworks, such as Hadoop or Spark.

References

- [1] Amazon web service, . URL <https://aws.amazon.com/>.
- [2] Amazon ec2 instance types, . URL <https://aws.amazon.com/ec2/instance-types/>.
- [3] Amazon ec2 pricing, . URL <https://aws.amazon.com/ec2/pricing/>.
- [4] Amazon ec2 reserved instances, . URL <https://aws.amazon.com/ec2/purchasing-options/reserved-instances/>.
- [5] Amazon ec2 spot instances, . URL <https://aws.amazon.com/ec2/spot/>.
- [6] Microsoft azure: Cloud computing platform and services. URL <https://azure.microsoft.com/en-gb/>.
- [7] Google cloud platform, . URL <https://cloud.google.com/>.
- [8] Preemptible virtual machines, . URL <https://cloud.google.com/preemptible-vms/>.
- [9] Apache hadoop. URL <http://hadoop.apache.org/>.
- [10] Netflix and stolen time. URL <http://blog.sciencelogic.com/netflix-steals-time-in-the-cloud-and-from-users/03/2011>.
- [11] Apache spark - lightning-fast cluster computing. URL <http://spark.apache.org/>.
- [12] Gurobi Optimizer Reference Manual, 2015. URL <http://www.gurobi.com>.
- [13] Akka, 2016. URL <http://akka.io/>.
- [14] Amazon s3, 2016. URL <https://aws.amazon.com/s3/>.
- [15] Azure storage, 2016. URL <https://azure.microsoft.com/en-gb/services/storage/>.
- [16] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html>.
- [17] Tekin Bicer, David Chiu, and Gagan Agrawal. Time and cost sensitive data-intensive computing on hybrid clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 636–643, May 2012. doi: 10.1109/CCGrid.2012.95.

- [18] Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 0321356683, 9780321356680.
- [19] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 285–300, 2014.
- [20] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6): 599–616, 2009. ISSN 0167-739X. doi: <http://dx.doi.org/10.1016/j.future.2008.12.001>.
- [21] David Candeia, Ricardo Araujo, Raquel Lopes, and Francisco Brasileiro. Investigating business-driven cloudburst schedulers for e-science bag-of-tasks applications. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 343–350, Nov 2010. doi: 10.1109/CloudCom.2010.67.
- [22] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000) (Cat. No.PR00556)*, pages 349–363, 2000. doi: 10.1109/HCW.2000.843757.
- [23] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46(1):131–159, 2002. ISSN 1573-0565. doi: 10.1023/A:1012450327387. URL <http://dx.doi.org/10.1023/A:1012450327387>.
- [24] Ryan Chard, Kyle Chard, Kris Bubendorfer, Lukasz Lacinski, Ravi Madduri, and Ian Foster. Cost-Aware Elastic Cloud Provisioning for Scientific Workloads. *2015 IEEE 8th International Conference on Cloud Computing*, pages 971–974, 2015. doi: 10.1109/CLOUD.2015.130. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7214142>.
- [25] Ron C. Chiang, Jinho Hwang, H. Howie Huang, and Timothy Wood. Matrix: Achieving predictable virtual machine performance in the clouds. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 45–56, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-11-9. URL <https://www.usenix.org/conference/icac14/technical-sessions/presentation/chiang>.
- [26] Walfredo Cirne and Eitan Frachtenberg. *Job Scheduling Strategies for Parallel Processing: 16th International Workshop, JSSPP 2012, Shanghai, China, May 25, 2012. Revised Selected Papers*, chapter Web-Scale Job Scheduling, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-35867-8. doi: 10.1007/978-3-642-35867-8_1. URL http://dx.doi.org/10.1007/978-3-642-35867-8_1.
- [27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL <http://doi.acm.org/10.1145/1327452.1327492>.

