Resource Provisioning and Scheduling Algorithms for Scientific Workflows in Cloud Computing Environments

Maria Alejandra Rodriguez Sossa

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

Department of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE, AUSTRALIA

June 2016

Produced on archival quality paper.

Resource Provisioning and Scheduling Algorithms for Scientific Workflows in Cloud Computing Environments

Maria Alejandra Rodriguez Sossa
Principal Supervisor: Prof. Rajkumar Buyya

Abstract

Scientific workflows describe a series of computations that enable the analysis of data in a structured and distributed manner. Their importance is exacerbated in todays big data era as they become a compelling mean to process and extract knowledge from the ever-growing data produced by increasingly powerful tools such as telescopes, particle accelerators, and gravitational wave detectors. Due to their large-scale nature, scheduling algorithms are key to efficiently automate their execution in distributed environments, and as a result, to facilitate and accelerate the pace of scientific progress.

The emergence of the latest distributed system paradigm, cloud computing, brings with it tremendous opportunities to run workflows at low costs without the need of owning any infrastructure. In particular, Infrastructure as a Service (IaaS) clouds, offer an easily accessible, flexible, and scalable infrastructure for the deployment of these scientific applications by providing access to a virtually infinite pool of resources that can be acquired, configured, and used as needed and are charged on a pay-per-use basis.

This thesis investigates novel resource provisioning and scheduling approaches for scientific workflows in IaaS clouds. They address fundamental challenges that arise from the multi-tenant, resource-abundant, and elastic resource model and are capable of fulfilling a set of quality of service requirements expressed in terms of execution time and cost. It advances the field by making the following key contributions:

1. A taxonomy and survey of the state-of-the-art scientific workflow scheduling algorithms designed exclusively for IaaS clouds.

2. A novel static scheduling algorithm that leverages Particle Swarm Optimization to generate a workflow execution and resource provisioning plan that minimizes the infrastructure cost while meeting a deadline constraint.

3. A hybrid algorithm based on a variation of the Unbounded Knapsack Problem that finds a trade-off between making static decisions to find better-quality schedules and dynamic decisions to adapt to unexpected delays.

4. A scalable algorithm that combines heuristics and two different Integer Programming models to generate schedules that minimize the execution time of the workflow while meeting a budget constraint.

5. The implementation of a cloud resource management module and its integration to an existing Workflow Management System.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Maria Alejandra Rodriguez Sossa, June 2016

Acknowledgements

I would like to thank my supervisor, Professor Rajkumar Buyya, for giving me the opportunity to undertake this PhD. I am deeply grateful for his invaluable guidance, advice, and motivation throughout my candidature. I especially thank him for providing me with all the tools that enabled me to successfully complete this thesis. I am also deeply grateful to my co-supervisor Professor Raomamohanarao Kotagiri for his useful comments and advice.

I would like to express my gratitude to the PhD committee members, Professor James Bailey and Professor Benjamin Rubinstein, for their constructive comments and guidance during my candidature. Special thanks to Dr. Rodrigo Calheiros for helping me improve my research skills, for his invaluable guidance and advice, and his suggestions on improving my work. I would also like to thank all the past and current members of the CLOUDS Laboratory, at the University of Melbourne for their friendship and support. In particular I thank Dr. Amir Vahid, Dr. Adel Nadjaran Toosi, Dr. Nikolay Grozev, Dr. Deepak Poola, Sareh Fotuhi, Atefeh Khosravi, Yaser Mansouri, Chenhao Qu, Yali Zhao, Jungmin Jay Son, Bowen Zhou, Farzad Khodadadi, Safiollah Heidari, Deborah Magalhes, Tiago Justino, Liu Xunyun, Caesar Wu, and Mohammed Alrokayan. I am especially grateful to Rekha Mangal for her helpful advice and sincere friendship.

I acknowledge the University of Melbourne and Australian Federal Government for providing me with scholarships to pursue my doctoral studies.

I would like to thank my parents, my brothers, my sister, and Roci for their unconditional and loving support. I thank my father for always believing in me, for encouraging me to achieve every step of this difficult journey, and for his words of praise when they were most needed. I thank my mother for being an exceptional role model, for teaching

me the meaning of perseverance and hard work, and for her words of advice and caring love.

I thank my parents- and sister-in-law for being a loving, caring, and supportive family away from home. Finally, I would like to thank my husband for his selflessness, unconditional love, endless support, and inspiration. Achieving this amazing goal would not have been possible without him and I will forever be grateful for that.

Maria Alejandra Rodriguez Sossa
Melbourne, Australia
June 2016

Preface

This thesis research has been carried out in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapters 2- 6 and are based on the following publications:

- **Maria A. Rodriguez** and Rajkumar Buyya. "A Taxonomy and Survey on Scheduling Algorithms for Scientific Workflows in IaaS Cloud Computing Environments." *Concurrency and Computation: Practice and Experience*, DOI:10.1002/cpe.4041, 2016.

- **Maria A. Rodriguez** and Rajkumar Buyya. "Deadline Based Resource Provisioning and Scheduling Algorithm for Scientific Workflows on Clouds." *IEEE Transactions on Cloud Computing*, Volume 2, Issue 2, Pages: 222-235, 2014.

- **Maria A. Rodriguez** and Rajkumar Buyya. "A Responsive Knapsack-based Algorithm for Resource Provisioning and Scheduling of Scientific Workflows in Clouds." *In Proceedings of the 44th International Conference on Parallel Processing (ICPP)*, Pages 839-848, 2015.

- **Maria A. Rodriguez** and Rajkumar Buyya. "Budget-Driven Resource Provisioning and Scheduling of Scientific Workflow in IaaS Clouds with Fine-Grained Billing Periods." *ACM Transactions on Autonomous and Adaptive Systems*, 2016 (under second review).

- **Maria A. Rodriguez** and Rajkumar Buyya. "Scientific Workflow Management System for Clouds" *Software Architecture for Big Data and the Cloud*, Elsevier - Morgan Kaufmann, 2017 (in press).

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Workflows are a commonly used application model in computational science. They describe a series of computations that enable the analysis of data in a structured and distributed manner and have been successfully used to make significant scientific advances in various fields such as biology, physics, medicine, and astronomy [53]. Their importance is highlighted in today's big data era as they offer an efficient way of processing and extracting knowledge from the ever-growing data produced by increasingly powerful tools such as telescopes, particle accelerators, and gravitational wave detectors.

The emergence of cloud computing has brought with it several advantages for the deployment of large-scale scientific workflows. In particular, Infrastructure as a Service (IaaS) clouds offer an easily accessible, flexible, and scalable infrastructure for the deployment of these applications. IaaS vendors provide the opportunity to deploy workflows at low costs and without the need of owning any infrastructure by leasing virtualized compute resources, or Virtual Machines (VMs). This allows workflows to be easily packaged and deployed and more importantly, enables workflow management systems to access a virtually infinite pool of VMs that can be elastically acquired and released and are charged on a pay-per-use basis. In this way, a workflow's resource usage can be adjusted over time based on the current application needs.

Scheduling algorithms are key in leveraging these benefits and in general, to efficiently automate the execution of scientific workflows in distributed environments. These algorithms are an essential component of workflow management systems and are responsible for orchestrating the execution of tasks in a set of compute resources while preserving the data dependencies. Figure 1.1 depicts a high-level overview of the compo-

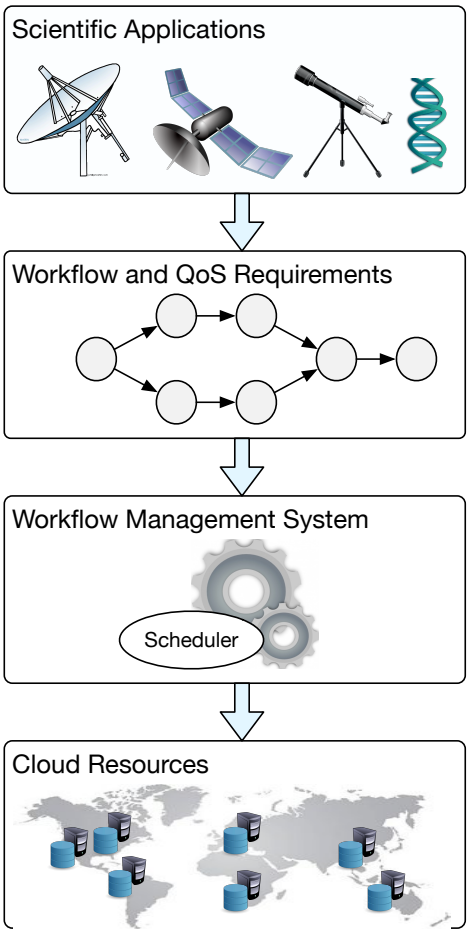


Figure 1.1: A high-level view of a cloud workflow execution environment.

nents involved in the deployment of a workflow in a cloud computing environment. The decisions made by scheduling algorithms are generally guided by a set of user-defined Quality of Service (QoS) requirements. Their success in fulfilling these QoS requirements relies on the effective use of the underlying resources and as a result, schedulers need to be aware of various challenges that are derived from features inherent to the cloud resource model.

Firstly, when compared to other distributed systems such as grids, clouds offer more control over the type and quantity of resources used. This flexibility and abundance of resources creates the need for a resource provisioning strategy that works together with the scheduling algorithm; a heuristic that decides the type and number of VMs to use and when to lease and to release them. Another challenge that must be addressed by

schedulers is finding a trade-off between performance, non-functional requirements, and cost to avoid paying unnecessary and potentially prohibitive prices. Finally, algorithms need to be aware of the dynamic nature of cloud platforms and the uncertainties this brings with it. For instance, VM provisioning and deprovisioning delays are generally highly variable and unpredictable and performance variation is observed in resources such as VM CPUs, network links, and storage systems. This indeterminism makes it difficult for algorithms to make accurate scheduling decisions.

As a result, this thesis addresses the problem of efficiently scheduling large-scale scientific workflows in IaaS cloud computing environments. It investigates novel scheduling and resource provisioning strategies that address the key challenges derived from the particular characteristics of clouds. This is achieved by developing a detailed taxonomy and a comprehensive survey based on state-of-the art algorithms. Additionally, a set of algorithms are proposed. They are tailored for the multi-tenant, elastic, utility-based, and resource-abundant cloud resource model and are highly successful in generating schedules capable of fulfilling a set of QoS requirements expressed in terms of execution time and cost. Finally, the architecture of an existing workflow management system is extended to support the cloud resource model. We present its implementation and demonstrate its capabilities with a case study using a real-life scientific workflow from the astronomy field.

1.1 Background

This section presents a high-level overview of the fundamental concepts related to the research problem addressed in the thesis.

1.1.1 Cloud Computing

Cloud computing enables the delivery of computing resources over the Internet on a pay-per-use basis. The NIST [82] defines this paradigm as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly

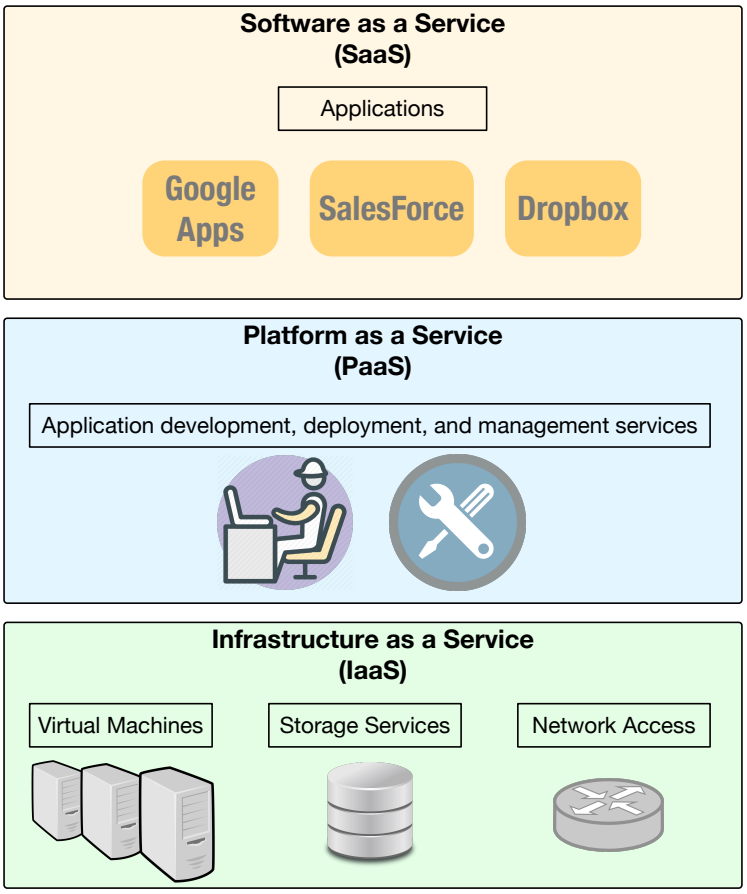


Figure 1.2: Cloud computing service offerings.

provisioned and released with minimal management effort or service provider interaction". The cloud model is composed of three different service offerings that can be classified into a hierarchy of as-a-service terms, namely Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). As shown in Figure 1.2, each of these service models represents a different abstraction level and as a whole, they can be understood as a layered architecture where the layers above leverage from the services provided by the layers below [33].

At the top of the stack is the SaaS layer. It provides access to applications over the web enabling users to utilize online software instead of locally installed one. Salesforce.com [17], a SaaS provider that offers CRM applications, describes software as a service and its benefits as "a way of delivering applications over the Internet as a ser-

vice. Instead of installing and maintaining software, you simply access it via the Internet, freeing yourself from complex software and hardware management."

The next layer in the stack is PaaS. Vendors who offer this type of service provide an environment in which developers can easily create and deploy their applications. Some features offered at this level include the provisioning of different programming models and specialized services that make the creation of applications simpler and more robust. For instance, applications deployed in a PaaS environment may have the ability scale automatically freeing developers from the responsibility of estimating the resource capacity of their applications under different scenarios. Google App Engine [7] is an example of a PaaS vendor.

Finally, at the bottom of the cloud computing stack is the IaaS layer. At this level, fundamental computing resources are provided. This is done by leasing VMs with a predefined CPU, memory, storage, and bandwidth capacity. Different resource bundles (i.e., VM types) are available at varying prices to suit a wide range of application needs. VMs can be elastically leased and released and are charged per time frame, or billing period. IaaS providers offer billing periods of different granularity, for example, Amazon EC2 [6] charges per hour while Microsoft Azure [13] is more flexible and charges on a per-minute basis. Aside from VMs, IaaS providers also offer storage services and network infrastructure to transport data in, out, and within their facilities.

The focus of this thesis is IaaS clouds and from here on, the word IaaS and cloud will be used to refer to this particular type of platforms interchangeably, unless specified otherwise. Although PaaS and SaaS clouds can be used to provide some services for workflow applications, IaaS clouds are the most relevant to this work as they provide all of the basic services needed for the deployment of workflows, including processors, storage, and network access.

1.1.2 Scientific Workflows

The concept of workflow has its roots in commercial enterprises as a business process modeling tool. These business workflows aim to automate and optimize the processes of an organization, seen as an ordered sequence of activities, and are a mature research

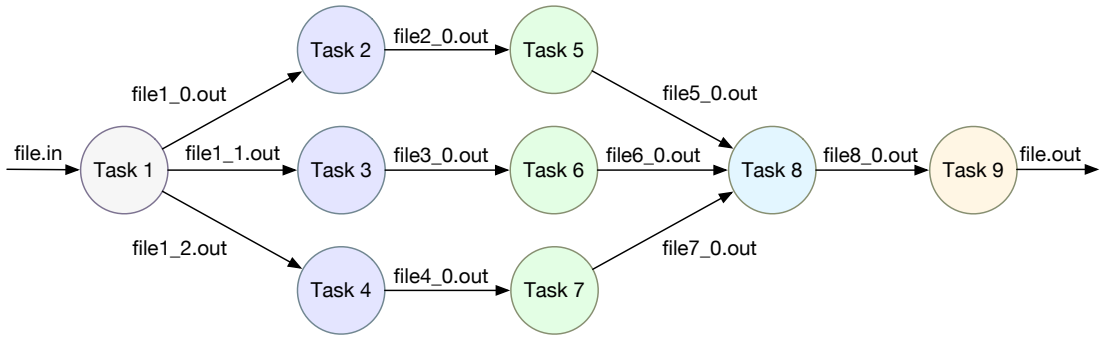


Figure 1.3: Sample workflow with nine tasks. The graph nodes represent computational tasks and the edges the data dependencies between these tasks.

area [115] lead by the Workflow Management Coalition [1] (WfMC), founded in 1993. This notion of workflow has extended to the scientific community where *scientific workflows* are used to support large-scale, complex scientific processes. They are designed to conduct experiments and prove scientific hypotheses by managing, analyzing, simulating and visualizing scientific data [26]. Therefore, even though both business and scientific workflows share the same basic concept, both have specific requirements and need separate consideration. This thesis focuses on scientific workflows and from now on, we will refer to them simply as workflows.

A workflow is defined by a set of computational tasks with dependencies between them. In scientific applications, it is common for the dependencies to represent a data flow from one task to another; the output data generated by one task becomes the input data for the next one. Figure 1.3 shows a sample workflow with nine tasks. In the example, task 1 produces three intermediate output data files that become the input for tasks 2, 3, and 4. Based on this, tasks 2, 3, and 4 cannot start their execution until task 1 has completed its execution and produced its output data.

These applications can be CPU, memory, or I/O intensive (or a combination of these), depending on the nature of the problem they are designed to solve. In a CPU intensive workflow, most tasks spend the majority of their time performing computations. Memory-bound workflows are those in which most of the tasks require high physical memory usage. Finally, I/O intensive workflows are composed of tasks that require and

[1] http://www.wfmc.org/

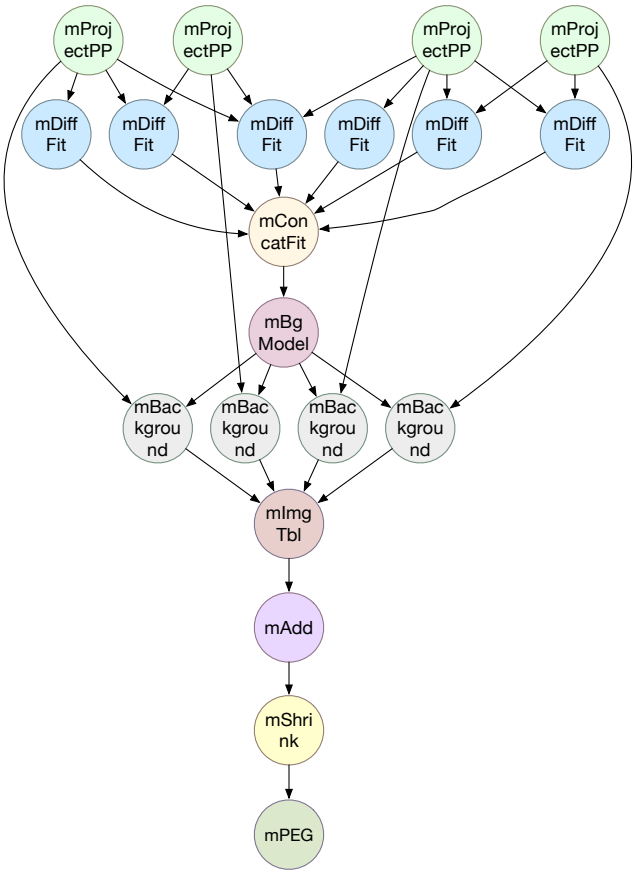


Figure 1.4: Sample Montage workflow.

produce large amounts of data and hence spend most of their time performing I/O operations.

Many scientific areas have embraced workflows as a mean to express complex computational problems that can be efficiently processed in distributed environments. For example, the Montage workflow [29] is an astronomy application characterized by being I/O intensive that is used to create custom mosaics of the sky based on a set of input images. It enables astronomers to generate a composite image of a region of the sky that is too large to be produced by astronomical cameras or that has been measured with different wavelengths and instruments. During the workflow execution, the geometry of the output image is calculated from that of the input images. Afterwards, the input data is re-projected so that they have the same spatial scale and rotation. This is followed by a standardization of the background of all images. Finally, all the processed input images

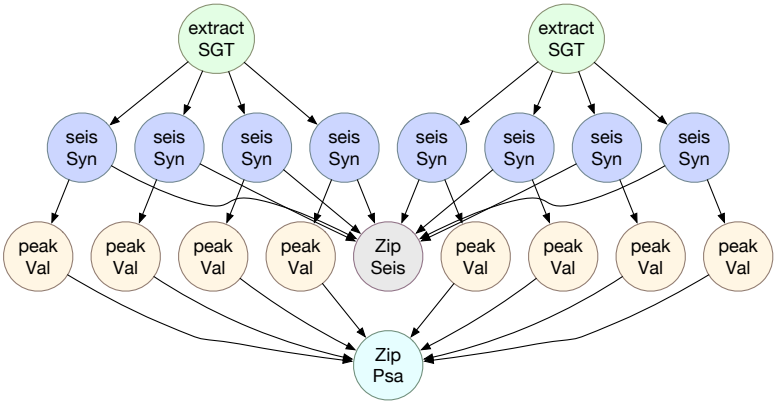


Figure 1.5: Sample CyberShake workflow.

are merged to create the final mosaic of the sky region. The structure of this workflow is shown in Figure 1.4.

Another example of a workflow is Cybershake [56], a data and memory intensive earthquake hazard characterization application used by the Southern California Earthquake Centre [18]. The workflow begins by generating Strain Green Tensors (SGTs) for a region of interest via a simulation. These SGT data are then used to generate synthetic seismograms for each predicted rupture followed by the creation of acceleration and probabilistic hazard curves for the given region. A sample CyberShake workflow is depicted in Figure 1.5.

Other examples include the Laser Interferometer Gravitational Wave Observatory (LIGO) [21], SIPHT [73] and Epigenomics [19] workflows. LIGO is a memory intensive application used in the physics field with the aim of detecting gravitational waves. In bioinformatics, SIPHT is used to automate the process of searching for sRNA encoding-genes for all bacterial replicons in the National Centre for Biotechnology Information [14] database. Also in the bioinformatics field, the Epigenomics workflow is a CPU intensive application that automates the execution of various genome sequencing operations. Figures 1.6, 1.7, and 1.8 shows the structure of these three scientific workflows respectively.

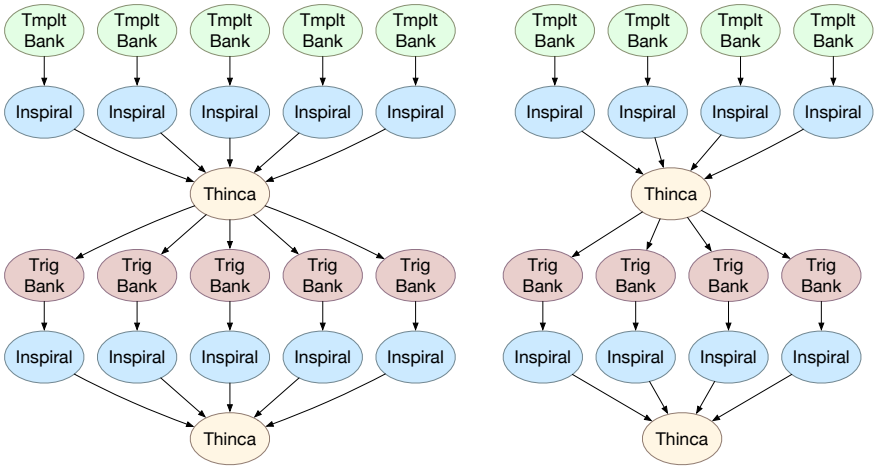


Figure 1.6: Sample LIGO workflow.

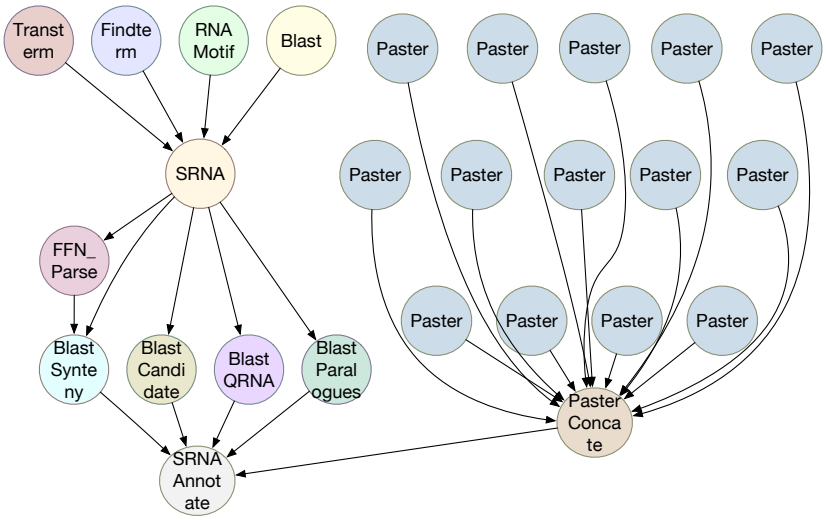


Figure 1.7: Sample SIPHT workflow.

The aforementioned applications are a good representation of scientific workflows as they are taken from different domains and together provide a broad overview of how workflow technologies are used to manage complex analyses. Each of the workflows have different topological structures all common in scientific workflows such as pipelines, data distribution, and data aggregation [30]. They also have varied data and computational characteristics, including CPU, I/O, and memory intensive tasks. Their full characterization is presented by Juve et al. [64].

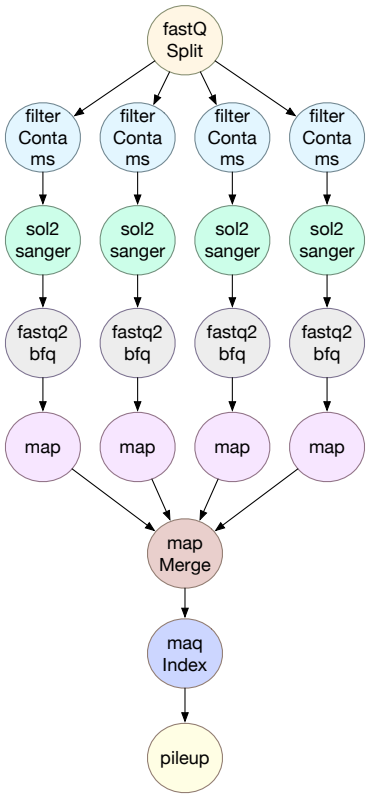


Figure 1.8: Sample Epigenomics workflow.

DAG Modeling

The scope of this thesis is limited to workflows modeled as Directed Acyclic Graphs (DAGs) which by definition, have no cycles or conditional dependencies. Although there are other models of computation that could be used to express and process scientific workflows such as best effort, superscalar, and streaming pipelines, this thesis focuses on DAGs as they are commonly used by the scientific and research community. For instance, workflow management systems such as Pegasus [44], Cloudbus WfMS [88], ASKALON [47], and DAGMan [40] support the execution of workflows modeled as DAGs. The work by Pautasso and Alonso [90] presents a detailed characterization of different models of computation that can be used for optimizing the performance of large scale scientific workflows.

Formally, a DAG representing a workflow application $W = (T, E)$ is composed of a set of tasks $T = \{t_1, t_2, \ldots, t_n\}$ and a set of directed edges E. An edge e_{ij} of the form

(t_i, t_j) exists if there is a data dependency between t_i and t_j, case in which t_i is said to be the parent task of t_j and t_j is said to be the child task of t_i. Based on this definition and as stated earlier, a child task cannot run until all of its parent tasks have completed their execution and its input data is available in the corresponding compute resource.