- [28] Rubing Duan, Radu Prodan, and Xiaorong Li. Multi-objective game theoretic scheduling of bag-of-tasks workflows on hybrid clouds. *IEEE Transactions on Cloud Computing*, 2(1):29–42, Jan 2014. ISSN 2168-7161. doi: 10.1109/TCC.2014.2303077.
- [29] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the ACM European Conference on Computer Systems*, pages 99–112, 2012.
- [30] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. In *USENIX Annual Technical Conference*, pages 459–471, 2015.
- [31] Nikolay Grozev and Rajkumar Buyya. Inter-cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390, 2014. ISSN 1097-024X. doi: 10.1002/spe.2168. URL <http://dx.doi.org/10.1002/spe.2168>.
- [32] J. Octavio Gutierrez-Garcia and Kwang Mong Sim. A family of heuristics for agent-based elastic cloud bag-of-tasks concurrent scheduling. *Future Gener. Comput. Syst.*, 29(7):1682–1699, September 2013. ISSN 0167-739X. doi: 10.1016/j.future.2012.01.005. URL <http://dx.doi.org/10.1016/j.future.2012.01.005>.
- [33] Martin Haugh. The monte carlo framework, examples from finance and generating correlated random variables. *Course Notes*, 2004.
- [34] Mohammad Reza HoseinyFarahabady, Young Choon Lee, and Albert Zomaya. Pareto-optimal cloud bursting. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2670–2682, Oct 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.218.
- [35] Alexandru Iosup and Dick Epema. Grid computing workloads. *IEEE Internet Computing*, 15(2):19–26, March 2011. ISSN 1089-7801. doi: 10.1109/MIC.2010.130. URL <http://dx.doi.org/10.1109/MIC.2010.130>.
- [36] Hyejeong Kang, Jung-in Koh, Yoonhee Kim, and Jaegyeon Hahm. A sla driven vm auto-scaling method in hybrid cloud environment. In *2013 15th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 1–6, Sept 2013.
- [37] Huafeng Xu Ron O. Dror Michael P. Eastwood Brent A. Gregersen John L. Klepeis Istvan Kolossvary Mark A. Moraes Federico D. Sacerdoti John K. Salmon Yibing Shan David E. Shaw Kevin J. Bowers, Edmond Chow. Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters. In *ACM/IEEE Supercomputing Conference*, pages 43–43, 2006.
- [38] Dirk P. Kroese, Tim Brereton, Thomas Taimre, and Zdravko I. Botev. Why the monte carlo method is so important today. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6(6):386–392, 2014. ISSN 1939-0068. doi: 10.1002/wics.1314. URL <http://dx.doi.org/10.1002/wics.1314>.
- [39] Ulrich Lampe, Melanie Siebenhaar, Ronny Hans, Dieter Schuller, and Ralf Steinmetz. *Economics of Grids, Clouds, Systems, and Services: 9th International Conference, GECON 2012, Berlin, Germany, November 27-28, 2012. Proceedings*, chapter Let the Clouds Compute: Cost-Efficient Workload Distribution in Infrastructure Clouds,

- pages 91–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35194-5. doi: 10.1007/978-3-642-35194-5_7. URL http://dx.doi.org/10.1007/978-3-642-35194-5_7.
- [40] Philipp Leitner and Jürgen Cito. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16(3):15:1–15:23, April 2016. ISSN 1533-5399. doi: 10.1145/2885497. URL <http://doi.acm.org/10.1145/2885497>.
- [41] Steve Lohr. Google and I.B.M. Join in 'Cloud Computing' Research. 2007. <http://www.nytimes.com/2007/10/08/technology/08cloud.html>.
- [42] Sifei Lu, Xiaorong Li, Long Wang, H. Kasim, H. Palit, T. Hung, E. F. T. Legara, and G. Lee. A dynamic hybrid resource provisioning approach for running large-scale computational applications on cloud spot and on-demand instances. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 657–662, Dec 2013. doi: 10.1109/ICPADS.2013.117.
- [43] Zoltán Ádám Mann. Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms. *ACM Comput. Surv.*, 48(1):11:1–11:34, August 2015. ISSN 0360-0300. doi: 10.1145/2797211. URL <http://doi.acm.org/10.1145/2797211>.
- [44] Ming Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430, June 2012. doi: 10.1109/CLOUD.2012.103.
- [45] Ming Mao, Jie Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 41–48, Oct 2010. doi: 10.1109/GRID.2010.5697966.
- [46] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.
- [47] Ishai Menache, Ohad Shamir, and Navendu Jain. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 177–187, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-11-9. URL <https://www.usenix.org/conference/icac14/technical-sessions/presentation/menache>.
- [48] John Norstad. Financial planning using random walks, 1999.
- [49] A. Oprescu and T. Kielmann. Bag-of-Tasks Scheduling under Budget Constraints. In *IEEE International Conference on Cloud Computing Technology and Science*, pages 351–359, 2010.
- [50] Ana-Maria Oprescu, Thilo Kielmann, and Haralambie Leahu. Stochastic tail-phase optimization for bag-of-tasks execution in clouds. In *IEEE Fifth International Conference on Utility and Cloud Computing, UCC 2012, Chicago, IL, USA, November 5-8, 2012*, pages 204–208, 2012. doi: 10.1109/UCC.2012.23. URL <http://doi.ieeecomputersociety.org/10.1109/UCC.2012.23>.