1.2 Problem Definition: Workflow Scheduling in IaaS Clouds

In general, the process of scheduling a workflow in a distributed system consists of assigning tasks to resources and orchestrating their execution so that the dependencies between them are preserved. The mapping is also done so that different user-defined QoS requirements are met. These QoS parameters determine the scheduling objectives and are generally defined in terms of performance metrics such as execution time, and non-functional requirements such as security and energy consumption.

This problem is NP-complete [105] in its general form and there are only three special cases that can be optimally solved within polynomial time. The first one is the scheduling of tree-structured graphs with uniform computation costs on an arbitrary number of processors [60]. The second one is the scheduling of arbitrary graphs with uniform computation costs on two processors [81] and the third one is the scheduling of interval-ordered graphs [49]. The problem addressed in this thesis does not fit any of these three scenarios, hence no optimal solution can be found in polynomial time.

To plan the execution of a workflow in a cloud environment, two sub problems need to be considered. The first one is known as *resource provisioning* and it consists of selecting and provisioning the compute resources that will be used to run the tasks. This means having heuristics in place that are capable of determining how many VMs to lease, their type, and when to start them and shut them down. The second sub problem is the actual *scheduling* or *task allocation* stage, in which each task is mapped onto the best-suited resource. The term *scheduling* is often used to refer to the combination of these two sub problems by authors developing algorithms targeting clouds and we follow the same pattern throughout the rest of this thesis.

Formally, the problem can be defined as follows. Let the different VM types offered

by an IaaS vendor be represented by the set $VMT = \{vmt_1, vmt_2, \ldots, vmt_n\}$. Let $R = \{r_1, r_2, \ldots, r_k\}$ be the set of concrete resources used for the workflow execution where $r_i = (vmt_j, st, et)$ represents a VM of type vmt_j leased at time st and shutdown at time et. Then the problem consists on finding the functions,

$$prov : VMT \mapsto R$$

$$sched : T \mapsto R$$

where *prov* determines the number and types of VMs to use as well as their leasing periods and *sched* maps each task $t_i \in T$ to a resource $r_i \in R$ so that the scheduling objectives are met.

1.2.1 Challenges

Scheduling algorithms need to address various challenges derived from the characteristics of the cloud resource model. In this section we discuss these challenges as well as the importance of considering them in order to leverage the flexibility and convenience offered by these environments.

Resource provisioning. The importance of addressing the resource provisioning problem as part of the scheduling strategy is demonstrated in several studies. The works by Gutierrez-Garcia and Sim [58], Michon et al. [83], and Villegas et al. [108] have demonstrated a dependency between both problems when scheduling Bags of Tasks (BoTs) in clouds while Frincu et al. [50] investigated the impact that resource provisioning has on the scheduling of various workflows in clouds. They all found a relationship between the two problems and concluded that the VM provisioning strategy has a direct impact on the cost and makespan achievable by the scheduling strategy.

Choosing the optimal configuration for the VM pool that will be used to run the workflow tasks is a challenging problem. Firstly, the combination of VMs needs to have the capacity to fulfill the scheduling objectives while considering their cost. If VMs are underprovisioned, then the scheduler will not be able to achieve the expected performance or

throughput. On the other hand, if VMs are over-provisioned, then the system's utilization will be low resulting in capacity wastage and unnecessary costs.

Secondly, provisioners need to dynamically scale in and out their resource pool. They need to decide when to add or remove VMs in response the application's demand as this may aid in improving the performance of the application, the overall system utilization, and reducing the total infrastructure cost.

Finally, provisioners have a wide range of options when selecting the VM type to use. Clouds offer VM instances with varying configurations in terms of compute, memory, storage, and networking performance. Different instance types are designed so that they are optimal for certain types of applications. For example, Amazon EC2 offers a family of compute-optimized instances designed to work best for applications requiring high compute power, a family of memory-optimized instances which have the lowest cost per GB of RAM and are best for memory intensive applications, and a storage-optimized instance family best suited for applications with specific disk I/O and storage requirements, among others [6]. Moreover, the price of VM instances varies with each configuration and it does not necessarily increase linearly with an increase in capacity. While this large selection of VM types offers applications enormous flexibility, it also challenges algorithms to be able to identify not only the best resource for a given task, but also the optimal combination of different instance types that allow for the user's QoS requirements to be met.

Performance variation and other sources of uncertainty. Characteristics such as multi-tenancy, virtualization, and the heterogeneity of non-virtualized hardware in clouds result in a variability in the performance of resources. For example, VMs deployed in cloud data centers do not exhibit a stable performance in terms of execution times [57,62,63,86, 99]. In fact, Schad et al. [99] report an overall CPU performance variability of 24% in the Amazon EC2 cloud. Performance variation is also observed in the network resources, with studies reporting a data transfer time variation of 19% in Amazon EC2 [99]. Additionally, if resources are located in different data centers or under different providers, they may be separated by public internet channels with unpredictable behavior. In case of

scientific workflows with large data dependencies, this may have a considerable impact in the workflow runtime. This indeterminism makes it extremely difficult for schedulers to estimate runtimes and make accurate scheduling decisions to fulfill QoS requirements.

A variability in performance has also been observed in other environments such as grids [28,85,106] and several performance estimation techniques have been developed as a result. However, the public and large scale nature of clouds makes this problem a more challenging one. For instance, to achieve a more accurate prediction of the runtime of a job in a specific VM, IaaS providers would have to allow access to information such as the type of host where the VM is deployed, its current load and utilization of resources, the overhead of the virtualization software, and network congestion, among others. Since access to this information is unlikely for users, algorithms, especially those dealing with constraints such as budget and deadline, need to acknowledge their limitations when estimating the performance of resources and have mechanisms in place that will allow them to recover from unexpected delays.

Other sources of uncertainty in cloud platforms are VM provisioning and deprovisioning delays. A VM is not ready for use immediately after its request. Instead, it takes time for it to be deployed on a physical host and booted; we refer to this time as the VM provisioning delay. Providers make no guarantees on the value of this delay and studies [78,86,99] have shown that it can be significant (in some providers more than others) and highly variable, making it difficult to predict or rely on an average measurement of its value. As an example, Osterman et al. [86] found the minimum resource acquisition time for the m1.large Amazon EC2 instance to be 50 seconds and the maximum to be 883 seconds; this demonstrates the potential variability that users may experience when launching a VM. As for the deprovisioning delay, it is defined as the time between the request to shutdown the VM and the actual time when the instance is released back to the provider and stops being charged. Once again, IaaS vendors make no guarantees on this time and it varies from provider to provider and VM type to VM type. However, Osterman et al. [86] found it has a lower variability and average value than the acquisition time and hence it may be slightly easier to predict and have a smaller impact on the execution of a workflow.

Utility-based pricing model. Schedulers planning the execution of workflows in clouds need to consider the cost of using the infrastructure. The use of VMs, as well as network and storage services, are charged for on a pay-per-use basis. Algorithms need to find a balance between performance, non-functional requirements, and cost. For example, some schedulers may be interested in a trade-off between execution time, energy consumption and cost. It is not only this trade-off that adds to the scheduling complexity but also the already mentioned difficulty in predicting the performance of resources, which translates in a difficulty in estimating the actual cost of using the chosen infrastructure.

Additionally, some IaaS providers offer a dynamic pricing scheme for VMs. In Amazon's EC2 terminology for example, these dynamically priced VMs are known as spot instances and their prices vary with time based on the market's supply and demand patterns [1]. Users can acquire VMs by bidding on them and their price is generally significantly lower than the *on-demand* price; however, they are subject to termination at any time if the spot price exceeds the biding price. Offerings such as this give users the opportunity to use VMs at significantly lower prices and scientific workflows can greatly benefit from this. But schedulers need to address challenges such as selecting a bid, determining the actions to take when a spot instance is terminated, and deciding when it is appropriate to use spot VMs versus statically priced, more reliable, ones.

1.3 Motivation

Many scientific areas have embraced workflows as a way of expressing complex computational problems that can be efficiently processed on distributed environments. They have become a prevailing mean to achieve significant scientific advances at an increased pace. An example is the success of the Advanced Laser Interferometer Gravitational-Wave Observatory (LIGO) [59] project in detecting gravitational waves. On September 2015, for the first time scientists were able to observe ripples in the fabric of space-time called gravitational waves, arriving at the earth from a large-scale, violent, event in the distant universe [20]. This confirms a major prediction of Albert Einsteins 1915 general theory of relativity and opens an unprecedented new window onto the cosmos [11]. The

project harnesses scientific workflows to process the vast amount of data collected from several interferometric gravitational wave detectors placed around the world. The efficient processing of these data enabled the identification of these waves and it "marks the beginning of a new era of gravitational wave astronomy where the possibilities for discovery are as rich and boundless as they have been with light-based astronomy" [12].

The ever-growing availability of data collected from increasingly powerful scientific instruments means workflows often analyze large quantities of data and are resource-intensive. This requires a distributed platform in order for meaningful results to be obtained in a reasonable amount of time. The latest distributed computing paradigm, cloud computing, offers several key benefits for the deployment of large-scale scientific workflows. Firstly, resources can be elastically provisioned and de-provisioned on-demand. This enables workflow management systems to use resources opportunistically based on the number and type of workflow tasks that need to be processed at a given point in time. In this way, the available resource pool can be scaled out and in as the workflow requirements change. This is a convenient feature for scientific workflows as common topological structures such as data distribution and aggregation [30] lead to significant changes in the parallelism of the workflow over time. This leads to situations in which adjusting the number of resources being used is highly desirable in order to increase performance and ensure the available resources are efficiently utilized.

Another benefit of deploying workflows in cloud computing environments derives from the fact that they are generally legacy applications that contain heterogeneous software components. Virtualization allows for the execution environment of these components to be easily customized. The operating system, software packages, directory structures, and input data files, among others, can all be tailored for a specific component and stored as a VM image. This image can then be easily used to deploy VMs capable of executing the software component they were designed for. Another advantage of using VM images for the deployment of workflow tasks is the fact that they enable scientific validation by supporting experiment reproducibility. Images can be stored and redeployed whenever an experiment needs to be reproduced as they enable the creation of the same exact environment used in previous experiments.

Scheduling algorithms are crucial in taking advantage of the benefits offered by clouds. Although the workflow scheduling problem has been widely studied over the years on platforms such as clusters and grids, research targeting the specific resource model offered by clouds is key in enabling the efficient automation of workflows on these platforms. Approaches developed for other parallel platforms could be applied to scheduling workflows on IaaS clouds, however, they would fail to leverage the on-demand access to *unlimited* resources, lead to unnecessary costs by not considering the IaaS cost model, and fail to capture the characteristics of clouds in terms of performance variation and sources of uncertainty.

Also, the majority of existing algorithms targeting clouds fail to embed key cloud computing features: they assume a fixed number of resources are available beforehand, they assume all VMs are of a single type, they do not consider the pay-as-you-go model, or they fail to consider different delays such as VM provisioning time and performance degradation exhibited by cloud resources. The solutions developed in this research will consider these key features, and more importantly, combine the scheduling and resource provisioning problems as a response to the dynamic and elastic nature of the cloud resources. This will allow schedulers to decide how many VMs to lease, of what type, and for how long so that the QoS requirements are met.

Finally, the main reason for running workflows in a distributed environment is to optimize their performance. The performance of a workflow execution is measured using a metric known as makespan, which is defined as the time elapsed between the execution of the first task until the completion of the last one. The utility-based pricing model offered by clouds also means that considering the cost of using the infrastructure is an important requirement for scheduling algorithms. For instance, the number of VMs leased, their type, and the amount of time they are used for, all have an impact on the total cost of running the workflow in the cloud. Consequently, this thesis studies algorithms that aim to fulfill a set of QoS requirements expressed in terms of these two important metrics, cost and makespan.

1.4 Thesis Contributions

Based on the previously defined scheduling problem and its challenges, this thesis makes the following **key contributions**:

- The identification and description of the challenges particular to cloud environments that scheduling algorithms must address;
- A taxonomy based on the scheduling, resource, and application models of state-of-the-art scientific workflow scheduling algorithms for clouds;
- A survey and detailed discussion of state-of-the-art algorithms within the scope of this thesis;
- A novel scheduling algorithm that leverages Particle Swarm Optimization to generate a workflow execution and resource provisioning plan;
- An algorithm based on the Unbounded Knapsack Problem capable of making resource provisioning and scheduling decisions while adapting to unexpected delays;
- A scalable algorithm that combines heuristics and two different Integer Programming models capable of scheduling workflows under budget constraints while considering fine-grained billing periods;
- A budget distribution strategy that allocates a portion of the budget to individual workflow tasks in order to facilitate resource provisioning decisions;
- The implementation of a cloud resource management module and its integration to an existing workflow management system.

1.5 Thesis Organization

The core chapters of this thesis are structured as shown in Figure 1.9 and are derived from several conference and journal papers published during the PhD candidature. The remainder of the thesis is organized as follows:

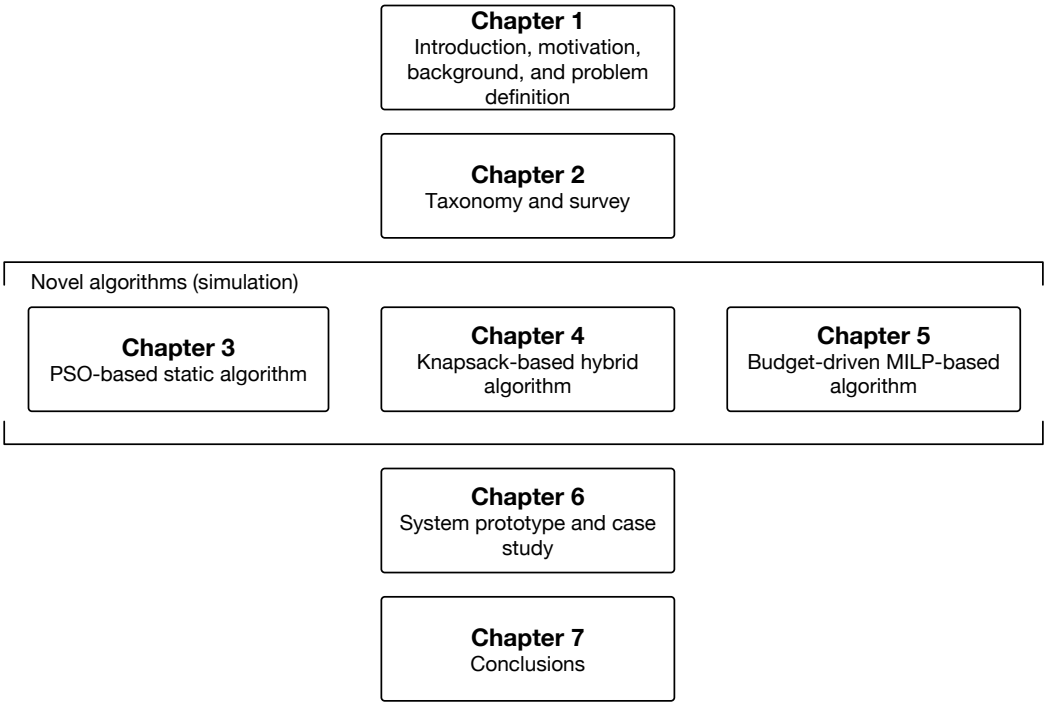


Figure 1.9: Thesis organization.

- Chapter 2 presents a comprehensive taxonomy and survey on scheduling algorithms for scientific workflows in IaaS clouds. This chapter is derived from:

  – **Maria A. Rodriguez** and Rajkumar Buyya. "A Taxonomy and Survey on Scheduling Algorithms for Scientific Workflows in IaaS Cloud Computing Environments." *Concurrency and Computation: Practice and Experience (CCPE)*, DOI:10.1002/cpe.4041, 2016.

- Chapter 3 presents a static cost-minimization, deadline-constrained heuristic for scheduling a scientific workflow application. It considers fundamental features of IaaS providers such as the dynamic provisioning and heterogeneity of unlimited

computing resources as well as VM performance variation. Both resource provisioning and scheduling are merged and modeled as an optimization problem. Particle Swarm Optimization (PSO) is then used to solve such problem. This chapter is derived from [96]:

  - **Maria A. Rodriguez** and Rajkumar Buyya. "Deadline Based Resource Provisioning and Scheduling Algorithm for Scientific Workflows on Clouds." *IEEE Transactions on Cloud Computing*, Volume 2, Issue 2, Pages: 222-235, 2014.

- Chapter 4 presents an algorithm modeled as a variation of the Unbounded Knapsack Problem. It is dynamic to a certain extent in order to be able to respond to the dynamics of the cloud infrastructure and it has a static component that allows it to generate high-quality solutions that meet a user-defined deadline and minimize the overall cost of the used infrastructure. This chapter is derived from [97]:

  - **Maria A. Rodriguez** and Rajkumar Buyya. "A Responsive Knapsack-based Algorithm for Resource Provisioning and Scheduling of Scientific Workflows in Clouds." *In Proceedings of the 44th International Conference on Parallel Processing (ICPP)*, Pages 839-848, 2015.

- Chapter 5 describes a budget-driven algorithm whose objective is to optimize the way in which the budget is spent so that the makespan of the workflow is minimized. It includes a budget distribution strategy that guides the individual expenditure on tasks and makes dynamic resource provisioning and scheduling decisions to adapt to changes in the environment. Also, to improve the quality of the optimization decisions made, two different mathematical models are proposed to estimate the optimal resource capacity for parallel tasks derived from data distribution structures. This chapter is derived from:

  - **Maria A. Rodriguez** and Rajkumar Buyya. "Budget-Driven Resource Provisioning and Scheduling of Scientific Workflow in IaaS Clouds with Fine-Grained Billing Periods." *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2016 (under second review).

- Chapter 6 describes the functionality of an existing workflow management system and the cloud-based extensions made to it in order to enable the dynamic provisioning of cloud resources. A case study using the Montage workflow and the algorithm presented in Chapter 4 is also presented. This chapter is derived from:
  - **Maria A. Rodriguez** and Rajkumar Buyya. "Scientific Workflow Management System for Clouds" *Software Architecture for Big Data and the Cloud*, Elsevier - Morgan Kaufmann, 2017 (in press).
- Chapter 7 concludes the thesis, summarizes its findings, and provides directions for future work.

Chapter 2

A Taxonomy of Scheduling Algorithms for Scientific Workflows in IaaS Clouds

In this chapter, we propose a taxonomy that characterizes and classifies scheduling algorithms based on the scheduling model they adopt as well as the resource and application models they consider. State-of-the-art algorithms are surveyed; they are discussed and classified according to the proposed taxonomy. In this way, we not only provide a comprehensive understanding of existing literature and highlight the similarities and differences of existing algorithms but also provide an insight into future directions and open issues.

2.1 Introduction

The scheduling of workflow tasks in distributed platforms has been widely studied over the years. Researchers have developed algorithms tailored for different environments; from homogeneous clusters with a limited set of resources, to large-scale community grids, to the most recent paradigm, utility-based, heterogeneous, and resource-abundant cloud computing. This chapter focuses on the latter case, it studies algorithms developed to orchestrate the execution of scientific workflow tasks exclusively in IaaS cloud computing environments. The aim is to identify how different algorithms deal with the particular features of the cloud resource model and how they benefit from the ease of access, scalability, and flexibility of these environments.

This chapter is derived from: **Maria A. Rodriguez** and Rajkumar Buyya. "A Taxonomy and Survey on Scheduling Algorithms for Scientific Workflows in IaaS Cloud Computing Environments." *Concurrency and Computation: Practice and Experience (CCPE)*, DOI:10.1002/cpe.4041, 2016.

In this chapter different characteristics of existing algorithms are analyzed. In particular, the scheduling, application, and resource models are studied. Since extensive research has been done on the scheduling field in general, widely accepted and accurate classifications already exist for some of these features. We extend and complement with a cloud-focused discussion those that are of particular importance to the studied problem. Some of these include the dynamicity of the scheduling and provisioning decisions, the scheduling objectives, and the optimization strategy. The application model is studied form the workflow multiplicity point of view, that is, the number and type of workflows that algorithms are capable of processing. Finally, classifications for the resource model are based on different cloud features such as storage and data transfer costs, pricing models, VM delays, data center deployment model, and VM heterogeneity among others.

2.2 Taxonomy

The scope of this survey is limited to algorithms developed to schedule workflows exclusively in public IaaS clouds. As a result, only those that consider a utility-based pricing model and address the VM provisioning problem are studied. Other scope-limiting features are derived from the application model and all the surveyed algorithms consider workflows with the following characteristics. Firstly, they are modeled as DAGs with no cycles or conditional dependencies. Secondly, their execution requires a set of input data, generates intermediate temporary data, and produces a result in terms of an output data set. Thirdly, tasks are assumed to be non-parallel in the number of VMs they require for their execution. Finally, the structure of the workflows is assumed to be static, that is, tasks or dependencies cannot be updated, added, or removed at runtime.

This section is limited to providing an explanation of each taxonomy classification, examples and references to algorithms for each class are presented later in Section 2.3.

2.2.1 Application Model Taxonomy

All algorithms included in this survey share most of the application model features. They differ however in terms of their ability to schedule either a single or multiple workflows.

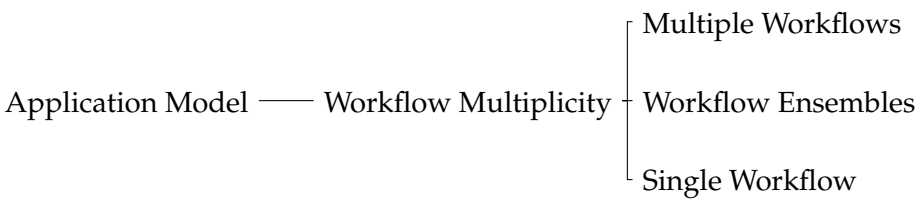


Figure 2.1: Application model taxonomy.

Workflow multiplicity

Algorithms can be designed to schedule a single instance of a workflow, multiple instances of the same workflow or multiple workflows. Based on this we identify three types of scheduling processes from the workflow multiplicity perspective.

Single workflow. Algorithms in this class are designed to optimize the schedule of a single workflow. This is the traditional model used in grids and clusters and is still the most common one in cloud computing. It assumes the scheduler manages the execution of workflows sequentially and independently. In this way, the scheduling algorithm can focus on optimizing cost and meeting the QoS requirements for a single user and a single DAG.

Workflow ensembles. Many scientific applications [43, 74, 109] are composed of more than one workflow instance. These interrelated workflows are known as ensembles and are grouped together because their combined execution produces a desired output [76]. In general, the workflows in an ensemble have a similar structure but differ in size and input data. Scheduling algorithms in this category focus on executing every workflow on the ensemble using the available resources. Policies need to be aware of the fact that the QoS requirements are meant for multiple workflows and not just a single one. For example, all 100 workflows in an ensemble with a 1-hour deadline need to be completed before this time limit. Based on this, algorithms are generally concerned with the amount of work (number of executed workflows) completed and tend to include this in the scheduling objectives. Another characteristic of ensembles is that the number instances is gen-

erally known in advance and hence the scheduling strategy can use this when planning the execution of tasks.

Multiple workflows. This category is similar to the workflow ensembles one, but differs from it in the fact that the workflows being scheduled are not necessarily related to each other and might vary in structure, size, input data, application, etc. More importantly, the number and type of workflows are not known in advance and therefore the scheduling is viewed as a dynamic process in which the workload is constantly changing and workflows with varying configurations are continuously arriving for execution. Yet another difference is that each workflow instance has its own, independent, QoS requirements. Algorithms in this category need to deal with the dynamic nature of the problem and need to efficiently use the resources in order to meet the QoS requirements of as many workflows as possible.

An example of a system addressing these issues is proposed by Tolosana-Calasanz et al. [103]. They develop a workflow system for the enforcement of QoS of multiple scientific workflow instances over a shared infrastructure such as a cloud computing environment. However their proposal uses a superscalar model of computation for the specification of the workflows as opposed to the DAG model considered in this survey.

2.2.2 Scheduling Model Taxonomy

There have been extensive studies on the classification of scheduling algorithms on parallel systems. For instance, Casavant and Kuhl [38] proposed a taxonomy of scheduling algorithms in general-purpose distributed computing systems, Kwok and Ahmad [67] developed a taxonomy for static scheduling algorithms for allocating directed task graphs to multiprocessors, while Yu et al. [118] studied the workflow scheduling problem in grid environments. Since these scheduling models still apply to the surveyed algorithms, in this section we identify, and in some cases extend, those characteristics that are most relevant to our problem. Aside from a brief introduction to their general definition, we aim to keep the discussion of each category as relevant to the scheduling problem addressed in this work as possible. Figure 2.2 illustrates the features selected to study the scheduling

model.

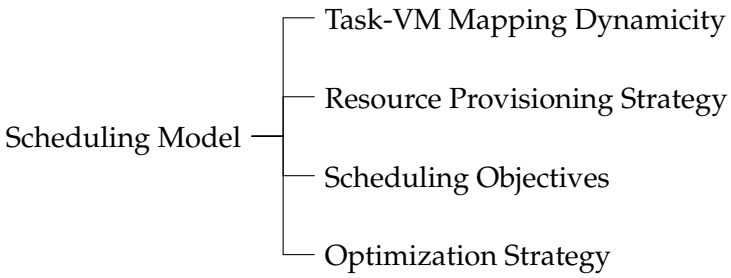


Figure 2.2: Scheduling model taxonomy.

Task-VM Mapping Dynamicity

Following the taxonomy of scheduling for general purpose distributed systems presented by Casavant and Kuhl [38], workflow scheduling algorithms can be classified as either static or dynamic. This classification is common knowledge to researchers studying any form of scheduling and hence it provides readers with a quick understanding of key high-level characteristics of the surveyed algorithms. Furthermore, it is highly relevant for cloud environments as it determines the degree of adaptability that the algorithms have to an inherently dynamic environment. In addition to these two classes, we identify a third hybrid one, in which algorithms combine both approaches to find a trade-off between the advantages offered by each of them.

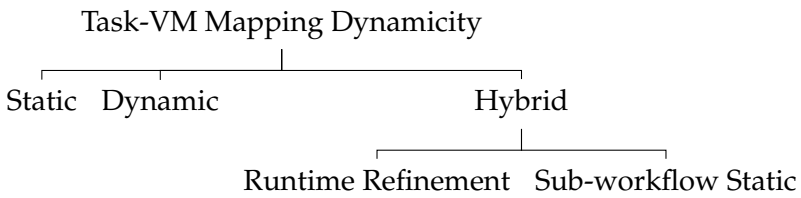


Figure 2.3: Types of task-VM mapping dynamicity.

Static. These are algorithms in which the task to VM mapping is produced in advance and executed once. Such plan is not altered during runtime and the workflow engine must adhere to it no matter what the status of the resources and the tasks is. This rigidity

does not allow them to adapt to changes in the underlying platform and makes them extremely sensitive to execution delays and inaccurate task runtime estimation; a slight miscalculation might lead to the actual execution failing to meet the user's QoS requirements. This is especially true for workflows due to the domino effect the delay in the runtime of one task will have in the runtime of its descendants. Some static algorithms have strategies in place to improve their adaptability to the uncertainties of cloud environments. These include more sophisticated or conservative runtime prediction strategies, probabilistic QoS guarantees, and resource performance variability models. The main advantage of static schedulers is their ability to generate high-quality schedules by using global, workflow-level, optimization techniques and to compare different solutions before choosing the best suited one.

Dynamic. These algorithms make task to VM assignment decisions at runtime. These decisions are based on the current state of the system and the workflow execution. For our scheduling scenario, we define dynamic algorithms to be those that make scheduling decisions for a single workflow task, at runtime, once it is ready for execution. This allows them to adapt to changes in the environment so that the scheduling objectives can still be met even with high failure rates, unaccounted delays, and poor estimates. This adaptability is their main advantage when it comes to cloud environments, however, it also has negative implications in terms of the quality of the solutions they produce. Their limited, task-level, view of the problem hurts their ability to find high-quality schedules from the optimization point of view.