- [51] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of amazon ec2. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing, HotCloud'12*, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association. URL <http://dl.acm.org/citation.cfm?id=2342763.2342767>.
- [52] John O'Loughlin and Lee Gillam. Good performance metrics for cloud service brokers. In *The Fifth International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 64–69. Citeseer, 2014.
- [53] Victor Pelaez, Antonio Campos, Daniel Garcia, and Joaquin Entrialgo. Autonomic scheduling of deadline-constrained bag of tasks in hybrid clouds. In *2016 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 1–8, July 2016. doi: 10.1109/SPECTS.2016.7570526.
- [54] Eric Pettijohn, Yanfei Guo, Palden Lama, and Xiaobo Zhou. User-centric heterogeneity-aware mapreduce job provisioning in the public cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 137–143, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-11-9. URL <http://blogs.usenix.org/conference/icac14/technical-sessions/presentation/pettijohn>.
- [55] Arkaitz Ruiz-Alvarez, In Kee Kim, and Marty Humphrey. Toward optimal resource provisioning for cloud mapreduce and hybrid cloud applications. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 669–677, June 2015. doi: 10.1109/CLOUD.2015.94.
- [56] Siqi Shen, Kefeng Deng, Alexandru Iosup, and Dick Epema. Scheduling jobs in the cloud using on-demand and reserved instances. In *Proceedings of the 19th International Conference on Parallel Processing, Euro-Par'13*, pages 242–254, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-40046-9. doi: 10.1007/978-3-642-40047-6_27. URL http://dx.doi.org/10.1007/978-3-642-40047-6_27.
- [57] Subhajit Sidhanta, Wojciech Golab, and Supratik Mukhopadhyay. Optex: A deadline-aware cost optimization model for spark. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 193–202, May 2016. doi: 10.1109/CCGrid.2016.10.
- [58] Shaojie Tang, Jing Yuan, and Xiang Yang Li. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. *Proceedings - 2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012*, pages 91–98, 2012. ISSN 2159-6182. doi: 10.1109/CLOUD.2012.134.
- [59] Long Thai, Blesson Varghese, and Adam Barker. Executing Bag of Distributed Tasks on the Cloud: Investigating the Trade-Offs between Performance and Cost. In *IEEE Conference on Cloud Computing Technology and Science*, pages 400–407, 2014.
- [60] Long Thai, Blesson Varghese, and Adam Barker. Budget constrained execution of multiple bag-of-tasks applications on the cloud. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 975–980, June 2015. doi: 10.1109/CLOUD.2015.131.

- [61] Long Thai, Blesson Varghese, and Adam Barker. Task scheduling on the cloud with hard constraints. In *Services (SERVICES), 2015 IEEE World Congress on*, pages 95–102, June 2015. doi: 10.1109/SERVICES.2015.22.
- [62] Long Thai, Blesson Varghese, and Adam Barker. Executing bag of distributed tasks on virtually unlimited cloud resources. In *CLOSER 2015 - Proceedings of the 5th International Conference on Cloud Computing and Services Science, Lisbon, Portugal, 20-22 May, 2015.*, pages 373–380, 2015. doi: 10.5220/0005403303730380. URL <http://dx.doi.org/10.5220/0005403303730380>.
- [63] Long Thai, Blesson Varghese, and Adam Barker. Minimising the Execution of Unknown Bag-of-Task Jobs with Deadlines on the Cloud. In *ACM International Workshop on Data-Intensive Distributed Computing*, pages 3–10, 2016.
- [64] Long Thai, Blesson Varghese, and Adam Barker. Algorithms for optimising heterogeneous cloud virtual machine clusters. In *Cloud Computing Technology and Science (CloudCom), 2016 IEEE 8th International Conference on*, Dec 2016.
- [65] Ruben Van Den Bossche, Kurt Vanmechelen, and Jan Broeckhove. Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. *Future Gener. Comput. Syst.*, 29(4):973–985, June 2013. ISSN 0167-739X. doi: 10.1016/j.future.2012.12.012. URL <http://dx.doi.org/10.1016/j.future.2012.12.012>.
- [66] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009. ISSN 0146-4833. doi: <http://doi.acm.org/10.1145/1496091.1496100>.
- [67] Blesson Varghese, Ozgur Akgun, Ian Miguel, Long Thai, and Adam Barker. Cloud benchmarking for performance. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 535–540, Dec 2014. doi: 10.1109/CloudCom.2014.28.
- [68] Bo Wang, Ying Song, Yuzhong Sun, and Jun Liu. Managing deadline-constrained bag-of-tasks jobs on hybrid clouds. In *Proceedings of the 24th High Performance Computing Symposium, HPC '16*, pages 22:1–22:8, San Diego, CA, USA, 2016. Society for Computer Simulation International. ISBN 978-1-5108-2318-1. doi: 10.22360/SpringSim.2016.HPC.039. URL <http://dx.doi.org/10.22360/SpringSim.2016.HPC.039>.
- [69] Jonathan Stuart Ward and Adam Barker. Observing the clouds: a survey and taxonomy of cloud monitoring. *Journal of Cloud Computing*, 3(1):24, 2014. doi: 10.1186/s13677-014-0024-2. URL <http://dx.doi.org/10.1186/s13677-014-0024-2>.
- [70] Min Yao, Peng Zhang, Yin Li, Jie Hu, Chuang Lin, and Xiang Yang Li. Cutting your cloud computing cost for deadline-constrained batch jobs. In *Web Services (ICWS), 2014 IEEE International Conference on*, pages 337–344, June 2014. doi: 10.1109/ICWS.2014.56.
- [71] Sangho Yi, Artur Andrzejak, and Derrick Kondo. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *IEEE Transactions on Services Computing*, 5(4):512–524, Fourth 2012. ISSN 1939-1374. doi: 10.1109/TSC.2011.44.

-
- [72] Xingquan Zuo, Guoxiang Zhang, and Wei Tan. Self-adaptive learning pso-based deadline constrained task scheduling for hybrid iaas cloud. *IEEE Transactions on Automation Science and Engineering*, 11(2):564–573, April 2014. ISSN 1545-5955. doi: 10.1109/TASE.2013.2272758.

Appendix A

Experiment Results

This chapter presents the detailed experiment results which were illustrated in Chapter 8.

A.1 Scheduling Approaches

Table A.1 Summary of Solving Times in Milliseconds of the Experiment in which the Number of Jobs Varied

Number of Jobs	Exact Approach				Hybrid Approach				Heuristic Approach			
	Mean	Std	Min	Max	Mean	Std	Min	Max	Mean	Std	Min	Max
5	804.2	517.13	226	1212	58.4	41.11	23	124	41	21.08	19	64
6	1974	1706.08	922	4977	45.6	32.04	23	102	33.2	21.88	12	68
7	5365	5728.27	2144	15485	50.4	27.87	26	91	44	32.86	16	98
8	122597.2	267324.94	1732	600792	31.8	6.02	26	40	19.4	3.97	14	24
9	19686	11961.77	6092	35153	36.4	5.86	30	44	28.8	8.76	19	40
10	185662.2	242698.15	17834	603303	43.6	4.83	38	49	34	10.77	24	48
11	190486.8	239852.84	26773	604623	61.4	12.90	43	75	36	11.51	26	55
12	395315	238504.11	99183	609248	60.4	9.63	51	75	41.6	8.76	28	52
13	494983.4	146981.08	266490	609168	64.6	14.59	44	84	44	15.25	32	70
14	545362.8	151698.30	274066	617591	73	11.07	60	86	62.4	14.57	53	88

Table A.2 Summary of Total Costs in Dollar of the Experiment in which the Number of Jobs Varied