Hybrid. Some algorithms aim to find a trade-off between the adaptability of dynamic algorithms and the performance of static ones. We identify two main approaches in this category, namely runtime refinement and sub-workflow static. In *runtime refinement*, algorithms first device a static assignment of tasks before runtime. This assignment is not rigid as it may change during the execution of the workflow based on the current status of the system. For example, tasks may be assigned to faster VMs or they may be mapped onto a different resource to increase the utilization of resources. Algorithms may choose to update the mapping of a single task or to update the entire schedule every cycle. When

updating a single task, decisions are made fast but their impact on the rest of the workflow execution is unknown. When re-computing the schedule for all of the remaining tasks, the initial static heuristic is used every scheduling cycle, resulting in high time and computational overheads.

The *sub-workflow static* approach consists on making static decisions for a group of tasks dynamically. That is, every scheduling cycle, a subset of tasks is statically scheduled to resources based on the current system conditions. This allows the algorithm to make better optimization decisions while enhancing its adaptability. The main disadvantage is that statically assigned tasks, although to a lesser extent, are still subject to the effects of unexpected delays. To mitigate this, algorithms may have to implement further rescheduling or refinement strategies for this subset of tasks.

Resource Provisioning Strategy

As with the task to VM mapping, algorithms may also adopt a static or dynamic resource provisioning approach. We define *static* resource provisioners to be those that make all of the decisions regarding the VM pool configuration before the execution of the workflow. *Dynamic* provisioners on the other hand, make all of the decisions or refine initial ones at runtime, selecting which VMs to keep active, which ones to lease, and which ones to release as the workflow execution progresses. Figure 2.4 illustrates different types of static and dynamic resource provisioning strategies.

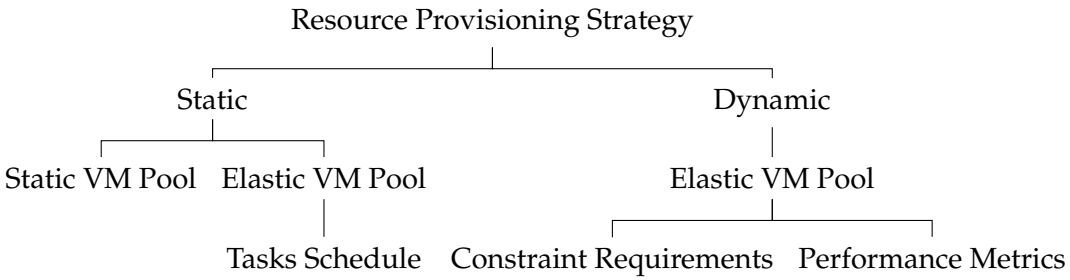


Figure 2.4: Types of resource provisioning strategies

Static VM Pool. This strategy may be used by algorithms adopting a static resource provisioning approach. Once the VM pool is determined, the resources are leased and they remain active throughout the execution of the workflow. When the application finishes running, the resources are released back to the provider. These algorithms are concerned with estimating the resource capacity needed to achieve the scheduling objectives. The advantage is that once the resource provisioning decision is made, the algorithm can focus solely on the task to VM allocation. The effects of VM provisioning and deprovisioning delays is highly amortized and becomes much easier to manage. However, this model does no take advantage of the elasticity of resources and ignores the cloud billing model. This may result in schedules that fail to meet the QoS requirements due to poor estimates and that are not cost-efficient as even billing periods in which VMs are idle are being charged for.

Elastic VM Pool. This strategy is suitable for algorithms adopting either a static or dynamic resource provisioning approach. This method allows algorithms to update the number and type of VMs being used to schedule tasks as the execution of the workflow progresses. Some algorithms make elastic decisions based on their cost-awareness and the *constraint requirements* of tasks. For instance, a new VM can be provisioned so that the task being scheduled can finish before its deadline while idle VMs can be shutdown to save cost. Another way of achieving this is by periodically estimating the resource capacity needed by tasks to meet the application's constraints and adjust the VM pool accordingly. Other algorithms make scaling decisions based on *performance metrics* such as the overall VM utilization and throughput of tasks. For example, new VMs may be provisioned if the budget allows for it and the utilization rises above a specified threshold or if the number of tasks processed by second decreases below a specified limit. Finally, static algorithms that use elastic VM pools do so by determining the leasing periods of VMs when generating the static schedule. These leasing periods are bounded by the estimated start time of the first task assigned to a VM and the estimated finish time of the last task assigned to it.

Scheduling objectives

Being cost-aware is the common denominator of all the surveyed algorithms. In addition to this objective, most algorithms also consider some sort of performance metric such as the total execution time or the number of workflows executed by the system. Furthermore, some state-of-the-art algorithms also incorporate energy consumption, reliability and security as part of their objectives. The scheduling objectives included in this taxonomy are derived from those being addressed by the reviewed literature presented in Section 2.3.

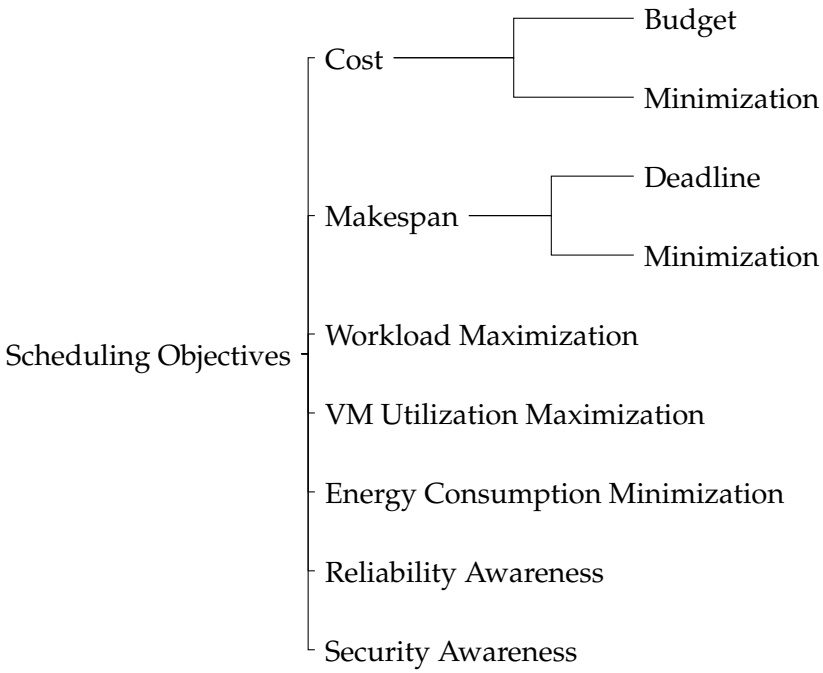


Figure 2.5: Types of scheduling objectives.

Cost. Algorithms designed for cloud platforms need to consider the cost of leasing the infrastructure. If they fail to do so, the cost of renting VMs, transferring data, and using the cloud storage can be considerably high. This objective is included in algorithms by either trying to minimize its value or by having a cap on the amount of money spent on resources (i.e., budget). All of the algorithms studied balance cost with other objectives related to performance or non-functional requirements such as security, reliability, and

energy consumption. For instance, the most commonly addressed QoS requirement is minimizing the total cost while meeting a user-defined deadline constraint.

Makespan. Most of the surveyed algorithms are concerned with the time it takes to run the workflow, or makespan. As with cost, it is included as part of the scheduling objectives by either trying to minimize its value, or by defining a time limit, or deadline, for the execution of the workflow.

Workload Maximization. Algorithms developed to schedule ensembles generally aim to maximize the amount of work done, that is, the number of workflows executed. This objective is always paired with constraints such as budget or deadline and hence, strategies in this category aim at executing as many workflows as possible with the given money or within the specified time frame.

VM Utilization Maximization. Most algorithms are indirectly addressing this objective by being cost-aware. Idle time slots in leased VMs are deemed as a waste of money as they were paid for but not utilized and as a result, algorithms try to avoid them in their schedules. However, it is not uncommon for this unused time slots to arise from a workflow execution, mainly due to the dependencies between tasks and performance requirements. Some algorithms are directly concerned with minimizing these idle time slots and maximizing the utilization of resources, which has benefits for users in terms of cost, and for providers in terms of energy consumption, profit, and more efficient usage of resources.

Energy Consumption Minimization. Individuals, organizations and governments worldwide have developed an increased concern to reduce carbon footprints in order to lessen the impact on the environment. Although not unique to cloud computing, this concern has also attracted attention in this field. A few algorithms that are aware of the energy consumed by the workflow execution have been recently developed. They consider a combination of contradicting scheduling goals as they try to find a trade-off between energy consumption, performance and cost. Furthermore, virtualization and the lack of

control and knowledge of the physical infrastructure limit their capabilities and introduce further complexity into the problem.

Reliability Awareness. Algorithms considering reliability as part of their objectives have mechanisms in place to ensure the workflow execution is completed within the users' QoS constraints even if resource or task failures occur. Algorithms targeting unreliable VM instances that are failure prone (e.g., Amazon EC2 spot instances) need to have policies addressing reliability in place. Some common approaches include replicating critical tasks and relying on checkpointing to reschedule failed tasks. However, algorithms need to be mindful of the additional costs associated with task replication as well as with the storage of data for checkpointing purposes. Furthermore, it is important to consider that most scientific workflows are legacy applications that are not enabled with checkpointing mechanisms and hence, relying on this assumption might be unrealistic.

Security Awareness. Some scientific applications may require that the input or output data are handled in a secure manner. Even more, some tasks may be composed of sensitive computations that need to be kept secure. Algorithms concerned with these security issues may leverage different security services offered by IaaS providers. They may handle data securely by deeming it *immovable* [122], or may manage sensitive tasks and data in such a way that either resources or providers with a higher security ranking are used to execute and store them. Considering these security measures has an impact when making scheduling decisions as tasks may have to be moved close to immovable data sets and the overhead of using additional security services may need to be included in the time and cost estimates.

Optimization Strategy

Scheduling algorithms can be classified as optimal or sub-optimal following the definition of Casavant and Kuhl [38]. As a result of the NP-completeness [105] of the discussed problem, finding *optimal* solutions is computationally expensive even for small-scale versions of the problem, rendering this strategy impractical in most situations. In addition,

the optimality of the solution is restricted by the assumptions made by the scheduler regarding the state of the system as well as the resource requirements and computational characteristics of tasks. Based on this, the overhead of finding the optimal solution for large-scale workflows that will be executed under performance variability may be unjustifiable. For small workflows however, with coarse-grained tasks that are computationally intensive and are expected to run for long periods of time, this strategy may be more attractive.

There are multiple methods that can be used to find optimal schedules [38]. In particular, Casavant and Kuhl [38] identify four strategies for the general multi-processor scheduling problem: solution space enumeration and search, graph theoretic, mathematical programming, and queuing theoretic. Most relevant to our problem are solution space enumeration and mathematical models; in the surveyed algorithms, Mixed Integer Linear Programs (MILPs) have been used to obtain workflow-level optimizations [52]. The same strategy and dynamic programming have been used to find optimal schedules for a subset of the workflow tasks or simplified versions of the problem [75,97], although this sub-global optimization does not lead to an optimal solution.

The vast majority of the algorithms focus on generating approximate or near-optimal solutions. For the *sub-optimal* category, we identify three different methods used by the studied algorithms. The first two are heuristic and meta-heuristic approaches as defined by Yu et al. [118]. We add to this model a third hybrid category to include algorithms combining different strategies.

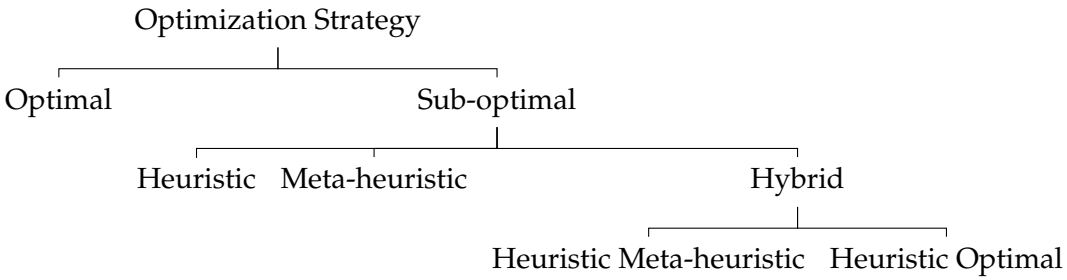


Figure 2.6: Types of optimization strategies.

Heuristics. In general, a heuristic is a set of rules that aim to find a solution for a particular problem [102]. Such rules are specific to the problem and are designed so that an approximate solution is found in an acceptable time frame. For the scheduling scenario discussed here, a heuristic approach uses the knowledge about the characteristics of the cloud as well as the workflow application in order to find a schedule that meets the user's QoS requirements. The main advantage of heuristic based scheduling algorithms is their efficiency in terms of performance; they tend to find satisfactory solutions in an adequate lapse of time. They are also easier to implement and more predictable than meta-heuristic based methods.

Meta-heuristics. While heuristics are designed to work best on a specific problem, meta-heuristics are general-purpose algorithms designed to solve optimization problems [102]. They are higher level strategies that apply problem specific heuristics in order to find a near-optimal solution to a problem. When compared to heuristic-based algorithms, meta-heuristic approaches are generally more computationally intensive and take longer to run; however, they also tend to find more desirable schedules as they explore different solutions using a guided search. Using meta-heuristics to solve the workflow scheduling problem in clouds involves challenges such as modeling a theoretically unbound number of resources, defining operations to avoid exploring invalid solutions (e.g., data dependency violations) to facilitate convergence, and pruning the search space by using heuristics based on the cloud resource model.

Hybrid. Algorithms using a hybrid approach, may use meta-heuristic methods to optimize the schedule of a group of workflow tasks. Another option is to find optimal solutions for simplified and/or smaller versions of the problem and combine them using heuristics. In this way, algorithms may be able to make better optimization decisions than heuristic-based methods while reducing the computational time by considering a smaller problem space.

2.2.3 Resource Model Taxonomy

In this section a taxonomy is presented based on the resource model considerations and assumptions made by algorithms. These design decisions range from high level ones such as the number of IaaS providers modeled to lower level ones concerned with the services offered by providers, such as the VM pricing model and the cost of data transfers. The characteristics of the resource model considered in this survey are illustrated in Figure 2.7.

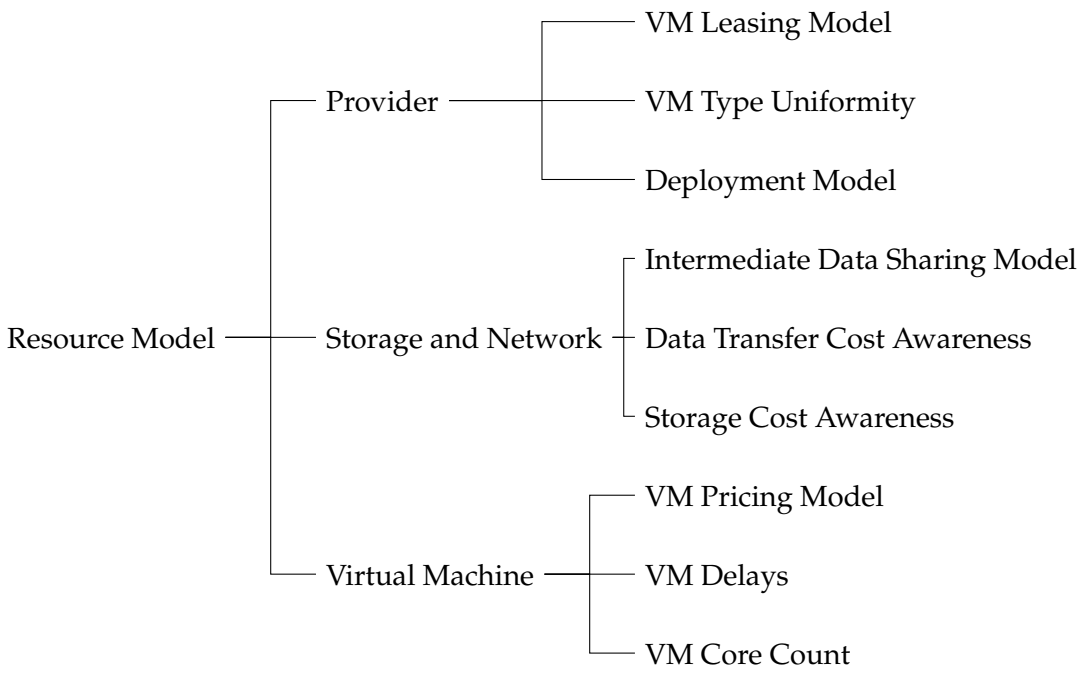


Figure 2.7: Resource model taxonomy.

VM Leasing Model

This feature is concerned with algorithms assuming providers offer either a bounded or an unbounded number of VMs available to lease for a given user (Figure 2.8).

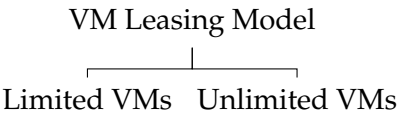


Figure 2.8: Types of VM leasing models.

Limited. These algorithms assume providers have a cap on the number of VMs a user is allowed to lease. In this way, the resource provisioning problem is somehow simplified and is similar to scheduling with a limited number of processors. However, provisioning decisions are still important due to the overhead and cost associated with leasing VMs.

Unlimited. Algorithms assume they have access to a virtually unlimited number of VMs. There is no restriction on the number of VMs the provisioner can lease and hence the algorithm needs to find efficient policies to manage this abundance of resources efficiently.

VM Type Uniformity

Algorithms may assume resource homogeneity by leasing VMs of a single type or may use heterogeneous VMs with different configurations based on their scheduling objectives (Figure 2.9).

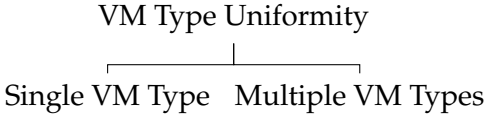


Figure 2.9: Types of VM uniformity.

Single VM Type. In this category, VM instances leased from the IaaS provider are limited to a single type. This assumption is in most cases made to simplify the scheduling process and the decision of which VM type to use is made without consideration of the workflow and tasks characteristics. This may potentially have a negative impact on the

outcome of the algorithm and as a result, this strategy fails to take full advantage of the heterogeneous nature of cloud resources.

Multiple VM Types. These algorithms acknowledge the fact that IaaS clouds offer different types of VMs. They have policies to select the most appropriate types depending on the nature of the workflow, the characteristics of tasks, and the scheduling objectives. This enables the algorithms to use different VM configurations and efficiently schedule applications with different requirements and characteristics.

Deployment Model

Another way of classifying algorithms is based on the number of data centers and public cloud providers they lease resources from as shown in Figure 2.10.

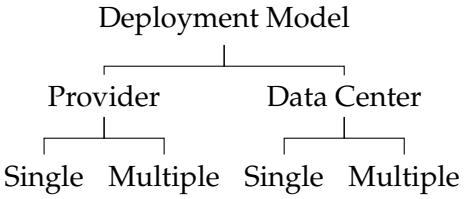


Figure 2.10: Types of provider and data center deployment models.

Single Provider. Algorithms in this category consider a single public cloud provider offering infrastructure on a pay-per-use basis. In general, they do not need to consider the cost of transferring data in or out of the cloud as this cost for input and output data sets is considered constant based on the workflow being scheduled.

Multiple Providers. This deployment model allows algorithms to schedule tasks onto resources owned by different cloud providers. Each provider has its own product offerings, SLAs, and pricing policies and it is up to the scheduler to select the best suited one. Algorithms should consider the cost and time of transferring data between providers as these are not negligible. This model may be beneficial for workflows with special security requirements or large data sets distributed geographically. A potential benefit of

these inter-cloud environments is taking advantage of the different billing period granularities offered by different providers. Smaller tasks may be mapped to VMs with finer billing periods such as one minute while larger ones to those with coarser-grained periods. While inter-clouds could be beneficial for the scheduling problem by providing a wider range of services with different prices and characteristics, the lack of standardization, network delays, and data transfer costs, pose real challenges in this area.

Single Data Centre. Often, algorithms choose to provision VMs in a single data center, or in the terminology of Amazon EC2, a single availability zone. This deployment model is sufficient for many application scenarios as it is unlikely that the number of VMs required for the execution of the workflow will exceed the data centers capacity. It also offers two key advantages in terms of data transfers. The first one is reduced latency and faster transfer times and the second one is potential cost savings as many providers do not charge for transfers made within a data center.

Multiple Data Centers. Using a resource pool composed of VMs deployed in different data centers belonging to the same provider is another option for algorithms. This choice is more suited for applications with geographically distributed input data. In this way, VMs can be deployed in different data centers based on the location of the data to reduce data transfer times. Other workflows that benefit from this model are those with sensitive data sets that have specific location requirements due to security or governmental regulations. Finally, algorithms under this model need to be aware of the cost of transferring data between different data centers as most providers charge for this service.

Intermediate Data Sharing Model

Workflows process data in the form of files. The way in which these files are shared has an impact on the performance of scheduling algorithms as they have an effect on metrics such as cost and makespan. A common approach is to assume a peer-to-peer (P2P) model while another technique is to use a global shared storage system as a file repository (Figure 2.11).

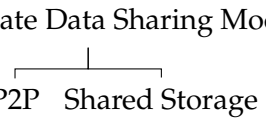


Figure 2.11: Types of intermediate data sharing models

P2P. These algorithms assume files are transferred directly from the VM running the parent task to the VM running the child task. This means tasks communicate in a synchronous manner and hence VMs must be kept running until all of the child tasks have received the corresponding data. This may result in higher costs as the lease time of VMs is extended. Additionally, the failure of a VM would result in data loss that can potentially require the re-execution of several tasks in order to recover. The main advantage of this approach is its scalability and lack of bottlenecks.

Data Transfer Cost Awareness

IaaS providers have different pricing schemes for different types of data transfers, depending if the data is being transferred into, within, or out of their facilities. Transferring inbound data is generally free and hence this cost is ignored by all of the studied algorithms. On the contrary, transferring data out of the cloud provider is generally expensive. Algorithms that schedule workflows across multiple providers are the only ones that need to be concerned with this cost, as data may need to be transferred between resources belonging to different providers. As for transferring data within the facilities of a single provider, it is common for transfers to be free if they are done within the same data center and to be charged if they are between different data centers. Hence, those algorithms considering multiple data centers in their resource model should include this cost in their estimations. Finally, regarding access to storage services, most providers such as Amazon S3 [2], Google Cloud Storage [8], and Rackspace Block Storage [16], do not charge for data transfers in and out of the storage system and hence this value can be ignored by algorithms making use of these facilities.

Storage Cost Awareness

Data storage is charged based on the amount of data being stored. Some providers have additional fees based on the number and type of operations performed on the storage system (i.e., GET, PUT, DELETE). This cost is only relevant if cloud storage services are used, and even in such cases, it is ignored in many models mainly due to the fact that the amount of data used and produced by a workflow is constant and independent of the scheduling algorithm. However, some algorithms do acknowledge this cost and generally estimate it based on the data size and a fixed price per data unit.

Storage Cost Awareness

Data storage is charged based on the amount of data being stored. Some providers have additional fees based on the number and type of operations performed on the storage system (i.e., GET, PUT, DELETE). This cost is only relevant if cloud storage services are used, and even in such cases, it is ignored in many models mainly due to the fact that the amount of data used and produced by a workflow is constant and independent of the scheduling algorithm. However, some algorithms do acknowledge this cost and estimate it based on the data size and a fixed price per data unit, achieving a good approximation to what the actual storage cost would be.

VM Pricing Model

As depicted in Figure 2.12, we identify four different pricing models considered by the surveyed algorithms that are relevant to our discussion: dynamic, static, subscription-based and time unit.

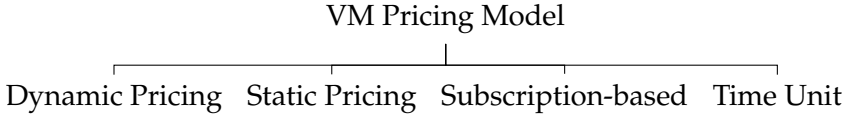


Figure 2.12: Types of VM pricing models.

Dynamic Pricing. The price of instances following this model varies over time and is determined by the market dynamics of supply and demand. Generally, users acquire dynamically priced VMs by means of auctions or negotiations. In these auctions, users request a VM by revealing the maximum amount of money they are willing to pay for it, providers then decide to accept or reject the request based on the current market conditions. These type of instances generally offer users an economical advantage over statically priced ones. An example of VMs following this pricing model are Amazon EC2 Spot Instances [1]. The spot market allows users to bid on VMs and run them whenever their bidding prices exceed the current market (i.e., spot) price. Through this model, users can lease instances at considerably lower prices but are subject to the termination of VMs when the market price becomes higher than the bidding one. Hence, tasks running on spot instances need to be either interruption-tolerant or scheduling algorithms need to implement a recovery or fault tolerant mechanism. VMs with dynamic pricing are often used opportunistically by scheduling algorithms [93, 124] in conjunction with statically priced ones in order to reduce the overall cost of executing the workflow.

Static Pricing. The static pricing model is the conventional cloud pricing model and is offered by most providers. VMs are priced per billing period and any partial utilization is charged as a full-period utilization. An example of a provider offering instances under this pricing model is Google Compute Engine [9]. All VM types are charged a minimum of 10 minutes and after this, they are charged in 1 minute intervals, rounded up to the nearest minute [10]. For example, if a VM is used for 3 minutes, it will be billed for 10 minutes of usage and if it is used for 12.4 minutes, it will be billed for 13.

Subscription-based. Under this model, instances are reserved for a longer time frame, usually monthly or yearly. Generally, payment is made upfront and is significantly lower when compared to static pricing. VMs are billed at the discounted rate for every billing period (e.g., hour) in the reserved term regardless of usage. For the cloud workflow scheduling problem, this pricing model means that schedulers need to use a fixed set of VMs with fixed configurations to execute the tasks. This transforms the problem, at least from the resource provisioning point of view, into one being designed for a platform

with limited availability of resources such as a grid or cluster. An example of a provider offering subscription-based instances is Cloudsigma [4]. It offers unbundled resources such as CPU, RAM and storage (users can specify exactly how much of each resource they need without having to select from a predefined bundle) that can be leased for either 1, 3 or 6 months or 1, 2 or 3 years. They offer a fixed discount based on the leasing period selected, the longer the leasing period, the larger the discount.

Time Unit. Algorithms in this category assume VMs are charged per time unit. Under this model, there is no resource wastage or additional costs due to unused time units in billing periods. Hence, the scheduling is simplified as there is no need to use idle time slots of leased VMs as the cost of using the resources is the actual exact time they are used for. This may be considered as an unrealistic approach as there are no known cloud providers offering this level of granularity and flexibility yet, however, some algorithms do assume this model for simplicity. Additionally, there is the possibility of new pricing models being offered by providers or emerging from existing ones, as is pointed out by Arabnejad et al. [25], a group of users may for example rent a set of VMs on a subscription-based basis, share them, and price their use on a time unit basis.

VM Delays

This category is concerned with the awareness of algorithms of the VM provisioning and deprovisioning delays (Figure 2.13).

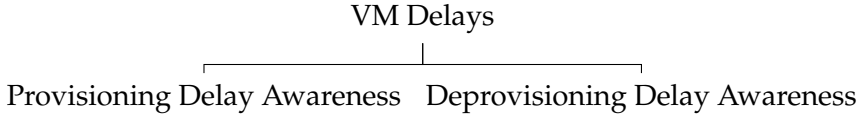


Figure 2.13: Types of VM delays

VM Provisioning Delay. As already stated in section 1.2.1, VM provisioning delays have non-negligible, highly variable values. To make accurate scheduling decisions, algorithms need to consider this delay when making runtime estimates. Its effect is spe-

cially noticeable in situations in which the number of VMs in the resource pool is highly dynamic due to performance requirements, topological features of the workflow, and provisioning strategies designed to save cost. All of the algorithms that acknowledge this delay, do so by associating an estimate of its value to each VM type. Another strategy used is to avoid these delays by reusing leased VMs when possible.