Number of Jobs	Exact Approach				Hybrid Approach				Heuristic Approach			
	Mean	Std	Min	Max	Mean	Std	Min	Max	Mean	Std	Min	Max
5	25.2	7.60	14	32	26.6	7.92	15	34	27.2	8.58	15	36
6	36.2	7.40	31	49	37.8	8.17	32	52	38.2	8.07	32	52
7	46.6	9.76	39	60	47.8	9.63	40	61	48.2	10.33	40	63
8	53.8	9.31	39	64	55.8	9.31	41	66	56.4	9.48	41	67
9	74.2	12.91	53	85	76.6	13.76	54	88	77.6	12.90	56	88
10	98.4	17.04	79	122	101.6	17.60	81	125	102.4	18.93	81	128
11	111.8	18.66	98	143	115.2	19.41	101	148	116.4	19.93	100	150
12	145.8	28.80	113	191	149.6	29.16	116	195	152.2	29.89	117	197
13	158	18.07	126	169	164	18.73	131	177	165.2	18.75	132	177
14	181.4	28.66	152	213	187.2	29.52	158	219	187.6	28.94	158	220

Table A.3 Summary of Solving Times in Milliseconds of the Experiment in which the Number of Tasks Varied

Number of Tasks	Exact Approach				Hybrid Approach				Heuristic Approach			
	Mean	Std	Min	Max	Mean	Std	Min	Max	Mean	Std	Min	Max
100	927	335.62	491	1317	58.8	41.91	28	131	37.4	19.92	21	70
200	6571.2	10145.68	839	24657	42.2	11.99	27	60	34.2	12.60	21	54
300	96982.4	196785.23	824	448698	41.8	17.31	25	68	36.6	10.64	23	51
400	9790.4	3327.00	5050	12912	33.6	4.93	27	40	28.4	7.47	19	37
500	17973.8	13331.18	2825	36816	34.8	13.41	20	50	24.2	12.07	11	41
600	145527.6	255658.24	16943	602483	35.8	8.47	28	49	28	7.81	21	41
700	372816.2	316189.49	18021	604518	48	16.31	27	69	51.2	23.37	25	87
800	147298.6	255060.25	27572	603507	50	12.92	31	61	48.2	17.82	21	68
900	78128.6	21256.75	60276	111231	60.2	21.41	44	96	52.8	18.07	36	80
1000	259813.2	148914.66	80561	458990	86.8	21.99	58	111	82.6	25.95	45	107

Table A.4 Summary of Total Costs in Dollars of the Experiment in which the Number of Tasks Varied

Number of Tasks	Exact Approach				Hybrid Approach				Heuristic Approach			
	Mean	Std	Min	Max	Mean	Std	Min	Max	Mean	Std	Min	Max
100	26.4	5.18	22	35	27.8	5.63	23	37	28	5.57	23	37
200	53.8	9.93	40	68	54.8	10.01	41	69	55.6	9.58	42	69
300	71	15.60	49	91	72.2	15.93	50	93	73	16.17	51	95
400	98.2	17.74	75	118	99.6	17.92	76	120	100.6	18.39	77	123
500	128	39.38	76	163	129.4	39.70	77	165	131	40.36	78	168
600	163.2	13.83	146	183	166	13.73	149	186	167	12.67	151	185
700	219.2	37.24	175	256	222	38.21	177	261	222	38.18	177	260
800	207.6	27.40	180	245	210	28.64	181	249	210.8	28.49	182	250
900	268	34.96	231	319	270.2	34.69	234	321	271	34.56	235	322
1000	313.6	39.18	278	361	318.6	41.42	281	372	318.8	42.53	280	373

Table A.5 Summary of Solving Times in Milliseconds of the Experiment in which the Number of Instance Types Varied

Number of Types	Exact Approach				Hybrid Approach				Heuristic Approach			
	Mean	Std	Min	Max	Mean	Std	Min	Max	Mean	Std	Min	Max
5	1611.6	1151.78	517	3317	60.8	43.08	32	137	44.6	34.40	27	106
6	631.6	400.76	168	1112	26	4.69	22	34	17.4	3.85	13	22
7	995.4	931.59	310	2588	21.4	2.79	19	26	14.2	2.49	12	18
8	1022.8	623.45	440	2034	21.4	6.54	14	30	13	4.36	9	20
9	1502.2	988.64	438	2622	17.4	1.95	15	20	9.8	1.48	8	12
10	1436.2	467.97	832	1892	16.4	2.51	14	20	9.2	3.56	5	13
11	1370.6	832.80	664	2308	16.2	1.10	15	18	10.4	1.52	9	12
12	1252.4	554.98	686	2161	15.4	1.14	14	17	8	2.24	5	11
13	2513	2522.01	559	6343	16.2	3.27	13	21	8	2.92	5	12
14	1494	690.92	635	2520	39.8	29.69	15	91	23.2	15.66	7	47