VM Deprovisioning Delay. The impact of VM deprovisioning delays is strictly limited to the execution cost. To illustrate this, consider the case in which a scheduler requests to shutdown a VM just before the end of the first billing period. By the time the instance is actually released, the second billing period has started and hence two billing periods have to be paid for. Those algorithms that consider this delay do so by allowing some time, an estimate of the deprovisioning value, between the request to shutdown the VM and the end of the billing period.

VM Core Count

This category refers to whether algorithms are aware of multi-core VMs for the purpose of scheduling multiple, simultaneous tasks on them (Figure 2.14).

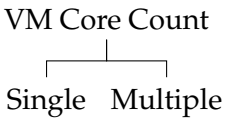


Figure 2.14: Types of VM core count

Single. Most algorithms assume VMs have a single core and hence are only capable of processing one task at a time. This simplifies the scheduling process and eliminates further performance degradation and variability due to resource contention derived from the co-scheduling of tasks.

Multiple. IaaS providers offer VMs with multiple cores. Algorithms that decide to take advantage of this feature may schedule multiple tasks to run simultaneously in the same

VM, potentially saving time, cost, and avoiding intermediate data transfers. However, this co-scheduling of tasks may result in significant performance degradation due to resource contention. Being mindful of this is essential when making scheduling decisions as estimating task runtimes assuming optimal performance will most definitely incur in additional, significant delays. Zhu et al. [126] for example, bundle tasks together based on their resource usage characteristics; tasks assigned to the same VM should have different computational requirements to minimize resource contention.

2.3 Survey

This section discusses a set of algorithms relevant to each of the categories presented in the taxonomy and depicts a complete classification including all of the surveyed algorithms, these results are summarized in tables 2.1, 2.2, 2.3, and 2.4.

2.3.1 Scheduling Multilevel Deadline-Constrained Scientific Workflows

Malawski et al. [75] present a mathematical model that optimizes the cost of scheduling workflows under a deadline constraint. It considers a multi-cloud environment where each provider offers a limited number of heterogeneous VMs and a global storage service is used to share intermediate data files. Their method proposes a global optimization of task and data placement by formulating the scheduling problem as a Mixed Integer Program (MIP). Two different versions of the algorithm are presented, one for coarse-grained workflows, in which tasks have an execution time in the order of one hour, and another for fine-grained workflows with many short tasks and with deadlines shorter than one hour.

The MIP formulation to the problem takes advantage of some characteristics of large-scale scientific workflows: they are composed of sequential levels of independent tasks. Based on this, the authors decided to group tasks in each level based on their computational cost and input/output data and schedule these groups instead of single tasks, reducing the complexity of the MIP problem considerably. Another design choice to keep the MIP model simple is that VMs cannot be shared between levels, however this

may potentially lead to low resource utilization and higher costs for some workflows. Since the MIP model already assumes VMs cannot be shared between levels, a potential improvement could be to design the MIP program so that the schedule for each level can be computed in parallel. Finally, the algorithm is too reliant on accurate runtime, storage, and data transfer time estimations, considering the fact that its main objective is to finish executions before a deadline.

2.3.2 SABA

The Security-aware and Budget-aware (SABA) algorithm [122] was designed to schedule workflows in a multi-cloud environment. The authors define the concept of immoveable and movable datasets. Movable data has no security restrictions and hence can be moved between data centers and replicated if required. Immoveable data on the other hand, are restricted to a single data center and cannot be migrated or replicated due to security or cost concerns. The algorithm consists of three main phases. The first one is the clustering and prioritization stage in which tasks and data are assigned to specific data centers based on the workflow's immoveable data sets. In addition to this, priorities are assigned to tasks based on their computation and I/O costs on a baseline VM type. The second stage statically assigns tasks to VMs based on a performance-cost ratio. Finally, the intermediate data is moved dynamically at runtime with the location of tasks that are ready for execution guiding this process. SABA calculates the cost of a VM based on the start time of the first task assigned to it and the end time of the last task mapped to it. Even though the authors do not specifically describe a resource provisioning strategy, the start and end times of VMs can be derived from the start and end times of tasks and therefore we classify it as adopting an elastic resource pool strategy.

In addition to the security of data, SABA also considers tasks that may require security services such as authentication, integrity, and confidentiality and includes the overheads of using these services in their time and cost estimations. What is more, instead of considering just the CPU capacity of VMs to estimate runtimes, SABA also considers features such I/O, bandwidth, and memory capacity. The cost of VMs is calculated based on the total units of time the machine was used for and billing periods imposed by providers

are not considered. This may results in higher VM costs than expected when using the algorithm on a real cloud environment. Other costs considered include data transfer costs between data centers as well as the storage used for input and output workflow data.

2.3.3 PSO-based Resource Provisioning and Scheduling Algorithm

The PSO-based algorithm developed by Rodriguez and Buyya and presented in Chapter 3 is a static, cost minimization, deadline-constrained algorithm that considers features such as the elastic provisioning and heterogeneity of unlimited compute resources as well as VM performance variation. Both resource provisioning and scheduling are merged and modeled as a Particle Swarm Optimization (PSO) problem. The output of the algorithm is hence, a near-optimal schedule determining the number and types of VMs to use, as well as their leasing periods and the task to resource mapping.

The global optimization technique is an advantage of the algorithm as it allows it to generate high-quality schedules. Also, to deal with the inability of the static schedule to adapt to environmental changes, the authors introduce an estimate of the degradation in performance that would be experienced by VMs when calculating runtimes. In this way, a degree of tolerance to the unpredictability of the environment is introduced. The unlimited resource model is successfully captured by the algorithm, however the computational overhead increases rapidly with the number of tasks in the workflow and the types of VMs offered by the provider.

2.3.4 MOHEFT

Durillo and Prodan developed the Multi-objective Heterogeneous Earliest Finish Time (MOHEFT) algorithm [46] as an extension of the well-known DAG scheduling algorithm HEFT [104]. The heuristic-based method computes a set of pareto-based solutions from which users can select the best-suited one. MOHEFT builds several intermediate workflow schedules, or solutions, in parallel in each step, instead of a single one as is done by HEFT. The quality of the solutions is ensured by using dominance relationships while their diversity is ensured by making use of a metric known as crowding distance. The al-

gorithm is generic in the number and type of objectives it is capable of handling, however, makespan and cost were optimized when running workflow applications in an Amazon-based commercial cloud.

The flexibility offered by MOHEFT as a generic multi-objective algorithm is very appealing. In addition, the pareto front is an efficient tool for decision support as it allows users to select the most appropriate trade-off solution based on their needs. For example, their experiments demonstrated that in some cases, cost could be reduced by half with a small increment of 5% in the schedule makespan. Finally, as noted by the authors, most of the solutions computing the pareto front are based on genetic algorithms. These approaches require high computation time while MOHEFT offers an approximate time complexity of $O(n \times m)$ where n is the number of tasks and m the number of resources.

2.3.5 Fault-Tolerant Scheduling Using Spot Instances

Poola et al. [93] propose an algorithm that schedules tasks on two types of cloud instances, namely on-demand and spot. Specifically, it considers a single type of spot VM type (the cheapest one) and multiple types of on-demand VMs. The authors define the concept of latest time to on-demand, or LTO. It determines when the algorithm should switch from using spot to on-demand instances to ensure the user-defined deadline is met. A bidding strategy for spot VMs is also proposed; the bidding starts close to the initial spot price and increases as the execution progresses so that it gets closer to the on-demand price as the LTO approaches. This lowers the risk of out-of-bid events closer to the LTO and increases the probability of meeting the deadline constrain.

This algorithm is one of the few exploring the benefits of using dynamically priced VMs. It addresses a challenging problem by aiming to meet deadlines not only under variable performance but also under unreliable VMs that can be terminated at any point in time. The benefits are clear with the authors finding that by using spot instances, the algorithm is able to considerably lower the execution cost. However, this advantage may be reduced due to the fact that only the cheapest spot VM is considered. If deadlines are tight, the cheapest VM may not be able to process many tasks before the LTO and hence most of the workflow execution would happen in on-demand instances. Another

potential drawback of the algorithm is its reliance on checkpointing. Not only are many scientific workflows legacy applications lacking checkpointing capabilities but storing data for this purpose may considerably increase the infrastructure cost.

2.3.6 IC-PCP

The IaaS Cloud Partial Critical Path (IC-PCP) algorithm [22] has as objective to minimize the execution cost while meeting a deadline constraint. The algorithm begins by finding a set of tasks, namely partial critical paths (PCPs), associated to each exit node of the workflow (an exit node is defined as a node with no children tasks). The tasks on each path are then scheduled on the same VM and are preferably assigned to an already leased instance which can meet the latest finish time requirements of the tasks. If this cannot be achieved, the tasks are assigned to a newly leased VM of the cheapest type that can finish them on time. PCPs are recursively identified and the process is repeated until all of the workflow tasks have been scheduled.

Along with IC-PCP and with the same scheduling objectives, the authors propose the IC-PCPD2 (IC-PCP with deadline distribution algorithm). The main difference between both algorithms is that, instead of assigning all tasks in a path to the same VM, IC-PCPD2 places each individual task on the cheapest VM that can finish it on time. According to the authors, IC-PCP outperforms IC-PCPD2 in most of the cases. This highlights one of the main advantages of IC-PCP and an important consideration regarding workflow executions in clouds: data transfer times can have a high impact on the makespan and cost of a workflow execution. IC-PCP successfully addresses this concern by scheduling parent and child tasks on the same VM thereby reducing the amount of VM to VM communication.

A disadvantage of IC-PCP is that it does not account for VM provisioning delays or for resource performance variation. This makes it highly sensitive to CPU performance degradation and causes deadlines to be missed due to unexpected delays. Its static and heuristic based nature however, allows it to find high quality schedules efficiently, making it suitable to schedule large-scale workflows with thousands of tasks. Hence, IC-PCP could be better suited to schedule large workflows with tasks that have low CPU require-

ments so that the impact of resource performance degradation is reduced.

2.3.7 EIPR

Calheiros and Buyya [36] propose the Enhanced IC-PCP with Replication (EIPR) algorithm, a scheduling and provisioning solution that uses the idle time of provisioned VMs and a budget surplus to replicate tasks in order to mitigate the effect of performance variation and meet the application's deadline. The first step of the algorithm consists in determining the number and type of VMs to use as well as the order and placement of the tasks on these resources. This is achieved by adopting the main heuristic of IC-PCP [22], that is, identifying partial critical paths and assigning their tasks to the same VM. The second step is to determine the start and stop time of VMs. EIPR does this by considering both, the start and end time of tasks as well as input and output data transfer times. Finally, the algorithm replicates tasks in idle time slots of provisioned VMs or on new VMs if the replication budget allows for it. The algorithm prioritizes the replication of tasks with a large ratio of execution to available time, then tasks with long execution times, and finally tasks with a large number of children.

Although a static algorithm, EIPR is successful in mitigating the effects of poor and variable performance of resources by exploiting the elasticity and billing scheme of clouds. This allows it to generate high quality schedules while being robust to unexpected environmental delays. However, the replication of tasks may not be as successful in cases in which the execution time of tasks is close to the size of the billing period. This is mainly because there are less chances or reusing idle time-slots. Another advantage of the algorithm is its accountability of VM provisioning delays and its data-transfer aware provisioning adjust, which enables VMs to be provisioned before the actual start time of their first task to allow for input data to be transferred beforehand.

2.3.8 Workflow Scheduling Considering Two SLA Levels

Genez et al. [52] implement a SaaS provider offering a workflow execution service to its customers. They consider two types of SLA contracts that can be used to lease VMs from

IaaS providers: static and subscription based. Specifically they consider the corresponding options offered by Amazon EC2, namely on-demand and reserved instances. In their model, the SaaS provider has a pool of reserved instances that are used to execute workflows before a user-defined deadline. However, if the reserved instance infrastructure is not enough to satisfy the deadline, then on-demand instances are acquired and used to meet the workflow's requirements. Even though their algorithm is presented in the context of a SaaS provider potentially serving multiple users, the solution is designed to schedule a single workflow at time. They formulate the scheduling problem as a mixed integer linear program (MILP) with the objective of minimizing the total execution cost while meeting the application deadline of the workflow. They then propose two heuristics to derive a feasible schedule from the relaxed version of the MILP. Their algorithm is capable of selecting the best-suited IaaS provider as well as the VMs required to guarantee the QoS parameters.

The scalability of the MILP model presented is a concern. The number of variables and constraints in the formulation increases rapidly with the number of providers, maximum number of VMs that can be leased from each provider, and the number of tasks in the DAG. This may rule the algorithm as impractical in many real-life scenarios, especially considering the fact that even after a time-expensive schedule computation, the workflow may still finish after its deadline due to poor and variable resource performance. Aware of this limitation, the authors propose a relaxation approach and application of time limits to the MILP solver, however the scalability concerns still remain in this cases as workflows are likely to have thousands of tasks. On the positive side, the MILP finds an optimal solution to their formulation of the problem and can be successfully used in scenarios where workflows are small or even as a benchmark to compare the quality of schedules generated by different algorithms.

2.3.9 PBTS

The Partitioned Balanced Time Scheduling (PBTS) algorithm [35] was designed to process a workflow in a set of homogeneous VMs by partitioning its execution so that scheduling decisions are made every billing period. Its main objective is to estimate, for each

scheduling cycle or partition, the minimum number of compute resources required to execute the workflow within the user-specified deadline. For each partition, PBTS first identifies the set of tasks to run based on an approximate resource capacity estimate that considers the total cost. Then, it estimates the exact number of resources needed to run the tasks during the partition using the Balanced Time Scheduling (BTS) algorithm [34], which was previously proposed by the same authors. Finally, the actual VMs are allocated and the tasks executed based on the schedule obtained from running BTS.

Adjusting the number of VMs and monitoring the execution of tasks every scheduling cycle allows the algorithm to have a higher tolerance to performance variability and take advantages of the elasticity of clouds. PBTS is a good example of an algorithm using a sub-workflow static hybrid approach to address the task to VM mapping, statically scheduling tasks every billing period. It also uses runtime refinement to handle delays on the statically scheduled tasks. The algorithm is capable of handling tasks that require multiple hosts for their execution (e.g., MPI tasks), and even though this is out of the scope of this survey, we include it as it still has the ability to schedule workflows were all tasks require a single host. PBTS was clearly designed for coarse-grained billing periods, such as one hour. For finer-grained periods, such as one minute, PBTS may not be as successful as tasks are unlikely to finish within a single partition and it would be difficult to assign a large-enough number of tasks to each partition to make the scheduling overhead worthwhile.

2.3.10 SPSS and DPDS

Malawski et al. [76] propose two algorithms to schedule workflow ensembles that aim to maximize the number of executed workflow instances while meeting deadline and budget constraints. The Dynamic Provisioning Dynamic Scheduling (DPDS) algorithm first calculates the initial number of VMs to use based on the budget and deadline. This VM pool is then updated periodically based on the VM utilization; if the utilization falls below a predefined threshold then VMs are shutdown and if it exceeds this threshold and the budget allows for it then new VMs are leased. The scheduling phase assigns tasks based on their priority to arbitrarily chosen VMs dynamically until all of the work-

flow instances are executed or until the deadline is reached. WA-DPDS (Workflow Aware DPDPS) is a variant of the algorithm that aims to be more efficient by executing only tasks of workflows that can be finished within the specified QoS constraints. It incorporates an admission control procedure so that only those workflow instances that can be completed within the specified budget are scheduled and executed. The authors demonstrate the ability of DPDS to adapt to unexpected delays, including considerable provisioning delays and inaccurate task runtime estimates. A drawback of the algorithm is leasing as many VMs as allowed by the budget from the beginning of the ensemble execution. This may result in VMs being idle for long periods of time as they wait for tasks to become ready for execution resulting in wasted time slots and additional billing periods.

The Static Provisioning Static Scheduling (SPSS) algorithm assigns sub-deadlines to each task based on the slack time of the workflow (the time that the workflow can extend its critical path and still finish by the deadline). The tasks are statically assigned to free time slots of existing VMs so that the cost is minimized and their deadline is met. If there are no time slots that satisfy these constraints then new VMs are leased to schedule the tasks. Being a static approach, the authors found that SPSS is more sensitive to dynamic changes in the environment than DPDS. However, it outperforms its dynamic counterpart in terms of the quality of schedules as it has the opportunity to use its knowledge on the workflow structure and to compare different outputs before choosing the best one. The major drawback of SPPS is its static nature and unawareness of VM provisioning times, as the authors found it to be too sensitive to these delays for it to be of practical use.

2.3.11 SPSS-ED and SPSS-EB

Pietri et al. [91] propose two algorithms to schedule workflow ensembles in clouds, both based on SPSS [76]. One of them, called SPSS-ED, focuses on meeting energy and deadline constraints while the other one, called SPSS-EB, focuses on meeting energy and budget constraints. Both algorithms aim to maximize the number of completed workflows. For each workflow in the ensemble, SPSS-EB plans the execution of the workflow by scheduling each task so that the total energy consumed is minimum. It then accepts the

plan and executes the workflow only if the energy and budget constraints are met. The same process is used in SPSS-ED but instead of budget, deadline is considered as a constraint.

This work does not consider data transfer times and considers only a single type of VM for simplicity. It also assumes the data center is composed of homogeneous hosts with fixed capacity in VMs. In reality however, data centers are composed of heterogeneous servers with different characteristics. Furthermore, it assumes physical hosts are exclusively used for the execution of the workflows in the ensemble and this again is an unrealistic expectation. Despite this disadvantages, this is the first work that considers energy consumption when scheduling ensembles and hence can be used as a stepping stone to make further advances in this area.

2.3.12 Dyna

Dyna [124] is a scheduling framework that considers the dynamic nature of cloud environments from the performance and pricing point of view. It is based on a resource model similar to Amazon EC2 as it considers both spot and on-demand instances. The goal is to minimize the execution cost of workflows while offering a probabilistic deadline guarantee that reflects the performance variability of resources and the price dynamics of spot instances. Spot instances are used to reduce the infrastructure cost and on-demand instances to meet the deadline constraints when Spot instances are not capable of finishing tasks on time. This is achieved by generating a static hybrid instance configuration plan (a combination of spot and on-demand instances) for every task. Each configuration plan indicates a set of spot instances to use along with their bidding price and one on-demand instance type which should be used in case the execution fails on each of the spot instances on the configuration set. At runtime, this configuration plan, in addition to instance consolidation and reuse techniques are used to schedule the tasks.

Contrary to most algorithms, Dyna recognizes that static task runtime estimations and deterministic performance guarantees are not suited to cloud environments. Instead, the authors propose offering users a more realistic, probabilistic deadline guarantee that reflects the cloud dynamics. Their probabilistic models are successful in capturing the

variability in I/O and network performance, as well as in spot prices. However, Dyna does not consider CPU performance variations as according to the authors, their findings show it is relatively stable. Additionally, by determining the best instance type for each task statically, Dyna is able to generate better quality schedules, however, it still makes scheduling decisions for one task at a time, limiting its global task to VM mapping optimization capabilities.

2.3.13 SCS

SCS [77] is a deadline-constrained algorithm that has an auto-scaling mechanism to dynamically allocate and deallocate VMs based on the current status of tasks. It begins by bundling tasks in order to reduce data transfer times and by distributing the overall deadline among tasks. Then, it creates a *load vector* by determining the most cost-efficient VM type for each task. This load vector is updated every scheduling cycle and indicates how many machines of each type are needed in order for the tasks to finish by their assigned deadline with minimum cost. Afterwards, the algorithm proceeds to consolidate partial instance hours by merging tasks running on different instance types into a single one. This is done if VMs have idle time and can complete the additional tasks by its original deadline. Finally, the Earliest Deadline First (EDF) algorithm is used to map tasks onto running VMs; that is, the task with the earliest deadline is scheduled as soon as an instance of the corresponding type is available.

SCS is an example of an algorithm that makes an initial resource provisioning plan based on a global optimization heuristic and then refines it at runtime to respond to delays that were unaccounted for. The optimization heuristic allows it to minimize the cost and the runtime refinement to ensure there are always enough VMs in the resource pool so that tasks can finish on time. However, the refinement of the provisioning plan is done by running the global optimization algorithm for the remaining tasks every time a task is scheduled. This introduces a high computational overhead and hinders its scalability in terms of the number of tasks in the workflow.

Table 2.1: Workflows used in the evaluation of the surveyed algorithms.

Algorithm	Evaluation Strategy	Montage	CyberShake	Epigenomics	SIPHT	LIGO	Randomly Generated	Other
Malawski et al. [75]	Real cloud	✓	✓	✓	✓	✓		
SABA [122]	Simulation				✓	✓	✓	Gene2Life, Motif, NCFS, PSMerge
WRPS [97]	Simulation	✓		✓	✓	✓		
RNPSO [71]	Simulation						✓	
Rodriguez&Buyya [96]	Simulation	✓	✓		✓	✓		
MOHEFT [46]	Simulation						✓	WIEN2K, POVRay
Poola et. al. [93]	Simulation					✓		
Poola et al.(Robust) [92]	Simulation	✓	✓	✓	✓	✓		
IC-PCP/IC-PCPD2 [22]	Simulation	✓	✓	✓	✓	✓		
EIPR [36]	Simulation	✓	✓		✓	✓		
Stretch&Compact [70]	Simulation	✓	✓	✓	✓	✓		
Oliveira et al. [41]	Real cloud							SciPhy
Genez et al. [52]	Simulation	✓					✓	
PBTS [35]	Simulation	✓	✓	✓	✓	✓	✓	
BTS [34]	Simulation	✓	✓	✓	✓	✓	✓	
Zhu et al. [127]	Simulation	✓	✓	✓	✓	✓	✓	
SPSS [76]	Simulation	✓	✓	✓	✓	✓		
DPDS [76]	Simulation	✓	✓	✓	✓	✓		
SPSS-ED [91]	Simulation	✓			✓	✓		
SPSS-EB [91]	Simulation	✓			✓	✓		
Wang et al. [110]	Simulation						✓	
ToF [125]	Real cloud	✓				✓		
Dyna [124]	Simulation						✓	
SCS [77]	Simulation						✓	

Table 2.2: Algorithm classification for the application model.

Algorithm	Workflow Dynamicity
Malawski et al. [75]	Single
SABA [122]	Single
WRPS [97]	Single
RNPSO [71]	Single
Rodriguez&Buyya [96]	Single
MOHEFT [46]	Single
Poola et. al. [93]	Single
Poola et al.(Robust) [92]	Single
IC-PCP/IC-PCPD2 [22]	Single
EIPR [36]	Single
Stretch&Compact [70]	Single
Oliveira et al. [41]	Single
Genez et al. [52]	Single
PBTS [35]	Single
BTS [34]	Single
Zhu et al. [127]	Single
SPSS [76]	Ensemble
DPDS [76]	Ensemble
SPSS-ED [91]	Ensemble
SPSS-EB [91]	Ensemble
Wang et al. [110]	Multiple
ToF [125]	Multiple
Dyna [124]	Multiple
SCS [77]	Multiple

2.3.14 Algorithm Classification

This section contains the classification of the surveyed algorithms based on the presented taxonomy. In addition to this classification, Table 2.1 presents a summary indicating whether the algorithms were evaluated in real cloud environments or using simulation. This table also indicates whether the algorithms were evaluated using any of the workflows presented in Section 1.1.2, randomly generated ones, or other scientific applications.

Table 2.2 displays the application model summary. Table 2.3 depicts the classification of the algorithms from the scheduling model perspective. In the task-VM mapping dynamicity category, *RR* refers to the hybrid runtime refinement class and *SS* to the hybrid sub-workflow static one. In the resource provisioning strategy, the term *SP* is short for static VM pool. As for the algorithms in the dynamic elastic VM pool category, the abbreviation *CR* is used for constraint requirement while *PM* is used for performance metric. Finally, *HO* refers to the hybrid heuristic-optimal class in the optimization strategy cate-

Table 2.3: Algorithm classification for the scheduling model.

Algorithm	Task-VM Mapping Dynamicity	Resource Provisioning Strategy	Scheduling Objectives	Optimization Strategy
Malawski et al. [75]	Static	Static EP	Deadline & Cost	Hybrid HO
SABA [122]	Hybrid RR	Static EP	Budget & Makespan & Security	Heuristic
WRPS [97]	Hybrid SS	Dynamic CR	Deadline & Cost	Hybrid HO
RNPSO [71]	Static	Static EP	Deadline & Cost	Meta-heuristic
Rodriguez&Buyya [96]	Static	Static EP	Deadline & Cost	Meta-heuristic
MOHEFT [46]	Static	Static EP	Generic Multi-Objective	Heuristic
Poola et al. [93]	Dynamic	Dynamic CR	Deadline & Cost & Reliability	Heuristic
Poola et al.(Robust) [92]	Static	Static EP	Makespan & Cost	Heuristic
IC-PCP/IC-PCPD2 [22]	Static	Static EP	Deadline & Cost	Heuristic
EIPR [36]	Static	Static EP	Deadline & Cost	Heuristic
Stretch&Compact [70]	Static	Static SP	Makespan & Res. Util. & Cost	Heuristic
Oliveira et al. [41]	Dynamic	Dynamic PM	Deadline & Budget & Reliability	Heuristic
Genez et al. [52]	Static	Static EP	Deadline & Cost	Optimal
PBTS [35]	Hybrid SS	Dynamic CR	Deadline & Cost	Heuristic
BTS [34]	Static	Static SP	Deadline & Cost	Heuristic
Zhu et al. [127]	Static	Static EP	Makespan & Cost	Meta-heuristic
SPSS [76]	Static	Static EP	Budget & Deadline & Workload	Heuristic
DPDS [76]	Dynamic	Dynamic PM	Budget & Deadline & Workload	Heuristic
SPSS-ED [91]	Static	Static EP	Deadline & Workload & Energy	Heuristic
SPSS-EB [91]	Static	Static EP	Budget & Workload & Energy	Heuristic
Wang et al. [110]	Dynamic	Dynamic CR	Makespan & Cost	Heuristic
ToF [125]	Static	Static EP	Deadline & Cost	Heuristic
Dyna [124]	Dynamic	Dynamic CR	Probabilistic Deadline & Cost	Heuristic
SCS [77]	Dynamic	Dynamic CR	Deadline & Cost	Heuristic

gory. Table 2.4 shows the resource model classification.

2.4 Summary

This chapter studied algorithms developed to schedule scientific workflows in cloud computing environments. In particular, it focused on techniques considering applications modeled as DAGs and the resource model offered by public cloud providers. It presented a taxonomy based on a comprehensive study of existing algorithms that focused on features particular to clouds offering infrastructure services, namely VMs, storage, and network access, on a pay-per use basis. It also included and extended existing

Table 2.4: Algorithm classification for the resource model.

Algorithm	VM Leasing Model	VM Type Uniformity	Provider Dep. Model	Data Center Dep. Model	Data Sharing Model	Data Transfer Cost	Storage Cost	VM Pricing Model	VM Delays	VM Core Count
Malawski et al. [75]	Limited	Multiple	Multiple	Multiple	Shared	Yes	No	Static	No	Multiple
SABA [122]	Limited	Multiple	Multiple	Multiple	P2P	Yes	Yes	Time Unit	No	Single
WRPS [97]	Unlimited	Multiple	Single	Single	Shared	No	No	Static	Prov. & Deprov.	Single
RNPSO [71]	Unlimited	Multiple	Single	Single	P2P	No	No	Static	Prov.	Single
Rodriguez&Buyya [96]	Unlimited	Multiple	Single	Single	P2P	No	No	Static	Prov.	Single
MOHEFT [46]	Limited	Multiple	Single	Single	P2P	Yes	Yes	Static	No	Single
Poola et al. [93]	Unlimited	Multiple	Single	Single	P2P	No	No	Dynamic & Static	Prov.	Single
Poola et al.(Robust) [92]	Unlimited	Multiple	Single	Single	P2P	No	No	Static	Prov.	Single
IC-PCP/IC-PCPD2 [22]	Unlimited	Multiple	Single	Single	P2P	No	No	Static	No	Single
EIPR [36]	Unlimited	Multiple	Single	Single	P2P	No	No	Static	Prov.	Single
Stretch&Compact [70]	Unlimited	Single	Single	Single	P2P	No	No	Static	No	Multiple
Oliveira et al. [41]	Unlimited	Multiple	Multiple	Multiple	Shared	Yes	No	Static	No	Single
Genez et al. [52]	Limited	Multiple	Multiple	Single	P2P	No	No	Static & Subscription	No	Multiple
PBTS [35]	Unlimited	Single	Single	Single	Shared	No	No	Static	No	Single
BTS [34]	Unlimited	Single	Single	Single	Shared	No	No	Time unit	No	Single
Zhu et al. [127]	Unlimited	Multiple	Single	Single	P2P	No	No	Static	No	Single
SPSS [76]	Unlimited	Single	Single	Single	Shared	No	No	Static	No	Single
DPDS [76]	Unlimited	Single	Single	Single	Shared	No	No	Static	Prov. & Deprov.	Single
SPSS-ED [91]	Unlimited	Single	Single	Single	Shared	No	No	Static	No	Single
SPSS-EB [91]	Unlimited	Single	Single	Single	Shared	No	No	Static	No	Single
Wang et al. [110]	Unlimited	Multiple	Single	Single	P2P	No	No	Static	No	Single
ToF [125]	Limited	Multiple	Single	Single	P2P	No	No	Static	Prov.	Multiple
Dyna [124]	Unlimited	Multiple	Single	Single	P2P	No	No	Dynamic & Static	Prov.	Single
SCS [77]	Unlimited	Multiple	Single	Single	P2P	No	No	Static	Prov.	Single

classification approaches designed for general-purpose scheduling algorithms [38] and DAG scheduling in grids [118]. These were included as they are of particular importance when dealing with cloud environments and were complemented with a cloud-focused discussion.

Additionally, existing algorithms within the scope of this thesis were reviewed and classified with the aim of providing an overview of the characteristics of existing research. A description and discussion of various algorithms was also included with the aim of providing further details and understanding of prominent techniques as well as further insight into the field's future directions.

The abundance of resources and flexibility to use only those that are required is a clear challenge particular to cloud computing. Most of the surveyed algorithms address this problem by elastically adding new VMs when additional performance is required and shutting down existing ones when they are not needed anymore. In this way, algorithms are careful not to over-provision so that cost can be reduced and not to under-provision so that the desired performance can be achieved. Furthermore, some studied algorithms recognize that the ability to scale horizontally does not only provide aggregated performance, but also a way of dealing with the potential indeterminism of a workflow's execution due to performance degradation. However, the difficulty of this provisioning problem under virtually unlimited resources calls for further research in this area and efficiently utilizing VMs to reduce wastage should be further studied. Also, increased awareness and the development of efficient techniques to deal with the provisioning and deprovisioning delays of VMs is also necessary as the majority of algorithms do not factor these when making scheduling decisions.

It is worthwhile noticing as well that there are a limited number of works that adopt hybrid approaches to the task-to-VM mapping problem. Exploring these strategies to find a trade-off between the advantages and disadvantages of static and dynamic strategies could benefit the research field. The same observation can be made for the optimization strategy used by algorithms, with the vast majority being heuristic-based. Investigating different hybrid approaches that combine either optimal or sub-optimal solutions to smaller and simplified versions to the problem using heuristics would be of value to

the scientific community.

Finally, most of the surveyed algorithms assume a resource cost model where VMs are leased with a static price and have a billing period that is much larger than the average task execution time. Hence, they focus on re-using idle slots on leased VMs so that cost is reduced, as long as the performance objectives are not compromised. While this is true for most applications and providers offering coarse-grained billing periods such as Amazon EC2 [6], emergent services are offering more flexibility by reducing the length of their billing periods. For example, Google Compute Engine [9] charges for the first ten minutes and per minute afterwards while Microsoft Azure [13] bills per minute. This finer-grained billing periods eliminate the need to reuse VMs to increase resource utilization and reduce cost and allows algorithms to focus on obtaining better optimization results. Therefore, it would be of benefit to design algorithms that focus on this specific scenario instead of focusing on either hourly billed instances or generic algorithms that work for any period length.

Chapter 3

A Static Meta-heuristic Based Scheduling Algorithm

This chapter proposes a resource provisioning and scheduling strategy that is based on the meta-heuristic optimization technique, Particle Swarm Optimization (PSO). It aims to minimize the overall workflow execution cost while meeting a deadline constraint. The algorithm is evaluated using CloudSim and various well-known scientific workflows of different sizes. The results show that it performs better than current state-of-the-art algorithms.

3.1 Introduction

The orchestration of workflow tasks onto distributed resources has been studied extensively over the years, focusing on environments like grids and clusters. However, with the emergence of new paradigms such as cloud computing, novel approaches that address the particular challenges and opportunities of these technologies need to be developed.

As stated in Section 1.2, there are two main stages when planning the execution of a workflow in a cloud environment. The first one is the resource provisioning phase and the second one is the scheduling, or task to VM mapping, stage. Previous works in this area, especially those developed for grids or clusters, focused mostly on the scheduling phase. The reason behind this is that these environments provide a static pool of resources that are readily available to execute the tasks and whose configuration is known in advance. Since this is not the case in cloud environments, both problems need to be

This chapter is derived from: **Maria A. Rodriguez** and Rajkumar Buyya. "Deadline Based Resource Provisioning and Scheduling Algorithm for Scientific Workflows on Clouds." *IEEE Transactions on Cloud Computing*, Volume 2, Issue 2, Pages: 222-235, 2014.

addressed and combined in order to produce an efficient execution plan.

Another characteristic of previous works developed for clusters and grids is their focus on meeting application deadlines or minimizing the makespan of the workflow while ignoring the cost of the utilized infrastructure. While this is well suited for such environments, policies developed for clouds are obliged to consider the pay-per-use model of the infrastructure.

VM performance is an additional challenge presented by cloud platforms. VMs provided by current cloud infrastructures do not exhibit a stable performance in terms of execution times. This may have a significant impact when scheduling workflows on clouds and may cause the application to miss its deadline. Many scheduling policies rely on the estimation of task runtimes on different VMs in order to make a mapping decision. This estimate is done based on the VMs computing capacity and if this capacity is always assumed to be optimal during the planning phase, the actual task execution will most probably take longer and the task will be delayed. This delay will also impact the task's children and the effect will continue to escalate until the workflow finishes executing.

This work is based on the meta-heuristic optimization technique, Particle Swarm Optimization (PSO). PSO was first introduced by Kennedy and Ebehart [66] and is inspired on the social behavior of bird flocks. It is based on a swarm of particles moving through space and communicating with each other in order to determine an optimal search direction. PSO has better computational performance than other evolutionary algorithms [66] and fewer parameters to tune, which makes it easier to implement. Many problems in different areas have been successfully addressed by adapting PSO to specific domains; for instance this technique has been used to solve problems in areas such as reactive voltage control [51], pattern recognition [87], and data mining [100], among others.

This chapter introduces a static cost-minimization, deadline-constrained heuristic for scheduling a scientific workflow application in a cloud environment. It considers fundamental features of IaaS providers such as the dynamic provisioning and heterogeneity of unlimited computing resources as well as VM performance variation. To achieve this, both resource provisioning and scheduling are merged and modeled as an optimization problem. PSO is then used to solve such problem and produce a schedule defining not

only the task to resource mapping but also the number and type of VMs that need to be leased, the time when they need to be leased and the time when they need to be released. The contribution of this work is therefore, an algorithm with higher accuracy in terms of meeting deadlines at lower costs that considers heterogeneous resources that can be dynamically acquired and released and are charged on a pay-per-use basis.

The rest of this chapter is organized as follows. Section 3.2 presents the related work followed by the application and resource models as well as the problem definition in Section 3.3. Section 3.4 gives a brief introduction to PSO while Section 3.5 explains the proposed approach. Finally, Section 3.6 presents the evaluation of the algorithm followed by a summary in Section 3.7.

3.2 Related Work

Workflow scheduling on distributed systems has been widely studied over the years and is NP-complete. Therefore, it is impossible to generate an optimal solution within polynomial time and algorithms focus on generating approximate or near-optimal solutions. Numerous algorithms that aim to find a schedule that meets the user's QoS requirements have been developed. A vast range of the proposed solutions target environments similar or equal to community grids. This means that minimizing the application's execution time is generally the scheduling objective, a limited pool of computing resources is assumed to be available and the execution cost is rarely a concern. For instance, Rahman et al. [95] propose a solution based on the workflow's dynamic critical paths, Chen and Zhang [39] elaborate an algorithm based on Ant Colony Optimization that aims to meet different user QoS requirements and, finally, Yu and Buyya use Genetic Algorithms to implement a budget constrained scheduling of workflows on utility grids [117].

The aforementioned solutions provide a valuable insight into the challenges and potential solutions for workflow scheduling. However, they are not optimal for utility-like environments such as IaaS clouds. There are various characteristics specific to cloud environments that need to be considered when developing a scheduling algorithm. For example, Mao and Humphrey [77] propose a dynamic approach for scheduling work-

flow ensembles on clouds. They acknowledge that there are various types of VMs with different prices and that they can be leased on demand, depending on the application's requirements. Furthermore, they tailor their approach so that the execution cost is minimized based on the cloud's pricing model, that is, VMs are paid by a fraction of time, which in most cases is one hour. They try to minimize the execution cost by applying a set of heuristics such as merging tasks into a single one, identifying the most cost-effective VM type for each task and consolidating instances. Although this is a valid approach capable of reducing the execution cost of workflows on clouds, the solution proposed only guarantees a reduction on the cost and not a near-optimal solution.

Another work on workflow ensembles developed for clouds is presented by Malawski et al. [76]. They propose various dynamic and static algorithms that aim to maximize the amount of work completed, which they define as the number of executed workflows, while meeting QoS constraints such as deadline and budget. Their solutions acknowledge different delays present when dealing with VMs leased from IaaS cloud providers such as provisioning and deprovisioning delays. Furthermore, their approach is robust in the sense that the task's estimated execution time may vary based on a uniform distribution and they use a cost safety margin to avoid generating a schedule that goes over budget. Their work, however, considers only a single type of VM, ignoring the heterogeneous nature of IaaS clouds.

While the algorithms presented by Mao and Humphrey [77] and Malawski et al. [76] are designed to work with workflow ensembles, they are still relevant to the work done in this chapter since they were developed specifically for cloud platforms and as so include heuristics that try to embed the platform's model. More in line with this work is the solution presented by Abrishami et al. [22] which presents a static algorithm for scheduling a single workflow instance in an IaaS cloud. Their algorithm is based on the workflow's partial critical paths and it considers cloud features such as VM heterogeneity, pay-as-you-go and time interval pricing model. They try to minimize the execution cost based on the heuristic of scheduling all tasks in a partial critical path on a single machine that can finish the tasks before their latest finish time (which is calculated based on the application's deadline and the fastest available instance). However, they do not have

a global optimization technique in place capable of producing a near-optimal solution; instead, they use a task-level optimization and hence fail to utilize the whole workflow structure and characteristics to generate a better solution.

Other authors have used PSO to solve the workflow scheduling problem. Pandey et al. [89] proposed a PSO based algorithm to minimize the execution cost of a single workflow while balancing the task load on the available resources. While the cost minimization objective is highly desired in clouds, the load balancing one makes more sense in a non-elastic environment such as a cluster or a grid. The execution time of the workflow is not considered in the scheduling objectives and therefore this value can be considerably high as a result of the cost minimization policy. The authors do not consider the elasticity of the cloud and assume a fixed set of VMs is available beforehand. For this reason, the solution presented is similar to those used for grids where the schedule generated is a mapping between tasks and resources instead of a more comprehensive schedule indicating the number and type of resources that need to be leased, when they should be acquired and released, and in which order the tasks should be executed on them.

Wu et al. [112] also use PSO to produce a near-optimal schedule. Their work focuses on minimizing either cost or time while meeting constraints such as deadline and budget. Despite the fact that their heuristic is able to handle heterogeneous resources, just as Pandey et al. [89], it assumes an initial set of VMs is available beforehand and hence lacks in utilizing the elasticity of IaaS clouds.

Finally, Byun et al. [35] develop an algorithm that estimates the optimal number of resources that need to be leased so that the execution cost of a workflow is minimized. Their algorithm also generates a task to resource mapping and is designed to run online. The schedule and resources are updated every billing period (i.e. every hour) based on the current status of the running VMs and tasks. Their approach takes advantage of the elasticity of the cloud resources but fails to consider the heterogeneous nature of the computing resources by assuming there is only one type of VM available.

3.3 Problem Formulation

3.3.1 Application and Resource Models

This work is based on the DAG application model defined in Section 1.1.2. In addition, each workflow W has a deadline δ_W associated to it defined as a time limit for the execution of the workflow.

The IaaS cloud provider offers a range of VM types. A VM type VM_i is defined in terms of its processing capacity P_{VM_i} and cost per unit of time C_{VM_i}. This work targets workflow applications such as those presented in Section 1.1.2 and characterized by Juve et al. [64]. Based on the profiling results obtained in their work for memory consumption and the VM types offered by Amazon EC2, it is assumed that VMs have sufficient memory to execute the workflow tasks.

It is also assumed that for every VM type, the processing capacity in terms of Floating Point Operations per Second (FLOPS) is available either from the provider or can be estimated [86]. This information is used in the proposed algorithm to calculate the execution time of a task on a given VM. Performance variation is modeled by adjusting the processing capacity of each leased VM and introducing a performance degradation percentage deg_{VM_i}.

The billing period τ in which the pay-per-use model is based is specified by the provider; any partial utilization of the leased VM is charged as if the full time period was consumed. For instance, for $\tau = 60$ minutes, if a VM is used for 61 minutes, the user will pay for 2 periods of 60 minutes, that is, 120 minutes. Also, the assumption that there is no limit on the number of VMs that can be leased from the provider is made.

The execution time $ET_{t_i}^{VM_j}$ of task t_i in a VM of type VM_j is estimated using the size I_{t_i} of the task in terms of Floating Point Operations (FLOP). This is depicted in Equation 3.1. Additionally, $TT_{e_{ij}}$ is defined as the time it takes to transfer data between a parent task t_i and its child t_j and is calculated as depicted in Equation 3.2. To calculate $TT_{e_{ij}}$, it is assumed that the size of the output data $d_{t_i}^{out}$ produced by task t_i is known in advance and that the entire workflow runs on a single data center or region. This means that all the leased instances are located on the same region and that the bandwidth β between

each VM is roughly the same. Notice that the transfer time between two tasks being executed on the same VM is zero. Finally, the total processing time $PT_{t_i}^{VM_j}$ of a task in a VM is computed as shown in Equation 3.3, where k is the number of edges in which t_i is a parent task and s_k is zero whenever t_i and t_j run on the same VM or one otherwise.

$$ET_{t_i}^{VM_j} = I_{t_i}/(P_{VM_j} \times (1 - deg_{vm_j})) \tag{3.1}$$

$$TT_{e_{ij}} = d_{t_i}^{out}/\beta \tag{3.2}$$

$$PT_{t_i}^{VM_j} = ET_{t_i}^{VM_j} + (\sum_{1}^{k}(TT_{e_{ij}} \times s_k) \tag{3.3}$$

Many IaaS providers do not charge for data transfers if they are made within the same data center; hence, this work does not consider this fee when calculating the workflow's execution cost. Nevertheless, it does consider the fact that a VM needs to remain active until all the output data of the running task is transferred to the VMs running the child tasks. Moreover, when a VM is leased, it requires an initial provisioning time in order for it to be properly initialized and be made available to the user; this time is not negligible and needs to be considered in the schedule generation as it could have a considerable impact on the same. This algorithm acknowledges such delay present in most cloud providers [99] when generating a schedule and calculating the overall cost for the execution of the workflow.

3.3.2 Problem Definition

Resource provisioning and scheduling algorithms may have different objectives; this work focuses on finding a schedule to execute a workflow on IaaS computing resources such that the total execution cost is minimized and the deadline is met. A schedule $S = (R, M, TEC, TET)$ is defined in terms of a set of resources, a task to resource mapping, the total execution cost and the total execution time. $R = r_1, r_2, \ldots, r_n$ is the set of VMs that need to be leased; each resource r_i has a VM type VM_{r_i} associated to it as well

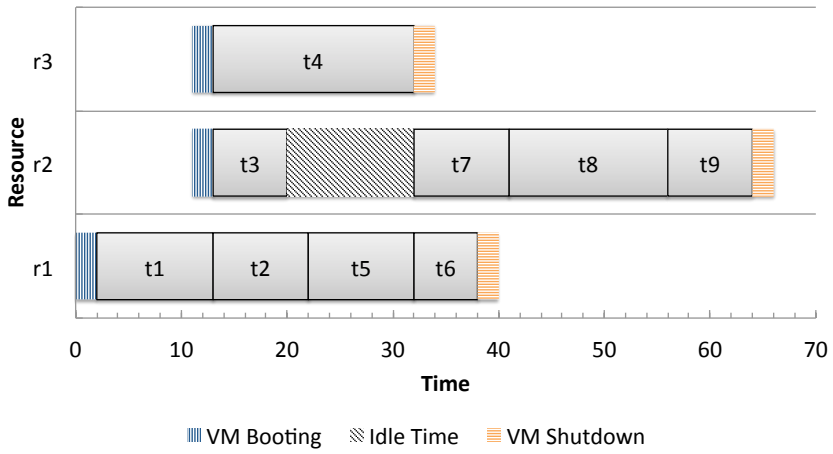


Figure 3.1: Example of a schedule generated for the workflow shown in Figure 1.3.

as an estimated lease start time LST_{r_i} and lease end time LET_{r_i}. M represents a mapping and is comprised of tuples of the form $m_{t_i}^{r_j} = (t_i, r_j, ST_{t_i}, ET_{t_i})$, one for each workflow task. A mapping tuple $m_{t_i}^{r_j}$ is interpreted as follows: task t_i is scheduled to run on resource r_j and is expected to start executing a time ST_{t_i} and complete by time ET_{t_i}. Equations 3.4 and 3.5 show how the total execution cost TEC and total execution time TET are calculated. Figure 3.1 shows a sample schedule generated for the workflow depicted in Figure 1.3 in Section 1.1.2.

$$TEC = \sum_{i=1}^{|R|} C_{VM_{r_i}} \times \lceil (LET_{r_i} - LST_{r_i})/\tau \rceil \tag{3.4}$$

$$TET = \max\{ET_{t_i} : t_i \in T\} \tag{3.5}$$

Based on the previous definitions, the problem can be formally defined as follows: find a schedule S with minimum TEC and for which the value of TET does not exceed the workflow's deadline.

$$\begin{gathered}\text{Minimize } TEC \\ \text{subject to } TET \leq \delta_W\end{gathered} \tag{3.6}$$

3.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an evolutionary computational technique based on the behavior of animal flocks. It was developed by Eberhart and Kennedy[66] in 1995 and has been widely researched and utilized ever since. The algorithm is a stochastic optimization technique in which the most basic concept is that of particle. A particle represents an individual (i.e. fish or bird) that has the ability to move through the defined problem space and represents a candidate solution to the optimization problem. At a given point in time, the movement of particles is defined by their velocity, which is represented as a vector and therefore has magnitude and direction. This velocity is determined by the best position in which the particle has been so far and the best position in which any of the particles has been so far. Based on this, it is imperative to be able to measure how good (or bad) a particle's position is; this is achieved by using a fitness function that measures the quality of the particle's position and varies from problem to problem, depending on the context and requirements.

Each particle is represented by its position and velocity. Particles keep track of their best position *pbest* and the global best position *gbest*; values that are determined based on the fitness function. The algorithm will then at each step, change the velocity of each particle towards the *pbest* and *gbest* locations. How much the particle moves towards these values is weighted by a random term, with different random numbers generated for acceleration towards *pbest* and *gbest* locations [68]. The algorithm will continue to iterate until a stopping criterion is met; this is generally a specified maximum number of iterations or a predefined fitness value considered to be good enough. In each iteration, the position and velocity of a particle are updated based in Equations 3.7 and 3.8 respectively. The pseudo code for the algorithm is shown in Algorithm 1.

$$\vec{x_i}(t+1) = \vec{x_i}(t) + \vec{v_i}(t) \tag{3.7}$$

$$\vec{v_i}(t+1) = \omega \cdot \vec{v_i}(t) + c_1 r_1(\vec{x_i^*}(t) - \vec{x_i}(t)) + c_2 r_2(\vec{x^*}(t) - \vec{x_i}(t)) \tag{3.8}$$

Algorithm 1 Particle Swarm Optimization

```
1: procedure PSO
2:     set the dimensions of the particles to d
3:     initialize the population of particles with random positions and velocities
4:     while stopping criterion is not met do
5:         for each particle do
6:             calculate fitness value f
7:             if f better than pbest then
8:                 set pbest to the current fitness value and location
9:             end if
10:            if f better than gbest then
11:                set gbest to the current fitness value and location
12:            end if
13:            update the position and velocity of the particle according to equations 3.7 and 3.8
14:        end for
15:    end while
16: end procedure
```

Where:

$$\omega = \text{inertia}$$

$$c_i = \text{acceleration coefficient }, i = 1, 2$$

$$r_i = \text{random number, } i = 1, 2 \text{ and } r_i \in [0, 1]$$

$$x_i^* = \text{best position of particle } i$$

$$x^* = \text{position of the best particle in the population}$$

$$x_i = \text{current position of particle } i$$

The velocity equation contains various parameters that affect the performance of the algorithm; moreover, some of them have a significant impact on the convergence of the algorithm. One of these parameters is ω, which is known as the inertia factor or weight and is crucial for the algorithm's convergence. This weight determines how much previous velocities will impact the current velocity and defines a tradeoff between the local cognitive component and global social experience of the particles. On one hand, a large inertia weight will make the velocity increase and therefore will favor global exploration. On the other hand, a smaller value would make the particles decelerate and hence favor local exploration. For this reason, a ω value that balances global and local search implies fewer iterations in order for the algorithm to converge.

Conversely, c_1 and c_2 do not have a critical effect in the convergence of PSO. However,

tuning them properly may lead to a faster convergence and may prevent the algorithm to get caught in local minima. Parameter c_1 is referred to as the cognitive parameter as the value $c_1 r_1$ in Equation 3.8 defines how much the previous best position matters. On the other hand, c_2 is referred to as the social parameter as $c_2 r_2$ in Equation 3.8 determines the behavior of the particle relative to other neighbors.

There are other parameters that are not part of the velocity definition and are used as input to the algorithm. The first one is the number of particles; a larger value generally increases the likelihood of finding the global optimum. This number varies depending on the complexity of the optimization problem but a typical range is between 20 and 40 particles. Other two parameters are the dimension of the particles and the range in which they are allowed to move, these values are solely determined by the nature of the problem being solved and how it is modeled to fit into PSO. Finally, the maximum velocity defines the maximum change a particle can have in one iteration and can also be a parameter to the algorithm; however, this value is usually set to be as big as the half of the position range of the particle.

3.5 Proposed Approach

3.5.1 PSO Modeling

There are two key steps when modeling a PSO problem. The first one is defining how the problem will be encoded, that is, defining how the solution will be represented. The second one is defining how the goodness of a particle will be measured, that is, defining the fitness function.

To define the encoding of the problem, the meaning and dimension of a particle needs to be established. For the scheduling scenario presented here, a particle represents a workflow and its tasks; thus, the dimension of the particle is equal to the number of tasks in the workflow. The dimension of a particle will determine the coordinate system used to define its position in space. For instance, the position of a 2-dimensional particle is specified by 2 coordinates, the position of a 3-dimensional one is specified by 3 coordinates and so on. As an example, the particle depicted in Figure 3.2 represents a

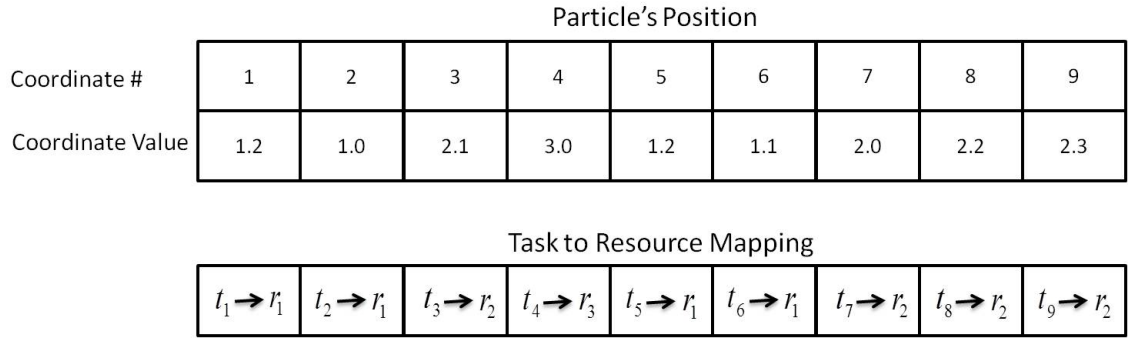


Figure 3.2: Example of the encoding of a particle's position.

workflow with 9 tasks; the particle is a 9-dimensional one and its position is defined by 9 coordinates, coordinates 1 through 9.

The range in which the particle is allowed to move is determined in this case by the number of resources available to run the tasks. As a result, the value of a coordinate can range from 0 to the number of VMs in the initial resource pool. Based on this, the integer part of the value of each coordinate in a particle's position corresponds to a resource index and represents the compute resource assigned to the task defined by that particular coordinate. In this way, the particle's position encodes a mapping of task to resources. Following the example given in Figure 3.2; there are 3 resources in the resource pool so each coordinate will have a value between 0 and 3. Coordinate 1 corresponds to task 1 and its value of 1.2 means that this task was assigned to resource 1. Coordinate 2 corresponds to task 2 and its value of 1.0 indicates that task 2 was assigned to resource 1. The same logic applies to the rest of the coordinates and their values.

Since the fitness function is used to determine how good a potential solution is, it needs to reflect the objectives of the scheduling problem. Based on this, the fitness function will be minimized and its value will be the total execution cost TEC associated to the schedule S derived from the particle's position. How this schedule is generated is explained later in this section.

Because of the elasticity and dynamicity of the resource acquisition model offered by IaaS providers, there is no initial set of available resources we can use as an input to the algorithm. Instead, we have the illusion of an unlimited pool of heterogeneous VMs that can be acquired and released at any point in time. Consequently, a strategy to define

an initial pool of resources that the algorithm can use to explore different solutions and achieve the scheduling objective needs to be put in place.

Such strategy needs to reflect the heterogeneity of the VMs and give PSO enough options so that a suitable particle (i.e. solution) is produced. If this initial resource pool is limited, then so will be the resources that can be used to schedule the tasks. If it is very large, then the number of possible schedules becomes very large and so does the search space explored by PSO, making it difficult for the algorithm to converge and find a suitable solution.

A possible approach would be to project the illusion of unlimited resources into the algorithm by simulating a pool of VMs, one of each type for each task. Notice that at this stage, the algorithm is evaluating various solutions and therefore no VMs need to be actually leased; a simple representation of them is sufficient for the algorithm to work at this point. This strategy though, may result in a very large VM pool and hence a very large search space.