Table A.6 Summary of Total Costs in Dollars of the Experiment in which the Number of Instance Types Varied

Number of Types	Exact Approach				Hybrid Approach				Heuristic Approach			
	Mean	Std	Min	Max	Mean	Std	Min	Max	Mean	Std	Min	Max
5	28.4	5.22	20	33	29.8	5.07	22	35	30	5.34	22	36
6	26	4.95	20	33	27.2	5.07	21	34	27.6	5.41	21	35
7	30.4	7.20	25	43	31.6	7.64	26	45	31.2	7.33	26	44
8	23.4	4.72	18	31	24.8	4.38	20	32	25	4.36	20	32
9	29	7.04	22	37	30	7.04	23	38	30.4	7.33	23	39
10	27	3.32	23	32	28.2	3.42	24	33	28.2	3.70	23	33
11	31.2	5.12	27	40	32.8	5.26	29	42	32.8	5.26	29	42
12	27.2	5.89	20	34	28.6	5.94	21	35	28.6	5.94	21	35
13	30.6	7.96	20	41	32.2	8.44	21	43	32.4	8.56	21	44
14	27.2	7.66	17	38	28.6	8.11	18	40	29	8.15	18	40

Table A.7 Summary of Solving Times in Milliseconds of the Experiment in which the Deadline Varied

Deadline	Exact Approach				Hybrid Approach				Heuristic Approach			
	Mean	Std	Min	Max	Mean	Std	Min	Max	Mean	Std	Min	Max
600	2122.6	2311.69	626	6208	51.6	28.52	30	101	46.8	21.74	28	82
700	304.8	145.78	165	545	22.6	8.47	16	36	21.6	7.30	16	34
800	351.4	103.34	205	457	16.2	3.42	13	22	11	3.16	7	15
900	966	1100.81	90	2680	16	1.58	14	18	11.4	1.52	10	13
1000	177	65.49	105	270	12	2.24	9	15	7.2	1.92	5	10
1100	162	61.48	77	249	9.8	0.84	9	11	5.4	0.55	5	6
1200	298.6	158.65	116	543	10.8	1.30	9	12	6.2	1.30	5	8
1300	145.2	60.69	61	220	9.4	1.14	8	11	4.6	0.89	4	6
1400	192.8	114.02	54	340	10	0.71	9	11	5.2	0.45	5	6
1500	197.6	60.16	111	281	10.2	2.77	7	14	5	0.71	4	6

Table A.8 Summary of Total Costs in Dollars of the Experiment in which the Deadline Varied

Deadline	Exact Approach				Hybrid Approach				Heuristic Approach			
	Mean	Std	Min	Max	Mean	Std	Min	Max	Mean	Std	Min	Max
600	25.4	6.07	17	32	27	5.61	19	33	27.4	6.11	18	33
700	20.2	4.76	16	27	21.4	4.72	17	28	22	5.00	17	29
800	18.6	3.36	14	21	19.2	3.03	15	22	20	3.24	16	23
900	18.8	1.92	16	21	20	2.24	17	23	20	2.24	17	23
1000	13.8	0.84	13	15	14.6	1.14	13	16	14.6	1.14	13	16
1100	11.8	3.03	9	17	12.6	3.13	10	18	12.6	3.13	10	18
1200	14.2	2.28	12	18	15	2.45	13	19	15	2.92	13	20
1300	10.4	1.95	8	13	11.4	1.82	9	14	11.4	2.30	9	15
1400	10.6	1.52	9	13	11.4	1.14	10	13	11.6	0.89	11	13
1500	9.6	1.52	8	11	10.4	1.82	8	12	10.4	1.82	8	12

A.2 Evaluating the Unknown Handling Mechanism

Table A.9 The results of the experiment evaluating the Unknown Handling Mechanism