Instead, to reduce the size of the search space, the following scheme is proposed. Let P be the set containing the maximum number of tasks that can run in parallel for a given workflow; then the initial resource pool $R_{initial}$ that PSO will use to find a near-optimal schedule will be comprised of one VM of each type for each task in P. The proposed algorithm will then select the appropriate number and type of VMs to lease from this resource pool. This strategy reflects the heterogeneity of the compute resources and reduces the size of the search space while still allowing the algorithm to execute all the tasks that can run in parallel to do so. The size of $R_{initial}$ would then be equal to $|P| \times n$ (where n is the number of available VM types) and thus, it is possible for PSO to select more than $|P|$ resources if required (unless $n = 1$).

As for the problem constraints, PSO was not designed to solve constrained optimization problems. To address this, a version of PSO that incorporates the constraint-handling strategy proposed by Deb et al [42] is used. In such strategy, whenever two solutions are being compared, the following rules are used to select the better one. If both of the solutions are feasible, then the solution with better fitness is selected. If on the other hand, one solution is feasible and the other one is not, then the feasible one is selected. Finally,

Algorithm 2 Schedule Generation

procedure GENERATESCHEDULE($T, R_{initial}, pos[|T|]$)
  $R = \emptyset$
  $M = \emptyset$
  $TEC = 0$
  $TET = 0$
  calculate $ExeTime[|T| \times |R_{initial}|]$
  calculate $TransferTime[|T| \times |T|]$
  for $i = 0$ to $i = |T| - 1$ **do**
    $t_i = T[i]$
    $r_{pos[i]} = R_{initial}[pos[i]]$
    if t_i has no parents **then**
      $ST_{t_i} = LET_{r_{pos[i]}}$
    else
      $ST_{t_i} = \max(\max\{ET_{t_p} : t_p \in parents(t_i)\}, LET_{r_{pos[i]}})$
    end if
    $exe = exeTime[i][pos[i]]$
    for each child t_c of t_i **do**
      if t_c is mapped to a resource different to $r_{post[i]}$ **then**
        $transfer+ = TrasnferTime[i][c]$
      end if
    end for
    $PT_{t_i}^{r_{pos[i]}} = exe + transfer$
    $ET_{t_i} = PT_{t_i}^{r_{pos[i]}} + ST_{t_i}$
    $m_{t_i}^{r_{pos[i]}} = (t_i, r_{pos[i]}, ST_{t_i}, ET_{t_i})$
    $M = M \cup m_{t_i}^{r_{pos[i]}}$
    if $r_{pos[i]} \notin R$ **then**
      $LST_{r_{pos[i]}} = \max(ST_{t_i}, bootTime)$
      $R = R \cup \{r_{pos[i]}\}$
    end if
    $LET_{r_{pos[i]}} = PT_{t_i}^{r_{pos[i]}} + ST_{t_i}$
  end for
  calculate TEC according to equation 3.4
  calculate TET according to equation 3.5
  $S = (R, M, TEC, TET)$
end procedure

if both solutions are infeasible, the one with the smaller overall constraint violation is selected. The latter scenario implies that a measure of how much a solution violates a constraint needs to be in place. Our problem specifies a single constraint, meeting the application's deadline. Therefore, the overall constraint violation value of a solution is defined as difference between the solution's makespan and the workflow's deadline. In this way, a solution whose makespan is closer to the deadline will be favored over a solution whose makespan is further away.

$$exeTime = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \\ t_9 \end{array} \begin{array}{c} \begin{array}{ccc} r_1 & r_2 & r_3 \end{array} \\ \begin{bmatrix} 2 & 1 & 4 \\ 4 & 3 & 6 \\ 10 & 6 & 15 \\ 7 & 4 & 12 \\ 8 & 4 & 10 \\ 3 & 2 & 7 \\ 12 & 7 & 18 \\ 9 & 5 & 20 \\ 13 & 8 & 19 \end{bmatrix} \end{array}$$

$$transferTime = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \\ t_9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 \end{array} \\ \begin{bmatrix} 0 & 9 & 9 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Figure 3.3: Sample execution and data transfer time matrix.

3.5.2 Schedule Generation

The pseudo code to convert a particle's position into a schedule is shown in Algorithm 2. Initially, the set of resources to lease R and the set of task to resource mappings M are empty and the total execution cost TEC and time TET are set to zero. After this, the algorithm estimates the execution time of each workflow task on every resource $r_i \in R_{initial}$. This is expressed as a matrix in which the rows represent the tasks, the columns represent the resources and the entry $ExeTime[i, j]$ represent the time it takes to run task t_i on resource r_j. This time is calculated using Equation 3.1. The next step is the calculation of the data transfer time matrix. Such matrix is represented as a weighted adjacency matrix of the workflow DAG where the entry $TransferTime[i, j]$ contains the time it takes to transfer the output data of task t_i to task t_j. This value is calculated using Equation 3.2 and is zero whenever $i = j$ or there is no directed edge connecting t_i and t_j. An example of these matrices is shown in Figure 3.3.

At this point the algorithm has all the information needed to begin decoding the par-

ticle's position and constructing the schedule. To achieve this, it iterates through every coordinate i in the position array pos and updates R and M as follows. Firstly, it determines which task and which resource are associated to the current coordinate and its value. This is accomplished by using the encoding strategy depicted earlier, which states that coordinate i corresponds to task t_i and its value $pos[i]$ corresponds to resource $r_{pos[i]} \in R_{initial}$. Now that the first two components of a mapping tuple are identified, the algorithm calculates the value of the remaining two, the start ST_{t_i} and end ET_{t_i} times of the task.

The start time value ST_{t_i} is based on two scenarios. In the first case, the task has no parents and therefore it can start running as soon as the resource it was assigned to is available; this value corresponds to the current end of lease time of resource $r_{pos[i]}$, which is $LET_{r_{pos[i]}}$. In the second case, the task has one or more parents. In this situation, the task can start running as soon as the parent task that is scheduled to finish last completes its execution and the output data is transferred. However, if the resource is busy with another task at this time, the execution has to be delayed until such VM is free to execute t_i.

The value of ET_{t_i} is calculated based on the total processing time and the start time of the task. To determine the processing time $PT_{t_i}^{r_{pos[i]}}$ we first need to compute the execution and the data transfer times. The former is simply the value in $ExeTime[i, pos[i]]$ whereas the latter is computed by adding the values in $TransferTime[i, child(i)]$ for every child task $t_{child(i)}$ of t_i which is mapped to run in a resource different to $r_{pos[i]}$. These two values are then added to obtain $PT_{t_i}^{r_{pos[j]}}$ as defined in Equation 3.3. Finally, the value of ET_{t_i}id obtained by adding ST_{t_i} and $PT_{t_i}^{r_{pos[j]}}$.

Now that all the elements of $m_{t_i}^{r_{pos[i]}} = (t_i, r_{pos[i]}, ST_{t_i}, ET_{t_i})$ have been computed, the algorithm needs to update two parameters associated to $r_{pos[i]}$ and add the resource to R if necessary. The first parameter is the time when the VM should be launched, $LST_{r_{pos[i]}}$. If the resource already exists in R then this means that it already has a start time and $LST_{r_{pos[i]}}$ does not need to be updated. However, if the resource is new and t_i is the first task to be assigned to it then R is updated so that it contains the new resource $r_{pos[i]}$ and $LST_{r_{pos[i]}}$ is set to be equal to either the start time of the task or the VM boot time, whichever is larger.

In this way, VM provisioning time is accounted for and the solution does no assume that a resource is available to use as soon as it is requested. The second parameter that needs to update is $LET_{r_{pos[i]}}$, this is the time when the resource is scheduled to finish executing the last task assigned to it and therefore is free to either run another task or be shutdown. The new lease end time of $r_{pos[i]}$ is thus the time it takes to process the task t_i ($PT_{t_i}^{r_{pos[j]}}$) plus the time when the resource is scheduled to start running, $LST_{r_{pos[i]}}$.

Once the algorithm finishes processing each coordinate in the position vector, R will contain all the resources that need to be leased as well as the times when they should be started and shutdown. Additionally, the entire task to resource mapping tuples will be in M and each task will have a resource assigned to it as well as an estimated start and end times. With this information, the algorithm can now use Equations 3.4 and 3.5 to compute the execution cost TEC and time TET associated to the current solution. After this, the algorithm has computed R, M, TEC and TET and therefore it can construct and return the schedule associated to the given particle's position.

Finally, Algorithms 1 and 2 are combined to produce a near optimal schedule. In Algorithm 1 instead of calculating the fitness value of the particle, a schedule is generated as outlined in Algorithm 2. Then TEC is used as a fitness value and the constraint handling mechanism is introduced, ensuring that TET does not exceed the application's deadline.

3.6 Performance Evaluation

In this section we present the experiments conducted in order to evaluate the performance of the proposed approach. We used the CloudSim framework [37] to simulate a cloud environment and chose four different workflows from different scientific areas: Montage, LIGO, SIPHT, and CyberShake. Each of these workflows has different structures as seen in Chapter 1, Section 1.1.2 and have different data and computational characteristics. Most of the tasks in the Montage workflow are characterized by being I/O intensive while not requiring much CPU processing capacity. LIGO is characterized by having CPU intensive tasks that consume large memory. The majority of tasks in the SIPHT application have a high CPU and low I/O utilization. Finally, CyberShake can

be classified as a data intensive workflow with large memory and CPU requirements. A comprehensive characterization of these workflows is presented by Juve et al. [64].

When leasing a VM from an IaaS provider, the user has the ability to choose different machines with varying configurations and prices. The first variation of the proposed algorithm, referred to as PSO, considers this heterogeneous environment. However, in order to evaluate if this resource heterogeneity has an impact on the makespan and cost of a workflow execution, a second variation of the algorithm called PSO_HOM is introduced. PSO_HOM considers a homogeneous environment in which only a single type of VM is used.

The IC-PCP [22] and SCS [77] algorithms were used as a baseline to evaluate the proposed solution. As mentioned in Section 3.2, IC-PCP was designed for the same problem addressed in this chapter: schedule a single workflow instance in an IaaS cloud whilst minimizing the execution cost and meeting the application's deadline. What is more, the algorithm considers many of the characteristics typical of IaaS resource models. For instance, the IC-PCP algorithm accounts for heterogeneous VMs which can be acquired on demand and are priced based on a predefined interval of time. It also considers data transfer times in addition to computation times of each task. However, IC-PCP does not account for VM startup time.

The IC-PCP algorithm begins by finding a set of tasks or critical paths associated to each exit node of the workflow (an exit node is defined as a node with no children tasks). The tasks on each path are scheduled on the same VM and are preferably assigned to an already leased instance which can meet the latest finish time requirements of the tasks. However, if this cannot be achieved, the cheapest instance that can finish the tasks before their latest finish time is leased and the path assigned to it. Finally, a critical path for each unassigned task on the scheduled path is calculated and the process is repeated until all tasks have been scheduled. At the end of this process, each task has been assigned to a VM and has a start and end times associated to it. Additionally, each VM has a start time determined by the start time of its first scheduled task and an end time determined by the end time of its last scheduled task.

SCS on the other hand, is a dynamic algorithm designed to schedule a group of work-

flows, instead of a single one. It can however be used to schedule a single instance without any modification. This algorithm was chosen as it has the same scheduling objectives and considers the same cloud model as this work. What is more, it is of interest to compare the performance of a static approach versus a dynamic one.

The SCS algorithm first identifies and bundles tasks with a one-to-one dependency into a single one. This is done in order to reduce data transfer times. After this, the overall deadline of the workflow is distributed over the tasks, with each task receiving a portion of the deadline based on the VM which is most cost-efficient for the task. The technique used to achieve this deadline assignment is done based on the work by Yu et al. [119]. The next step is to define a load vector for each VM type; this load vector indicates how many machines are needed in order for the tasks to finish by their assigned deadline at a given point in time. This value is calculated based on the execution interval derived from the deadline assignment phase and the estimated running time of a task on the specific instance type. Afterwards, the algorithm proceeds to consolidate partial instance hours by merging tasks running on different instance types into a single one if one of the VMs has idle time and can complete the additional task by its original deadline. Finally, Earliest Deadline First is used to map tasks onto running VMs; the task with the earliest deadline is scheduled as soon as an instance of the corresponding type is available.

An IaaS provider offering a single data center and six different types of VMs was modeled. The VM configurations are based on current Amazon EC2 offerings and are presented in Table 4.1. The work by Ostermann et al. [86] was used to estimate the processing capacity in MFLOPS based on the number of EC2 compute units. Each application was evaluated using workflows with approximately 100 tasks each.

A VM billing period of 1 hour was used and a boot time of 97 seconds assumed. The latter value was chosen based on the results obtained by Mao and Humphrey [78] for Amazon's EC2 cloud. Each experiment was executed 20 times.

One of the assumptions made by the evaluated algorithms is that the execution time of the tasks is known in advance; considering they target scientific workflows with well-known tasks and structures, it is reasonable to make such assumption and expect a quite accurate estimation. However, we acknowledge that such estimations are not 100% accu-

Table 3.1: VM types based on Amazon EC2 offerings.

Name	EC2 Units	Processing Capacity (MFLOPS)	Price per Hour
m1.small	1	4400	$0.06
m1.medium	2	8800	$0.12
m1.large	4	17600	$0.24
m1.xLarge	8	35200	$0.48
m3.xLarge	13	57200	$0.50
m3.doubleXLarge	26	114400	$1.00

rate and therefore introduce in the simulation a variation of $\pm 10\%$ to the provided task size based on a normal distribution.

Performance variation was modeled after the findings by Schad et al. [99]; the performance of each VM in the datacenter was diminished by at most 24% based on a normal distribution with mean 12% and standard deviation of 10%. In addition to this performance degradation, a data transfer variation of 19% [99] was modeled; the bandwidth available for each data transfer within the data center was subject to a degradation of at most 19% based on a normal distribution with mean 9.5% and a standard deviation of 5%.

The experiments were conducted using four different deadlines. These deadlines were calculated so that their values lie between the slowest and the fastest runtimes. To calculate these runtimes, two additional policies were implemented. The first one calculates the schedule with the slowest execution time; a single VM of the cheapest type is leased and all the workflow tasks are executed on it. The second one calculates the schedule with the fastest execution time; one VM of the fastest type is leased for each workflow task. Although these policies ignore data transfer times, they are still a good approximation to what the slowest and fastest runtimes would be. To estimate each of the four deadlines, the difference between the fastest and the slowest times is divided by five to get an interval size. To calculate the first deadline interval, one interval size is added to the fastest deadline, to calculate the second one two interval sizes are added and so on. In this way the behavior of the algorithms is analyzed as the deadlines increase from stricter values to more relaxed ones.

To select the best value for the PSO parameters c_1, c_2, and ω, an artificial workflow

with 100 tasks was defined and ran the algorithm with different parameter values. The values of c_1 and c_2 were varied from 1.5 to 2.0 and ω ranged from 0.1 to 1.0. To compare the results we considered the average workflow execution cost after running each experiment 10 times. The impact of c_1 and c_2 was found to be negligible. The inertia value ω had a slightly higher impact with the lowest average cost obtained with a value of 0.5. Based on this, we define $c_1 = c_2 = 2.0$ and $\omega = 0.5$ and use these parameter values in the rest of the experiments. The number of particles was set to 100.

3.6.1 Results and Analysis

Deadline Constraint Evaluation

To analyze the algorithms in terms of meeting the user defined deadline, we plotted the percentage of deadlines met for each workflow and deadline interval. The results are displayed in Figure 3.4. For the Montage workflow, IC-PCP fails to meet all of the deadlines. PSO-HOM meets fewer than 50% of the deadlines on interval 1 but improves its performance on interval 2 and achieves a 100% hit rate for both intervals 3 and 4. PSO and SCS are the best performing algorithms in terms of deadlines met with PSO meeting slightly less deadlines on intervals 1 and 2 but achieving 100% on the last two intervals.

The results for the SIPHT application again show that IC-PCP fails to meet the most number of deadlines and its performance is significantly worse than that of the other algorithms. SCS, PSO and PSO_HOM all meet the dead-line over 95% of the times for intervals 2, 3 and 4. The results obtained for the LIGO workflow and the SCS, PSO and PSO_HOM algorithms are very similar to those obtained for the SIPHT workflow. IC-PCP on the other hand is unable to meet any of the first 3 deadlines with a 0% hit rate. Its performance improves considerably for the 4th interval where it achieves a 100% hit rate.

As for the CyberShake workflow, IC-PCP meets the least amount of deadlines with the highest percentage being 30% on deadline interval 3. For deadline interval 1, SCS and PSO have the lowest percentages; however, as opposed to IC-PCP, SCS and the PSO based algorithms perform better as the deadline becomes more relaxed. The performance

of PSO, PSO_HOM and SCS is similar from deadline interval 2 onwards.

Overall, IC-PCP is outperformed by the other three algorithms. The percentage of deadlines met by this algorithm greatly differs from its counterparts and is in most cases under 50%. A possible explanation for this is the fact that IC-PCP fails to capture the dynamicity of the cloud by ignoring performance variation. Another feature that is not considered by the algorithm is the VM startup time, delay which is not negligible and might have a significant impact on the schedule, especially when a large number of VMs are needed to meet the specified deadline. SCS on the other hand, has a 100% hit rate in most of the cases. This are the results expected from a dynamic algorithm as it was designed to adapt to the conditions of the cloud to ensure the deadlines are met. Both PSO and PSO_HOM have a very similar performance to SCS having a 100% hit rate in most of the cases. Even though the PSO approach is offline, as is IC-PCP, it succeeds in considering the dynamic nature of the cloud and the variability of CPU performance. Overall, PSO, PSO_HOM and SCS meet the most number of deadlines for all of the workflows, making them the most appealing algorithms for the scheduling problem stated in this chapter. There is a considerable difference in the percentage of deadlines met by IC-PCP and these three algorithms.

Makespan Evaluation

The values obtained for the makespan of each of the workflows are displayed in Figure 3.5. The dotted line on each panel of each graph corresponds to the deadline value for the given interval. Evaluating the makespan with regards to this value is essential as all of the algorithms were designed to meet the given deadline. For the LIGO and Montage workflows, IC-PCP fails to meet this goal by producing schedules which take longer time to execute on average than the workflow's deadline. In fact, for the four deadline intervals, the Q1 (first quartile), median and Q3 (third quartile) values obtained with IC-PCP are considerably higher than the deadline. For the LIGO workflow, both of the PSO approaches and SCS have medians well below the deadline value for the four intervals and Q3 values smaller than the deadline on intervals 2, 3 and 4. This means that in at least 75% of the cases, they are able to produce schedules that finish on time. What is

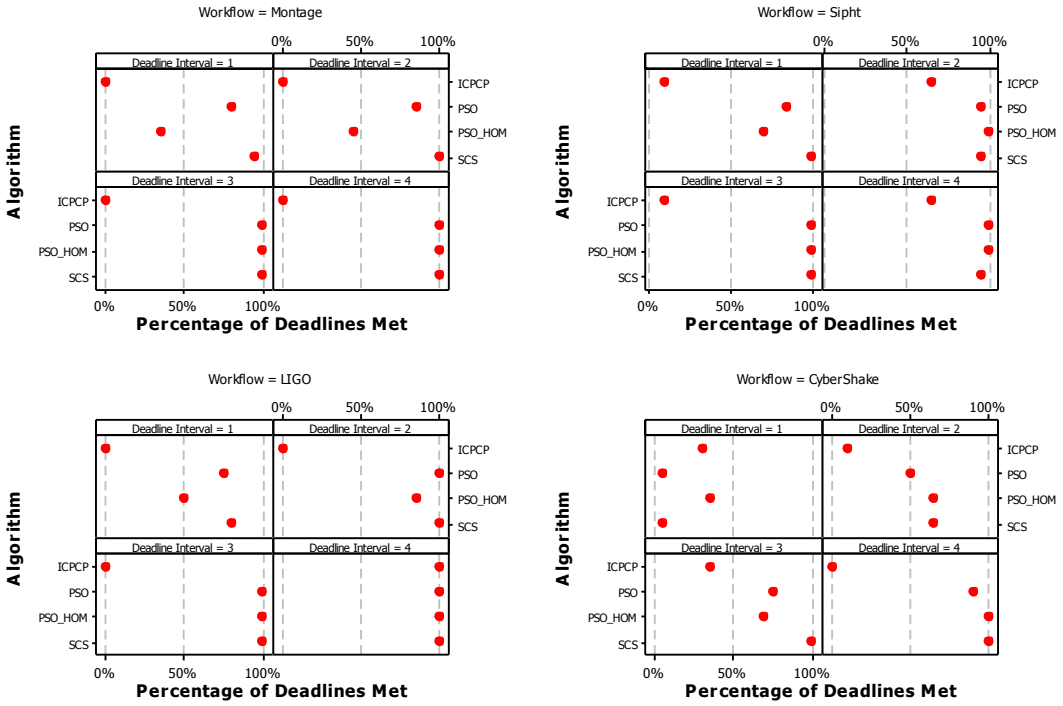


Figure 3.4: Individual value plot of deadlines met for each workflow and deadline interval.

more, PSO has the lowest average makespans on all the intervals followed by PSO_HOM for intervals 2 through 4. The results are similar for the Montage workflow.

In the SIPHT workflow case, PSO and PSO_HOM have the lowest average makespans while IC-PCP has the highest on every case. SCS has higher average makespans than both of the PSO approaches. PSO, PSO_HOM and SCS have median values lower than the deadline in every case and for deadline intervals 2 through 4 they have lower Q3 values.

IC-PCP performs better with the CyberShake workflow. It seems to have the best performance for deadline interval 1 with an average execution time close to the deadline. However, PSO performs similarly with a slightly higher median. For deadline interval 2, IC-PCP has the hi-guest Q1 value which is also higher than the deadline. PSO generates the fastest schedules on average with the lowest Q1, median and Q3 values. The results are similar for the deadline interval 3 but SCS has a lower Q3 value and less variation than PSO and PSO-HOM, resulting in a smaller average makespan. The average execution

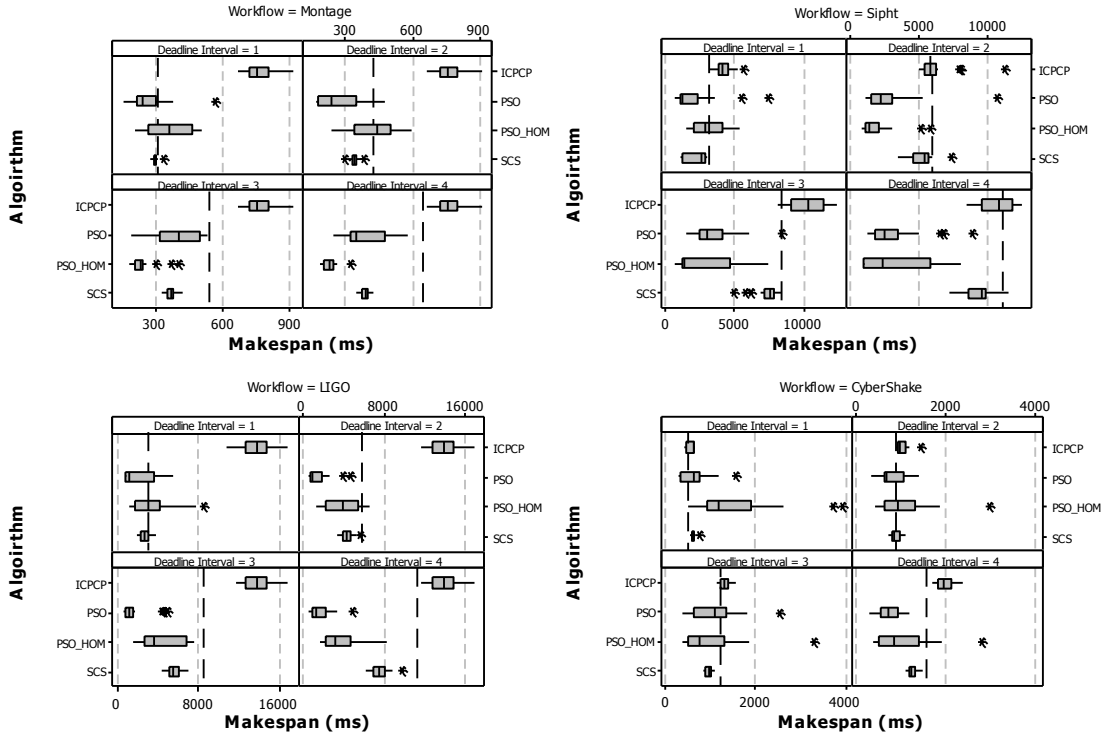


Figure 3.5: Boxplot of makespan by algorithm for each workflow and deadline interval. The reference line on each panel indicates the deadline value of the corresponding deadline interval.

time for IC-PCP on deadline interval 4 is above the deadline value, while both of the PSO based approaches and SCS have Q3 values smaller than the deadline.

These results are in line with those analyzed in the deadline constraint evaluation section, from which we were able to conclude that IC-PCP is not very efficient in meeting the deadlines whereas the other three heuristics are. For the LIGO and Montage workflows, the difference is substantial on every case. For the CyberShake and SIPHT workflows on the other hand, when larger than the deadline, the average IC-PCP makespan is only slightly larger than the deadline. Furthermore, IC-PCP and SCS being heuristic based algorithms are much more predictable in the sense that the execution time does not vary much from run to run. PSO_HOM and PSO on the contrary exhibit a larger makespan variation, which is expected as it is a meta-heuristic based approach with a very large search space.

Cost Evaluation

The average execution costs obtained for each workflow are shown in Figure 3.6. The mean makespan is also shown as the algorithms should be able to generate a cost-efficient schedule but not at the expense of a long execution time. The reference line on each panel displaying the mean makespan is the deadline corresponding to the given deadline interval. This is presented as there is no use in an algorithm generating very cheap schedules but not meeting the deadlines; the cost comparison is made therefore, amongst those heuristics which managed to meet the particular deadline in a give case.

For the Montage workflow, IC-PCP execution costs are the lowest ones for the four deadline intervals but its execution times are on average much higher than each of the four deadlines. Amongst the algorithms that do comply with the deadline constraint, PSO obtains the lowest cost on deadline interval 1; PSO_HOM on deadline interval 2 and finally, PSO for deadline intervals 3 and 4. From the results, it is clear that the PSO based strategies perform better than SCS in terms of cost by generating much cheaper schedules. The differences in costs are very pronounced for this workflow, with the costs obtained by the algorithms being significantly different to each other, especially when comparing the cheapest and the most expensive one. A possible explanation for this might be the fact that heuristics such as SCS and PSO lease more VMs in order to meet the deadline; this, combined with the fact that the Montage tasks are relatively small, means that the machines are only used for a small amount of time but charged for the full billing period. The reason for IC-PCP generating schedules with such a low cost and large makespan might be accounted to it not considering VM performance variation or boot time when estimating the schedule times, causing these to greatly differ from the actual execution ones.

The results for SIPHT show that the solution proposed in this chapter has the best performance in the first 3 deadline intervals; it achieves the lowest costs while having an average makespan smaller than the deadline. IC-PCP exhibits the lowest cost for the four deadlines; however, the average makespan obtained by this algorithm is higher than the deadline value for the first three intervals. Hence, IC-PCP outperforms the other heuristics with the most relaxed deadline but, on average, fails to produce schedules

that meet the three tighter deadlines. The other three algorithms succeed on meeting the deadline constraint on average in every case; this is supported by the boxplot of makespan depicted in Figure 7. Since SCS, PSO and PSO_HOM all meet the deadline; the best performing algorithm is the one capable of generating the schedule that leads to the lowest cost. For deadline interval 1, PSO_HOM achieves this, for deadline intervals 2 and 3 PSO does, and finally, as stated before, IC-PCP has the lowest cost on deadline interval 4. Overall, both PSO and SCS are capable of meeting the imposed deadline; however, PSO does so with a bigger window between the makespan and deadline and a lower cost while SCS results are closer to the deadline and have a higher price.

For the CyberShake application, the results for the IC-PCP algorithm show a considerably higher cost when compared to the other algorithms for deadline interval 1. In this particular scenario, IC-PCP has the lowest average makespan, with it being only slightly higher than the deadline value. The other algorithms on the other hand have lower costs but at the expense of having longer execution times which do not comply with the deadline constraint. PSO and SCS perform similarly in this particular scenario and obtain only slightly higher average makespans than IC-PCP but at much lower costs. For deadline intervals 2, 3 and 4, IC-PCP generates cheap schedules but it fails to meet each of the specified deadlines. PSO has the lowest average cost for deadline intervals 2, 3 and 4, making it the most efficient algorithm by meeting the deadlines at the lowest cost. Aside from deadline interval 1, all algorithms have very similar results in terms of cost for this application; PSO outperforms all the other heuristics on intervals 2, 3 and 4 by not only generating the cheapest schedule but also the fastest one in some of the cases. The performance of IC-PCP for the LIGO workflow is the same as for the SIPHT application; it achieves the lowest average cost in every case but produces the schedules with the longest execution times, which are well above the deadline value for the four intervals. PSO and SCS on the other hand meet on average the deadline on every case with PSO producing the most efficient schedules with shorter makespans and lower prices.

Overall, it was found that IC-PCP is capable of generating low cost schedules but fails to meet the deadline in these cases. SCS is very efficient when generating schedules that meet the deadline but because it is a dynamic and heuristic based approach, its cost

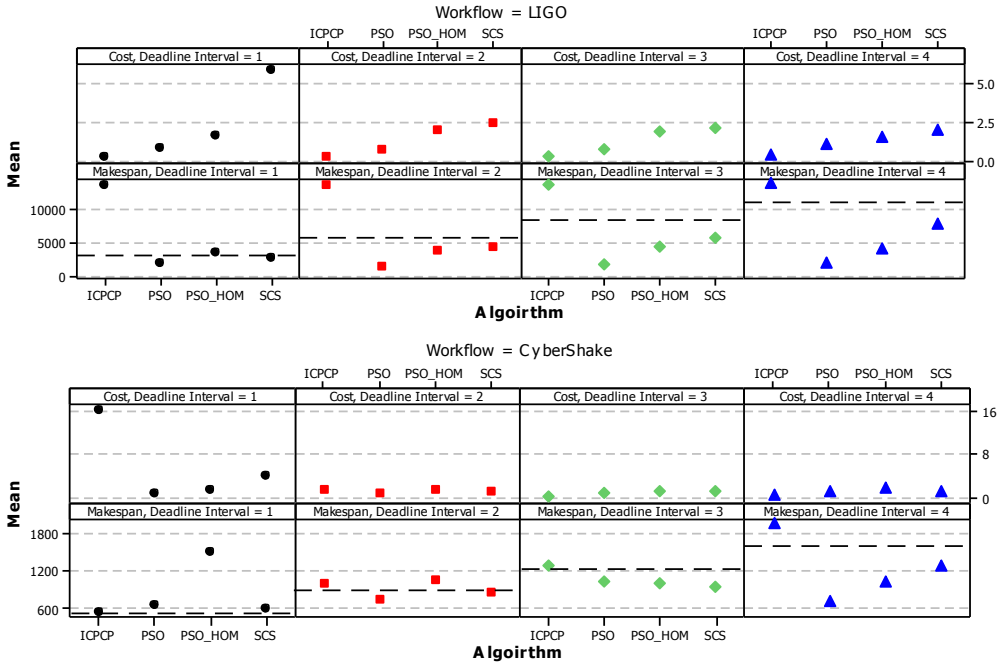


Figure 3.6: Lineplot of mean makespan (ms) and mean cost (USD) for each workflow and deadline interval. The reference line on each panel indicates the deadline value of the corresponding deadline interval.

optimization is not as good as that obtained by our solution. We found that in every case in which both algorithms meet the deadline, our approach incurs in cheaper costs, in some cases generating not only cheaper but faster solutions. As expected, PSO performs better than PSO_HOM.

Further Analysis

We found that in some cases our solution tends to generate schedules with lower makespans and higher costs as long as the deadline is met. Some users might prefer this as a solution whereas others might prefer the makespan to be closer to the deadline and pay a lower price.

Another finding is the significant impact that the selection of the initial pool of resources has on the performance of the algorithm. The same set of experiments was conducted with an initial VM pool composed of one VM of each type for each task. The results show that the algorithm takes longer to converge and find an optimal solution

producing schedules with higher execution times and costs. It was noticed that this strategy leads the algorithm into leasing more VMs with lower utilization rate and hence incurring in higher costs and more delays such as startup and data transfer times.

Overall, in most of the cases, IC-PCP fails to meet the deadlines whereas our approach and SCS succeed; the reason why IC-PCP fails to meet so many deadlines might be accounted to the fact that it does not consider VM boot time or performance variation. When compared to SCS, our algorithm is capable of generating cheaper schedules and hence outperforms it in terms of cost optimization.

Regarding the computational complexity of the algorithm, in each PSO iteration, the position and velocity of all particles is updated and their fitness is evaluated. The number of particles N and their dimension D determine the number of calculations required to update the position and velocity of particles. The fitness function complexity is based on the schedule generation algorithm and depends on the number of tasks T, and the number of resources being used R. Based on this and the fact that $D = T$ in our formulation, the proposed algorithm has an overall complexity of order $O(N \times T \times 2 \times R)$ per iteration. The convergence time is also influenced by the number of tasks and compute resources. IC-PCP and SCS being heuristic based, run much faster than our meta-heuristic based proposal. While IC-PCP and SCS have a polynomial time complexity, PSO has an exponential one. However, considering that our solution is designed to generate an offline schedule for a single workflow, the high time complexity of PSO is acceptable and the benefits in terms of better schedules outweigh this disadvantage.

3.7 Summary

This chapter presented a combined resource provisioning and scheduling strategy for executing scientific workflows on IaaS clouds. The scenario was modeled as an optimization problem which aims to minimize the overall execution cost while meeting a user defined deadline and was solved using the meta-heuristic optimization algorithm, PSO. The proposed approach incorporates basic IaaS cloud principles such as a pay-as-you-go model, heterogeneity, elasticity, and dynamicity of the resources. Furthermore,

our solution considers other characteristics typical of IaaS platforms such as performance variation and VM provisioning delays.

The simulation experiments conducted with four well-known workflows show that our solution has an overall better performance than the state-of-the-art algorithms, SCS and IC-PCP. In every case in which IC-PCP fails to meet the application's deadline, our approach succeeds. Furthermore, our heuristic is as successful in meeting deadlines as SCS, which is a dynamic algorithm. Also, in the best scenarios, when the proposed heuristic, SCS, and IC-PCP meet the deadlines, our solution is able to produce schedules with lower execution costs.

A drawback of the proposal presented in this chapter is its scalability in terms of the number of tasks in the workflow. Larger workflows imply larger search spaces that translate in longer convergence periods and lower-quality solutions. Also, the estimates used for the performance variation and the VM provisioning delays may not be accurate enough when deploying the algorithm in a real-cloud environment, hindering its performance. As a result, in the next chapter we propose an algorithm that addresses these drawbacks by using a more scalable strategy paired with a dynamic component that allows it to better adapt to the environment's uncertainties.

Chapter 4

A Responsive Knapsack-based Scheduling Algorithm

This chapter presents the Workflow Responsive resource Provisioning and Scheduling (WRPS) algorithm for scientific workflows in clouds. This strategy finds a balance between making dynamic decisions to respond to changes in the environment and planning ahead to produce better schedules. It aims to minimize the overall cost of the utilized infrastructure while meeting a user-defined deadline. The simulation results demonstrate it is scalable in terms of the number of tasks in the workflow, it is robust and responsive to the cloud performance variability, and it is capable of generating better quality solutions than state-of-the-art algorithms.

4.1 Introduction

There have been several works since the advent of cloud computing that aim to efficiently schedule scientific workflows. Many of them are dynamic algorithms capable of adapting to changes in the environment. Others are static algorithms that are sensitive to unexpected delays and runtime estimation of tasks but have the ability to perform workflow-level optimizations and compare various solutions before choosing the best-suited one. The work presented in this chapter aims to combine both approaches in order to find a better compromise between adaptability and the benefits of global optimization.

This chapter presents the Workflow Responsive resource Provisioning and Scheduling (WRPS) algorithm for scientific workflows in clouds. Its main strategy is based on a variation of the Unbounded Knapsack Problem (UKP) which is solved using dynamic

This chapter is derived from: **Maria A. Rodriguez** and Rajkumar Buyya. "A Responsive Knapsack-based Algorithm for Resource Provisioning and Scheduling of Scientific Workflows in Clouds." *In Proceedings of the 44th International Conference on Parallel Processing (ICPP)*, Pages 839-848, 2015.

programming. It uses a hybrid optimization strategy in which the optimal solutions to simplified versions of the problem are combined to produce the desired output. The algorithm finds a balance between making dynamic decisions to respond to changes in the environment and planning ahead to produce better schedules. It aims to minimize the overall cost of the utilized infrastructure while meeting a user-defined deadline and is capable of deciding what compute resources to use considering heterogeneous VM types. The simulation results demonstrate it is scalable in terms of the number of tasks in the workflow, it is robust and responsive to the cloud performance variability and it is capable of generating better quality solutions than the state-of-the-art algorithms.

The rest of this chapter is organized as follows. Section 4.2 presents the related work followed by the application and resource models in Section 4.3. Section 4.4 explains the proposed algorithm and Section 4.5 presents its evaluation. Finally, a summary is depicted in Section 4.6.

4.2 Related Work

There have been several works since the advent of cloud computing that aim to efficiently schedule scientific workflows. Many of them are dynamic algorithms capable of adapting to changes in the environment. An example is the Dynamic Provisioning Dynamic Scheduling (DPDS) algorithm [76] in which the number of leased VMs is adjusted, depending on how well the resources are being used by the application. Zhou et al. [124] also propose a dynamic approach designed to capture the dynamic nature of cloud environments from the performance and pricing point of view. Poola et al. [93] designed a fault tolerant dynamic algorithm based on the workflow's partial critical paths. The Partitioned Balanced Time Scheduling (PBTS) algorithm [35] estimates the optimal number of resources needed per billing period so that the application's deadline is met and the cost is minimized. Other dynamic algorithms include those developed by Xu et al. [113], Huu and Montagnat [61], and Oliveira et al. [41]. The main disadvantage of these approaches is their task-level optimization strategy, which is a trade-off for their adaptability to unexpected delays caused by changes in the environment or poor estimates.

On the other side of the spectrum are static algorithms. An example is the Static Provisioning Static Scheduling (SPSS) [76] algorithm. Designed to schedule a group of interrelated workflows (i.e. ensembles), it creates a provisioning and scheduling plan before running any task. Another example is the IaaS Cloud Partial Critical Path (IC-PCP) algorithm [22]. It is based on the workflow's partial critical paths and tries to minimize the execution cost while meeting a deadline constraint. Other examples include RDPS [112], DVFS-MODPSO [114] and EIPR [36]. In general, these algorithms are very sensitive to execution delays and runtime estimation of tasks, which is a trade-off for their ability to perform workflow-level optimizations and compare various solutions before choosing the best-suited one.

Contrary to fully dynamic or static approaches, our work combines both in order to find a better compromise between adaptability and the benefits of global optimization. SCS [77] is an example of an algorithm attempting to achieve this. It has a global optimization heuristic that allows it to find the optimal mapping of task to VM type. This mapping is then updated and used at runtime to scale the resource pool in or out and to schedule tasks as they become ready for execution. Our approach is different to SCS in that the static component does not analyze the entire workflow structure and instead optimizes the schedule of a subset of the workflow tasks. Moreover, our static component generates an actual schedule for these tasks rather than just selecting the optimal VM type.

4.3 Application and Resource Models

This work considers workflows modeled as DAGs as described in Section 1.1.2. Additionally, every workflow has a deadline δ_W, defined as a time limit for its execution. It is assumed that the size of a task S_t is measurable in Million of Instructions (MIs) and that, for every task, this information is provided as input to the scheduler.

A pay-as-you go model in which VMs are leased on-demand and are charged per time frame (i.e. billing periods) is considered. Any partial utilization results in the VM usage being rounded up to the nearest billing period. We model a VM type, VMT, in terms of

its processing capacity PC_{VMT} and cost per billing period C_{VMT}. The processing capacity of a VM determines how fast computations can be performed; we define it in terms of the number of instructions the CPU can process per second: Million of Instructions per Second (MIPS). We assume that for every VM type, its processing capacity in MIPS can be estimated based on the information offered by providers.

Workflows process data in the form of files. A common approach used to share these files among tasks is to use a peer-to-peer (P2P) model in which files are transferred directly from the VM running the parent task to the VM running the child task. Another technique is to use a global shared storage such as Amazon S3 as a file repository. In this case, tasks store their output in the global storage and retrieve their inputs from the same. We consider the latter model based on the advantages it offers. Firstly, the data is persisted and hence, can be used for recovery in case of failures. Secondly, it allows for asynchronous computation. In the P2P model, synchronous communication between tasks means that VMs must be kept running until all of the child tasks have received the corresponding data. With a shared storage on the contrary, the VM running the parent task can be released as soon as the data is persisted in the storage system. This may not only increase the resource utilization but also decrease the cost of VM leasing.

We assume data transfers in and out of the global storage system are free of charge, as is the case for products like Amazon S3, Google Cloud Storage and Rackspace Block Storage. As for the actual data storage, most cloud providers charge based on the amount of data being stored. We do not include this cost in the total cost calculation of neither our implementation nor the implementation of the algorithms used for comparison in the experiments. The reason for this is to be able to compare our approach with others designed to transfer files in a P2P fashion. Furthermore, regardless of the algorithm, the amount of stored data for a given workflow is most likely the same in every case or it is similar enough that it does not result in a difference in cost.

We acknowledge the existence of VM provisioning and deprovisioning delays [86] and assume that the CPU performance of VMs is not stable [99, 123]. Instead, it varies over time with its maximum achievable value being the CPU capacity advertised by the provider. We also assume network congestion causes a variation in data transfer

times [63, 99]. That is, the bandwidth assigned to a transfer depends on the current contention for the network link being used. Finally, we assume a global storage system with an unlimited storage capacity. The rates at which it is capable of reading and writing data vary based on how many processes are currently writing or reading data from the system.

The total processing time of a task t on a VM of type VMT, PT_t^{VMT}, is defined as the sum of its execution time and the time it takes to read the input files from the storage and write the generated output files to it. Note that whenever a parent and a child task are running in the same VM, there is no need to read the child's input file from the storage.

4.4 The WRPS Algorithm

This section provides the reasoning behind the main WRPS heuristics as well as a detailed explanation of the algorithm.

4.4.1 Overview and Motivation

WRPS has two main components, a dynamic and a static one. The dynamicity of the algorithm lies in the fact that the scheduling decisions are made at runtime, every time tasks are released into an execution queue. This allows it to adapt to any unexpected delays caused by poor estimates (of task sizes for example) or by environmental changes such as performance variation, network congestion, and VM provisioning and deprovisioning delays. The static component expands the ability of the algorithm from making decisions based on a single task (the next one in the scheduling queue) to making decisions based on a group of tasks. The purpose is to find a balance between the local knowledge of dynamic algorithms and the global knowledge of static ones. We achieve this by introducing the concept of *pipeline* and by statically scheduling all of the tasks in the scheduling queue at once. In this way, the algorithm is capable of making better optimization decisions and finding better quality schedules.

A *pipeline* is a common topological structure in scientific workflows and is simply a group of tasks with a one-to-one, sequential relationship between them. Formally, a

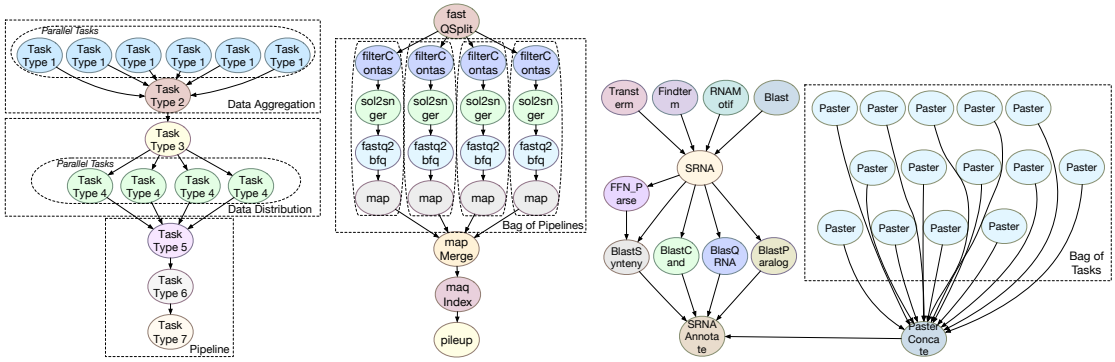


Figure 4.1: Examples of scientific workflows. (a) Example of bags of tasks and three different topological structures found in workflows: data aggregation, data distribution and pipelines. (b) An example of a bag of pipelines in the Epigenomics workflow. (c) Example of a bag of tasks in the SIPHT workflow.

pipeline P is defined as a set of tasks $T_p = \{t_1, t_2, \ldots, t_n\}$ where $n \geq 2$ and there is an edge $e_{i,i+1} = (t_i, t_{i+1})$ between task t_i and task t_{i+1}. In other words t_1 is the parent task of t_2, t_2 the parent task of t_3, and so on. The first task in a pipeline may have more than one parent but it must only have one child task. All other tasks must have a single parent (the previous pipeline task) and one child (the next pipeline task). A pipeline is associated with a deadline δ_P which is equal to the deadline of the last task in the sequence. Figure 4.1a shows an example of a pipeline.

By identifying pipelines in a workflow, we can easily expand the view from a single task to a set of tasks that can be scheduled more efficiently as a group rather than on their own. To avoid communication and processing overheads as well as VM provisioning and deprovisioning delays, tasks in a pipeline are clustered together and are always assigned to run on the same VM. The reasons are twofold. Firstly, the tasks are sequential and are required to run one after the other. There is no benefit in terms of parallelization on assigning them to different VMs. Secondly, the output file of a task becomes the input file of the next one, by running on the same VM, we avoid the cost and time of transferring these files in and out of the global storage.

The strategy used to optimize the scheduling of queued tasks is derived from the topological features of scientific workflows. Aside from sequential tasks (pipelines), a

workflow also has parallel structures composed of tasks with no direct dependencies between them. These tasks can run simultaneously and are generally found whenever data distribution or aggregation takes place. In data distribution [30], a tasks output is distributed to multiple tasks for processing. In data aggregation [30] the output of multiple tasks is aggregated, or processed, by a single task. Figure 4.1a shows an example of each of these structures.

The parallel tasks in these structures can be either homogeneous (same type) or heterogeneous (different types). The case in which the tasks are homogenous is common in scientific workflows; examples of well-known applications with this characteristic are Epigenomics, SIPHT, LIGO, Montage, and CyberShake. Based on this, we device a strategy to efficiently schedule homogeneous, parallel, tasks that are of the same size (MIs) and at the same level in the DAG. When using a level-based deadline distribution heuristic, these tasks will also have the same deadline. As an example, consider the data aggregation case. All the parallel tasks have to finish running before the aggregation task can start, therefore they can be assigned the same deadline which would be equal to the time the aggregation task is due to start. Note that the case in which the tasks are heterogeneous and at different levels in the workflow is uncommon but yet possible. An example is the data distribution of the $SRNA$ task in the SIPHT workflow, shown in Figure 4.1c. Also, there are other scenarios aside from distribution and aggregation where parallel tasks with the same properties can be found, however we focus on these as means for illustrating the motivation behind our scheduling policy.

The main static scheduling policy of WRPS consists then on grouping tasks of the same type and with the same deadline into bags. Two sample bags can be seen in Figure 4.1a, the first one is composed of all tasks of *type* 1 and the second one of all tasks of *type* 4. Scheduling these bags of tasks is much simpler than scheduling a workflow. There are no dependencies, the tasks are homogenous, and have to finish at the same time. We model the problem of running these tasks before their deadline and with minimum cost as a variation of the unbounded knapsack problem and find an optimal solution using dynamic programming. The same concept is applied to pipelines, they are grouped into bags according to their characteristics and scheduled in the same way as bags of tasks

are. An example of bag of pipelines is depicted in Figure 4.1b.

We have therefore designed an algorithm which is dynamic to a certain extent in order to adapt to unexpected delays product of the unpredictability of cloud environments but that also has a static component that enables it to generate better quality schedules and meet deadlines at lower costs. Moreover, it combines a heuristic-based approach with dynamic programming in order to be able to process large-scale workflows in an efficient and scalable manner. The details of WRPS are presented in Section 4.4.3.

4.4.2 The Unbounded Knapsack Problem

The knapsack problem is a combinatorial optimization problem. Its name derives from the problem faced when trying to select the most valuable items to pack in a fixed-size knapsack. Given a set of n items of different types, each item type $1 \leq i \leq n$ with a corresponding weight w_i and value v_i, the goal is to determine the number and type of items to pack so that the knapsack weight limit, W, is not exceeded and the total value of the items is the largest possible. The unbounded knapsack problem (UKP) is a variation in which unlimited quantities of each item type are assumed.

Let $x_i \geq 0$ be the number of items of type i to be placed in the bag. Then the unbounded knapsack problem can be defined as

$$\text{maximize} \sum_{i=1}^{n} v_i x_i \quad \text{subject to} \sum_{i=1}^{n} w_i x_i \leq W$$

UKP is a classic NP-hard problem [80]. A popular approach to solving it optimally is by using dynamic programming. UKP can be solved using this method by considering knapsacks of lesser capacities as sub problems and storing the best value of each capacity. Let $w_i > 0 \, \forall \, 1 \leq i \leq n$, then for each $w_i \leq W$ we can define a vector M where $m[w_i]$ is the maximum value that can be obtained with a weight less than or equal to w_i. In this way,

$$m[0] = 0$$

$$m[w_i] = \max_{w_j \leq w_i} (v_j + m[w_i - w_j])$$

The time complexity of this solution is $O(nW)$ as computing each $m[w_i]$ involves examining n items and there are W values of $m[w]$ to calculate. This running time is pseudo-polynomial as it grows exponentially with the length (the number of bits) of W. Yet, there are several algorithms designed to efficiently solve UKP. An example is the EDUK [23] algorithm, it combines the concepts of dominance [54], periodicity [55], and monotonic standard recurrence [24]. Experiments performed by the authors of EDUK demonstrate the scalability of the algorithm. For instance, for $W > 2 \times 10^8$, $n = 10^5$, and items with weights in the $[1, 10^5]$ range, the average running time was found to be 0.150 seconds.

4.4.3 Algorithm

The first step in the execution of WRPS is to preprocess the DAG. This is done by identifying all the pipelines and by assigning a portion of the deadline δ_W to each task. To find the workflow pipelines, tasks are first sorted according to their topological order, in this way we ensure parent tasks are processed before child tasks and hence we preserve the data dependencies. A pipeline is recursively built by adding one task at a time based on the following logic. For each sorted task that has not been processed, the algorithm recursively tries to build a pipeline that begins with that task. The base cases of the recursion happen when the processed task has no children, when it has more than one children or, when it has a single child with more than one parent task. The recursive stage occurs when the processed task has strictly one child which at the same time has strictly one parent (the processed task). In this case the task is added to the pipeline and the recursion continues with the child task. This is repeated until all the tasks have been processed. A more detailed explanation of the recursive part of the algorithm can be found in Algorithm 3.

For the deadline distribution, the algorithm first calculates the earliest finish time eft_t of all tasks defined as $eft_t = \max_{p \in t.parents}\{eft_p\} + PT_t^{VMT}$. The slowest VM type is used to calculate task runtimes. In this way, task runtimes can only improve if different VM types are used. However if using the slowest VM type means not being able to meet the deadline, then the next fastest VM type is used to estimate runtimes and so on. Afterwards, the amount of spare time available to distribute to the tasks is calculated. It

Algorithm 3 Find a pipeline recursively

```
1: procedure FINDPIPELINE(Task t, Pipeline p)
2:     children = t.children
3:     if p.tasks.size > 0 then
4:         if children.size > 1 or children.size = 0 then
5:             p.addTask(t)
6:         end ifreturn
7:         if children[0].parents.size() > 1 then
8:             p.addTask(t)
9:         end ifreturn
10:    end if
11:    p.addTask(t)
12:    findPipeline(children[0], p)
13: end procedure
```

is defined as $\delta_W - \max_{t \in W}\{eft_t\}$ and is partitioned based on the runtime of a task and its level in the workflow. In particular, a task's deadline δ_t is defined by the equation $\delta_t = t.start + PT_t^{VMT} + t.level.spare$ where $t.start$ is the task's start time and $t.level.spare$ is the spare time assigned to the task's level.

Once a DAG is preprocessed the scheduling of its tasks can begin. During the first iteration, all the entry tasks (those that have no parent tasks) in the workflow become ready for execution and are placed in a scheduling queue. WRPS then schedules them onto resources and as they finish their execution, their child tasks are then released onto the queue. This process continues until all of the workflow tasks have been successfully executed.

The first step in processing the scheduling queue consists in grouping the tasks into bags of tasks and bags of pipelines. A bag of tasks *bot* is defined as a group of one or more tasks T_{bot} that can run in parallel. All of the tasks in a bag share the same deadline δ_{bot}, are of the same type τ_{bot}, and are not part of a pipeline. Formally, $bot = (T_{bot}, \delta_{bot}, \tau_{bot})$. The definition of bag of pipelines *bop* is similar to that of *bot* but instead of a group of tasks, the bag contains one or more pipelines P_{bop} that are parallelizable. The tuple $bop = (P_{bop}, \delta_{bop})$, where δ_{bop} is the deadline the pipelines in the bag have in common, formally defines the concept.

To find the sets of bags of tasks $BoT = \{bot_1, bot_2, \ldots, bot_n\}$ and bags of pipelines $BoP = \{bop_1, bop_2, \ldots, bop_n\}$, each task in the queue is processed in the following way. If the task does not belong to a pipeline, then it is placed in the bot_i that contains tasks

of the same type and with the same deadline. If no such bot_i exists, a new bag is created and the task assigned to it. If, on the other hand, the task belongs to a pipeline, then the corresponding pipeline is placed in the bop_i which contains pipelines with the same deadline and types of tasks. If there is no bop_i with these characteristics, a new bag is created with its only element being the given pipeline.

Once the algorithm has identified the sets BoP and BoT, it can proceed to schedule them. Both types of bags are scheduled using the same policy, with the only difference being that a pipeline has more than one task that must be treated as a unit. The details of the scheduling policy are explained using the set BoT as an example. Note however that the same rules apply when scheduling BoP.

The following process is repeated for each $bot_i \in BoT$ which has more than one task; bags with a single element are treated as a special case and scheduled accordingly. First, WRPS tries to reduce the size of the bag and reuse already leased VMs by trying to fit as many tasks from the bag as possible in running, non-busy, VMs. The number of tasks assigned to a free VM is determined by the number of tasks that can finish before their deadline and before the next billing period of the VM. In this way, wastage of already-paid CPU cycles is reduced without affecting the makespan of the workflow. After this, a bag of tasks resource provisioning plan is created for the remaining tasks in the bag.