	Late Task		Late Time		Cost (In Dollars)		Violation Cost
	Mean	Std	Mean	Std	Mean	Std	
known	2.4	1.2	46.0	30.69	2.63	0.17	0.0141
unknown	2.8	3.43	34.6	38.12	2.91	0.17	0.0182
medium.8	126.0	14.18	6174.2	514.49	2.91	0.03	1.1299
medium.10	19.4	1.36	865.6	62.37	2.92	0	0.1316
large.4	76.6	34.56	2292.0	1602.29	2.42	0.12	0.4972
large.5	5.6	4.08	85.8	69.35	2.92	0	0.0368
xlarge.2	360.8	16.79	38812.4	2922.91	3.52	0	14.2217
xlarge.3	203.4	42.32	5582.6	1946.4	4.04	0.34	3.3351

A.3 Dynamic Reassignment

Table A.10 Cost in Dollars of Each Approach When Dynamic Scheduling Is Turned On/Off

Coefficient of Variation	Exact Approach				Hybrid Approach				Heuristic Approach			
	Dyna		NoDyna		Dyna		NoDyna		Dyna		NoDyna	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
0.00	2.12	0.00	2.12	0.00	2.12	0.00	2.12	0.00	2.13	0.03	2.19	0.00
0.25	2.23	0.07	2.42	0.09	2.34	0.12	2.45	0.12	2.41	0.19	2.41	0.17
0.50	2.63	0.15	2.67	0.10	2.60	0.06	2.73	0.15	2.42	0.05	2.64	0.13
0.75	2.74	0.14	3.14	0.13	2.86	0.18	3.11	0.14	2.73	0.17	2.96	0.07
1.00	3.33	0.13	3.33	0.19	3.26	0.11	3.39	0.15	3.15	0.17	3.29	0.10

Table A.11 Number of Violated Tasks for Each Approach When Dynamic Scheduling Is Turned On/Off

Coefficient of Variation	Exact Approach				Hybrid Approach				Heuristic Approach			
	Dyna		NoDyna		Dyna		NoDyna		Dyna		NoDyna	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.25	0.60	0.80	7.40	1.85	2.00	1.90	49.20	77.91	0.20	0.40	46.40	78.32
0.50	5.20	3.31	26.40	6.53	3.40	3.93	26.80	6.85	1.20	1.47	25.20	4.71
0.75	21.00	11.71	87.60	15.91	22.80	11.92	75.40	25.68	29.20	20.72	68.40	12.37
1.00	115.60	37.36	120.20	17.96	93.60	28.71	150.60	16.70	98.00	33.54	140.80	17.03

Table A.12 Average Amount of Violated Time in Seconds for Each Approach When Dynamic Scheduling Is Turned On/Off

Coefficient of Variation	Exact Approach				Hybrid Approach				Heuristic Approach			
	Dyna		NoDyna		Dyna		NoDyna		Dyna		NoDyna	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.25	7.80	10.17	76.40	13.92	40.40	35.83	106.00	26.43	6.00	12.00	93.60	36.82
0.50	83.40	47.49	385.20	57.69	55.20	61.69	308.80	48.14	18.80	24.90	298.20	120.31
0.75	377.80	236.43	1249.60	187.58	360.40	142.94	1041.40	205.37	347.20	153.51	746.20	37.39
1.00	1228.20	302.46	1479.40	327.77	1269.60	449.23	1626.00	188.15	910.20	271.02	1409.00	159.80

A.4 Cloud Experiments

Table A.13 The results of the experiments evaluating the feasibility of the proposed research in real life cloud, i.e. AWS

	Total Cost (In Dollars)	Dyna				NoDyna			
		Late Task		Late Time		Late Task		Late Time	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std
Exact	1.61	0.0	0.0	0.0	0.0	2.33	1.25	39.0	29.22
Hybrid	1.68	0.0	0.0	0.0	0.0	11.0	4.97	157.33	110.08
Heuristic	1.68	0.0	0.0	0.0	0.0	25.67	36.3	22.67	32.06
Medium	1.97	0.67	0.94	7.67	10.84	35.67	4.78	411.0	84.25
Large	1.75	0.0	0.0	0.0	0.0	11.33	3.09	160.33	55.42
XLarge	2.34	0.0	0.0	0.0	0.0	0.33	0.47	7.67	10.84