To generate an efficient resource provisioning plan, the algorithm must explore different solutions using different VM types and compare their associated costs. We achieve this and find the optimal combination of VMs that can finish the tasks in the bag in time with minimum cost by formulating the problem as a variation of UKP and solve it using dynamic programming. A knapsack item is defined by its type, weight, and value. For our problem, we define a scheduling knapsack item $SKI_j = (VMT_j, NT_j, C_j)$ where the item type corresponds to a VM type VMT_j, the weight is equal to the maximum number of tasks, NT_j, that can run in a VM of type VMT_j before their deadline, and the value is the associated cost C_j of running NT_j tasks in a VM of type VMT_j. Additionally, we assume there is an unlimited quantity of VMs of each type that can be potentially leased from the IaaS provider and define the knapsack weight limit as the number of tasks in the bag, that is, $W = |T_{bot}|$. The goal is to find a set of items SKI so that their combined

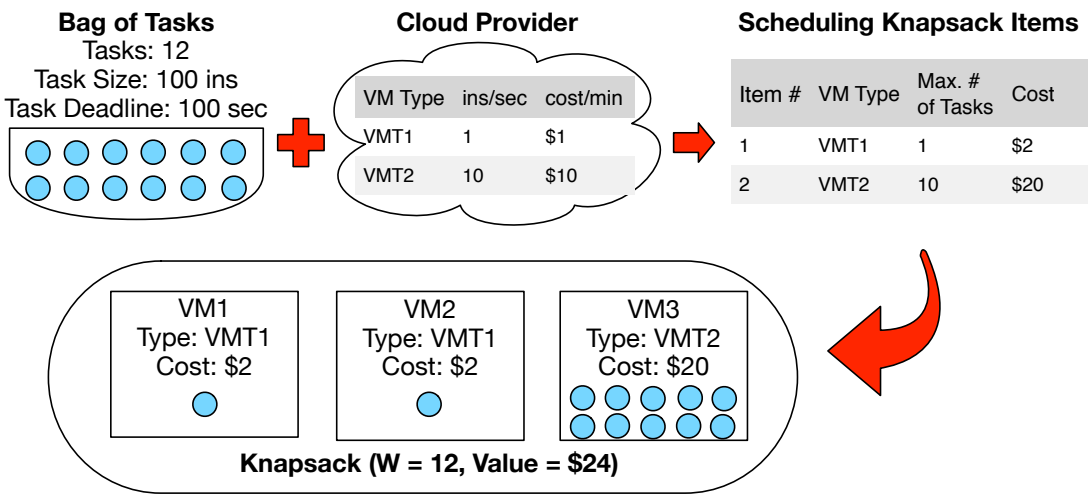


Figure 4.2: Example of a scheduling plan for a bag of tasks.

weight (i.e. the total number of tasks) is at least as large as the knapsack weight limit (i.e. the number of tasks in the bag) and whose combined value is minimum (i.e. the cost of running the tasks is minimum). Formally, the problem of scheduling a bag of tasks becomes

$$\text{minimize} \sum_{i=1}^{n} C_i \times x_i \quad \text{subject to} \sum_{i=1}^{n} NT_i \times x_i \geq |T_{bot}|$$

An example on how the scheduling plan is generated for a bag to tasks is shown in Figure 4.2. Assume two VM types, VMT_1 and VMT_2. The former can process 1 instruction per second at a cost of \$1 per minute and the latter 10 instructions per second at a cost of \$10 per minute. Consider a bag of 12 tasks, each task has a size of 100 instructions and a deadline of 100 seconds. The first knapsack item would be $SKI_1 = (VMT_1, 1, \$2)$. $NT_1 = 1$ as running a task in a VM that can process 1 instruction per second would take 100 seconds, with a deadline of 100 seconds, this means only one task can run on it before the deadline. $C_1 = \$2$ since the VM billing period is 60 seconds, this means that to use it for 100 seconds, two billing periods need to be paid for. Following the same reasoning, the second scheduling knapsack item would be $SKI_2 = (VMT_2, 10, \$20)$. The optimal combination of items to pack would then be 2 items of type SKI_1 and 1 item of type SKI_2. This means leasing 2 VMs of type VMT_1 and running 1 task on each and leasing 1 VM of type VMT_2 and running 10 tasks on it, for a total cost of \$24.

After solving the bag of tasks scheduling problem using dynamic programming we obtain a resource provisioning plan of the form $RP_i^{bot} = (VMT_i, numVM_i, NT_i)$ for each VM type. This indicates the number of VMs of type VMT_i to use ($numVM_i$) and the maximum number of tasks to run on each VM (NT_i). In rare cases in which the algorithm is running out of time and there are no VM types that can finish the tasks on time, a resource provisioning plan of the form $RP_{bot}^{fastest} = (VMT_{fastest}, W, 1)$ is created. This indicates that a VM of the fastest type must be leased for each task in the bag so that they run in parallel and finish as soon as possible.

For each RP_i^{bop} for which $numVM_i > 0$, WRPS first tries to find a VM of type VMT_i which has already been leased but that is not busy and is free to use. If such VM exists, then NT_i tasks from the bag are scheduled onto it. In this way we are able to save money by using time slots that have already been paid for and avoid risking long provisioning delays of newly leased VMs. If there is no free VM of the required type, a new one is provisioned and the specified number of tasks scheduled on to it.

We consider the case of a bag of tasks with a single task as a special one. Single tasks are scheduled on free VMs if they can finish the task on time and before their current billing period finishes. If there is no free VM that can achieve this, then a new VM of the type that is capable of finishing the task by its deadline at the cheapest cost is provisioned and the task scheduled to it. If no VM type can finish by the deadline, the fastest available VM type is used. The pseudo- code for the scheduling of BoT is shown in Algorithm 4.

WRPS continuously adjusts the task deadline distribution to reflect the actual finish time of tasks. If a task finishes earlier than expected, all of the remaining tasks will have more time to run and cost can be potentially reduced. In cases in which a task finishes later than expected, the deadline of all remaining tasks is updated to avoid delays that could lead to the overall deadline being missed. WRPS also has a rescheduling mechanism that enables it to deal with unexpected delays encountered while running a bag of tasks (or pipelines). Since multiple tasks are statically assigned to a VM, a delay in the execution of one task will have an impact on the actual finishing time of the other ones. To mitigate this effect, if a task belonging to a bag finishes after its deadline on VM_i, then the tasks in the execution queue of VM_i are analyzed in the following way. If all of the

Algorithm 4 BoT scheduling

Require: $W = (T, E)$, $t.deadline \forall t \in T$, BoT in queue
1: **procedure** SCHEDULEBOT
2: **for all** $bot \in BoT$ **do**
3: **if** $bot.tasks.size > 1$ **then**
4: $RP^{bot} = UKPBasedProvisioningPlan(bot)$
5: **for all** $RP_i^{bot} = (VMT_i, numVM_i, NT_i) \in RP^{bot}$ **do**
6: **for** $0 \leq k < numVM_i, k++$ **do**
7: $numTasks = \min_{NT_i, bot.size}$
8: $tasksToSchedule \leftarrow numTasks$ from bot
9: remove $numTasks$ from bot
10: $vm = findFreeVM(VMT_i)$
11: **if** $vm = null$ **then**
12: $vm = provisionNewVM(VMT_i)$
13: **end if**
14: $scheduleTasks(toSchedule, vm)$
15: **end for**
16: **end for**
17: **else**
18: $task = bot.taks[0]$
19: $vm = findFreeVMForTask(task, task.deadline)$;
20: **if** $vm = null$ **then**
21: $vm = provisionNewVM(VMT_i)$
22: **end if**
23: $scheduleTask(task, vm)$
24: **end if**
25: **end for**
26: **end procedure**

tasks remaining in the queue of VM_i can finish by their updated deadline then no action is taken. If VM_i cannot finish its queued tasks by their deadline, then the tasks are released back into the scheduling queue so that WRPS can schedule them again based on their updated deadline.

As mentioned earlier, the set of bags of pipelines BoP is scheduled following the same strategy as for BoT. Just as tasks, pipelines have a deadline and a size (the aggregated size of all the tasks in the pipeline). The only difference is that there is a group of tasks to run instead of a single one. Thus, we can apply the same scheduling heuristic and model the problem as a variation of UKP with a slight difference in the definition of a knapsack item. For the pipeline case, $SKI_j = (VMT_j, NP_j, C_j)$, where the weight of an item, NP_j, is equal to the maximum number of pipelines a VM type can finish before their deadline. The rescheduling strategy is also similar to that of tasks, except that whenever a task in a pipeline is delayed, the remaining tasks of the pipeline are left to finish in the VM while other pipelines are rescheduled according to their updated deadline. Once again, bags of

a single pipeline are treated as a special case and are scheduled onto free VMs if they can finish by the deadline or on a newly leased VM of the type that can finish them by the deadline at minimum cost.

Finally, the leasing period of VMs is managed by WRPS to avoid incurring in unnecessary costs. A VM is shutdown if its use is approaching the next billing period and there is no current task assigned to it. An estimate of the VM deprovisioning delay is used to ensure the VM shutdown request is sent early enough so that it stops being billed before the current billing period ends.

4.5 Performance Evaluation

The performance of WRPS was evaluated using four well-known workflows from various scientific areas. These are the Montage, Epigenomics, SIPHT, and LIGO from the astrophysics area. The structures of these workflows are presented in Chapter 1, Section 1.1.2. Each of them has different topological structures and different data and computational characteristics.

Two algorithms were used to evaluate the quality of the schedules produced by WRPS. The first one is the Static Provisioning Static Scheduling (SPSS) [76] algorithm which assigns sub-deadlines to tasks and schedules them onto existing or newly leased VMs so that cost is minimized. It was designed to schedule a group of interrelated workflows but it can easily be adapted to schedule a single one. It was chosen as it is a static algorithm capable of generating high-quality solutions. Its drawback is its inability to meet deadlines when unexpected delays occur. However, we are still interested in comparing WRPS to SPSS when both are able to meet the deadline constraints. Also, by comparing them, we are able to validate our solution's adaptability and demonstrate how when other algorithms fail to recover from unexpected delays WRPS succeeds in doing so.

The second algorithm is SCS [77], a state-of-the-art dynamic algorithm that has an auto-scaling mechanism that allocates and deallocates VMs based on the current status of tasks. It determines the most cost-efficient VM type for each task and creates a *load vector*. This load vector is updated every scheduling cycle and indicates how many VMs

Table 4.1: VM types based on Amazon's EC2 offerings.

Name	Memory	Google Compute Engine Units	Price per Minute
n1-standard-1	3.75GB	2.75	$0.00105
n1-standard-2	7.5GB	5.50	$0.0021
n1-standard-4	15GB	11	$0.0042
n1-standard-8	30GB	22	$0.0084

of each type are needed in order for the tasks to finish by their deadline with minimum cost. The purpose is to demonstrate how the static component of WRPS allows it to produce better quality schedules than SCS.

An IaaS provider offering a single data center and four types of VMs was modeled. The VM type configurations are based on Google Compute Engine offerings and are shown in Table 5.1. A VM billing period of 60 seconds was used, as offered by several providers such as Google Compute Engine and Microsoft Azure. For all VM types, the provisioning delay was set to 30 seconds [101] and the deprovisioning delay to 3 seconds [78]. CPU performance variation was modeled after the findings by Schad et al. [99]. The performance of a VM is degraded by at most 24% based on a normal distribution with a 12% mean and a 10% standard deviation. A network link's total available bandwidth is shared between all the transfers using the link at a given point in time. This bandwidth allocation was done using a progressive filling algorithm to model network congestion and data transfer time degradation. A global shared storage with a maximum reading and writing speeds was also modeled. The reading speed achievable by a given transfer is determined by the number of processes currently reading from the storage, the same rule applies for the writing speed. In this way, we simulate congestion when trying to access the storage system.

Workflows with approximately 1000 tasks were used for the evaluation. We acknowledge that the estimation of task sizes might not be 100% accurate and hence, introduce in our simulation a variation of $\pm 10\%$ to the size of each task based on a normal distribution. The experiments were conducted using four different deadlines, δ_{W1} being the strictest one and δ_{W4} being the most relaxed one. For each workflow, δ_{W1} is equal to the time it takes to execute the tasks in the critical path plus the time it takes to trans-

fer all the input files into the storage and the output files out of it. The other deadlines are based on δ_{W1} and an interval size $\delta_{int} = \delta_{W1}/2$: $\delta_{W2} = \delta_{W1} + \delta_{int}$, $\delta_{W3} = \delta_{W2} + \delta_{int}$, and $\delta_{W4} = \delta_{W3} + \delta_{int}$. The results displayed are the average obtained after running each experiment 20 times.

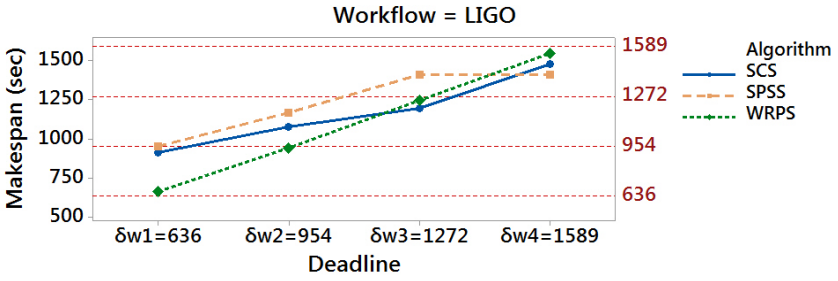
4.5.1 Results and Analysis

Makespan and Cost Evaluation

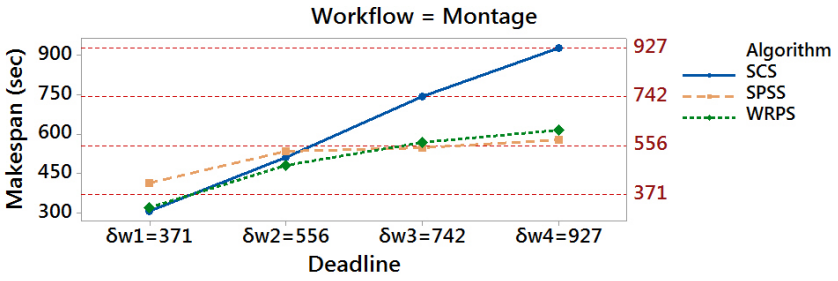
The average makespan and cost obtained for each of the workflows is depicted in Figure 4.3. The reference lines in the makespan line plots correspond to the four deadline values used for each workflow. Evaluating the makespan and cost with regards to this value is essential as the main objective of all the algorithms is to finish before the given deadline. The dashed bars in the cost bar charts indicate that the algorithm failed to meet the corresponding deadline.

For the LIGO workflow, the first deadline, δ_{W1}, proves to be to tight for any of the algorithms to finish on time. However, the difference between the makespan obtained by WRPS and the deadline is marginal. Additionally, WRPS generates the cheapest schedule in this case. The second deadline is still not relaxed enough for SCS or SPSS to achieve their goal, however, WRPS demonstrates its adaptability and ability to generate cheap schedules by being the only algorithm to finish its execution before the deadline and at the lowest cost. For the remaining deadlines, δ_{W3} and δ_{W4}, both SCS and WRPS are capable of meeting the constraint, in both cases SCS has a slightly lower makespan but WRPS has a lower cost. SPSS is only capable of meeting the most relaxed deadline, and in this case, WRPS outperforms it in terms of execution cost. Overall, WRPS meets the most deadlines and in all of the cases achieves better quality schedules with the cheapest costs.

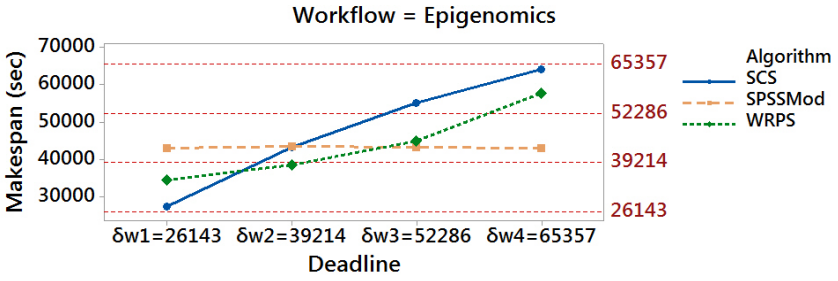
In the case of the Montage application, both SCS and WRPS meet all of the deadlines with WRPS consistently generating cheaper schedules. SPSS fails to meet the tightest deadline but succeeds in meeting δ_{W2}, δ_{W3}, and δ_{W4}. Its success in meeting 75% of the deadlines may be explained by the fact that most of the tasks in the Montage application



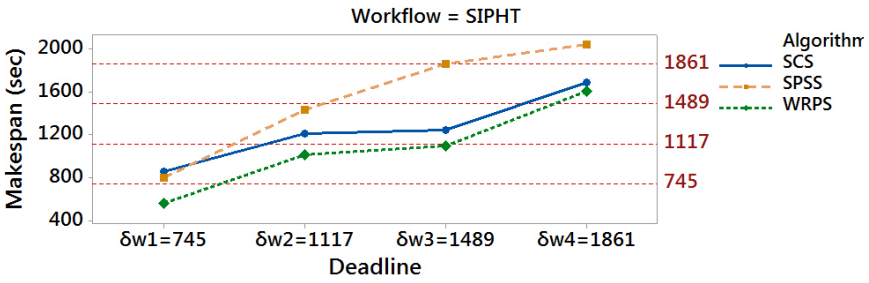
(a)



(b)

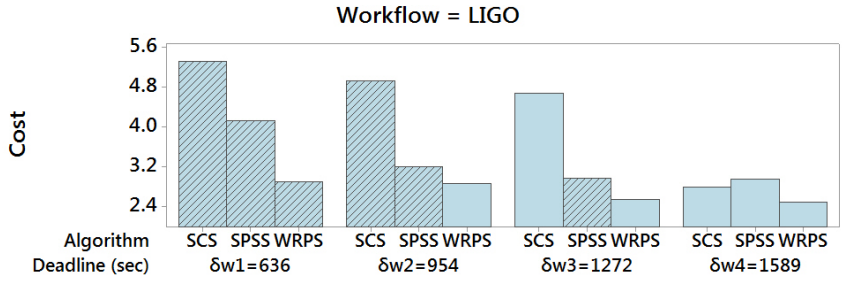


(c)

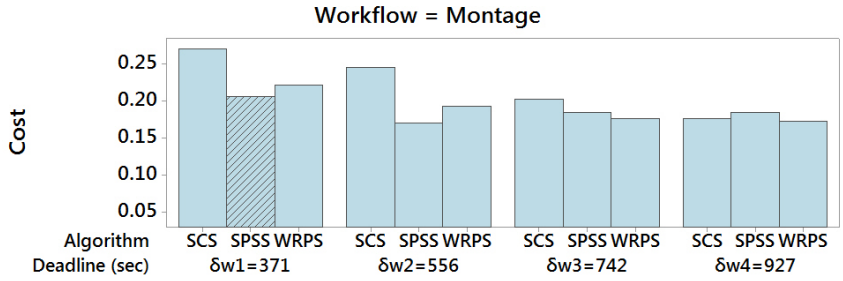


(d)

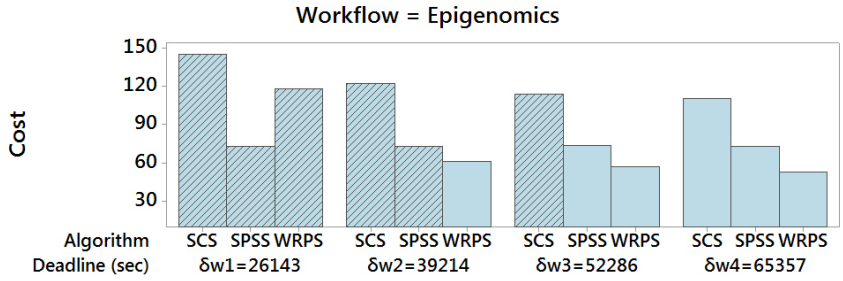
Figure 4.3: Makespan experiment results for each workflow application. The reference lines in the line plots indicate the four deadline values. (a) LIGO. (b) Montage. (c) Epigenomics. (d) SIPHT.



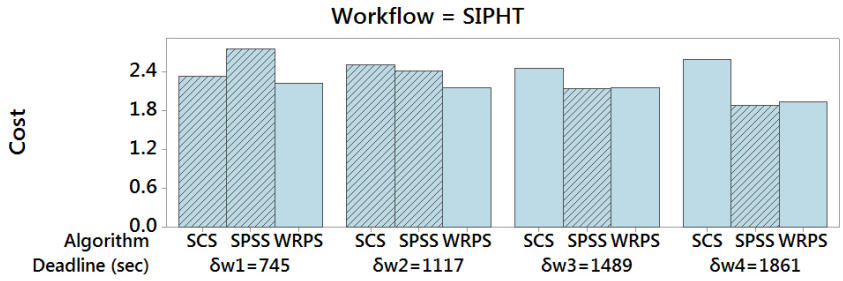
(a)



(b)



(c)



(d)

Figure 4.4: Cost experiment results for each workflow application. The dashed bars indicate the deadline was not met for the corresponding deadline value. (a) LIGO. (b) Montage. (c) Epigenomics. (d) SIPHT.

are considered small and require low CPU utilization, leading to a potentially low CPU performance variation impact on the static schedule. In the case of δ_{W2}, SPSS proves its capability of generating cheap schedules by performing better than its counterparts; although WRPS has a lower makespan in this case, its cost is slightly higher than that of SPSS. For δ_{W3} and δ_{W4} however, WRPS succeeds in generating cheaper solutions than its static counterpart.

The three algorithms fail to meet δ_{W1} of the Epigenomics workflow, the closest makespan to the deadline is achieved by SCS, followed by WRPS and finally SPSS. WRPS is the only algorithm capable of meeting δ_{W2} and still achieves the lowest cost. The third deadline constraint is met by SPSS and WRPS, with WRPS once again outperforming SPSS in terms of cost. Finally, as the deadlines become relaxed enough, the three algorithms succeed in meeting the deadline and WRPS does it with the lowest cost. The high deadline miss percentage of SCS and SPSS in this case is due to the high-CPU nature of the Epigenomics tasks, meaning that CPU performance degradation will have a significant impact on the execution time of tasks causing unexpected delays.

SIPHT is an interesting application to evaluate our algorithm due to its topological features. As mentioned earlier, it has data distribution and aggregation structures in which the parallel tasks differ on their type. The results demonstrate that even in cases like this, our algorithm remains responsive and efficient. WRPS succeeds in meeting the four deadlines with the lowest makespan in all cases and with the lowest cost amongst the algorithms that meet the constraint. The large number of files that need to be transferred when running this workflow lead to SPSS struggling to recover from the lower data transfer rates due to network congestion and hence failing to meet the four deadlines. Even SCS fails to adapt to these delays on time and fails to meet the three tightest deadlines.

Overall, WRPS is the most successful algorithm in meeting deadlines. On average, it succeeds in meeting the constraint in 87.5% of the cases while SCS succeeds in 56.25% of the cases and SPSS on 37.5%. These results are inline with what was expected of each algorithm. The static approach is not very efficient in meeting the deadlines whereas the dynamism in WRPS and SCS allows them to accomplish their goal more often. The ex-

periments also demonstrate the efficiency of WRPS in terms of its ability to generate low cost solutions. It outperforms SCS an SPSS as in all of the scenarios except one (Montage workflow, δ_{W2}); WRPS achieves the lowest cost when compared to those algorithms that met the deadline. Another desirable characteristic of WRPS that can be observed from the experiment results is its ability to consistently increase the time it takes to run the workflow and reduce the cost as the deadline becomes more relaxed. The importance of this relies in the fact that many users are willing to trade-off execution time for lower costs while others are willing to pay higher costs for faster executions. The algorithm needs to behave within this logic in order for the deadline value given by users to be meaningful.

Network Usage Evaluation

Network links are well-known bottlenecks in cloud environments. For instance, Jackson et al. [63] report a data transfer time variation of up to 65% in the Amazon EC2 cloud. Hence, as means of reducing the sources of unpredictability and improving the performance of workflow applications, it is important for scheduling algorithms to try to reduce the amount of data transferred through the cloud network infrastructure. In this section, we evaluate the number of files read from the global storage system by each of the algorithms. Remember that a task does not need to read from the storage system whenever the input files required by it are already available in the VM where it is running.

The bar charts in Figure 4.5 show the average number of files read across the four deadlines for each workflow and algorithm. The reference line indicates the total number of input files that are required by the workflow tasks. By scheduling pipelines in a single VM and by running as many tasks or pipelines from the same bag in a single VM, WRPS is successful in reducing by 50% or more the number of files read from the storage. In fact, WRPS reads the least amount of files when compared to SCS and SPSS in the cases of the LIGO and Epigenomics workflow. The files read from the storage are reduced by 58% in the LIGO case and by 75% in the Epigenomics case. For the Montage workflow, SCS and WRPS achieve a similar performance and reduce the number of files by approximately 50%. Finally, even though reduced by 23%, WRPS is not as successful in reducing the

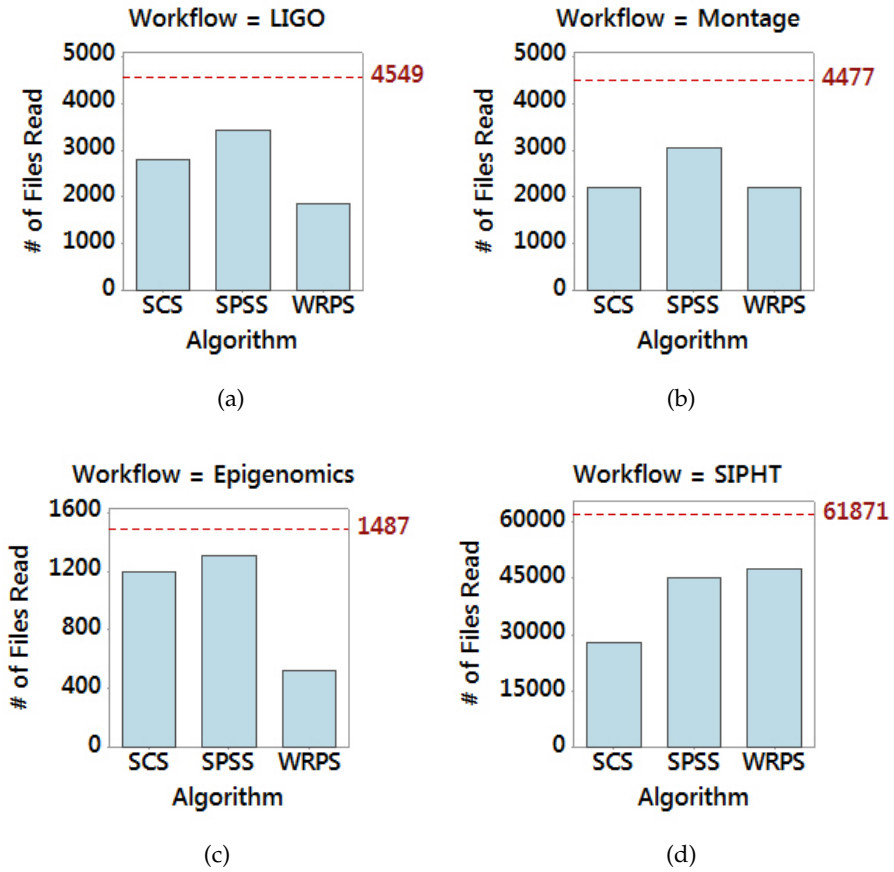


Figure 4.5: Average number of actual files read from the global storage system by each algorithm for each workflow. The reference lines in the bar charts indicate the total number of files required as input by the given workflow. (a) LIGO. (b) Montage. (c) Epigenomics. (d) SIPHT.

number of files as SCS and SPSS for the SIPHT workflow. The lack of pipelines and bags of tasks in the SIPHT application are the main cause for this and as a future work we would like to explore and develop new heuristics so that WRPS is capable of scheduling these type of workflows more efficiently.

4.6 Summary

WRPS, a responsive resource provisioning and scheduling algorithm for scientific workflows in clouds capable of generating high quality schedules was presented. It has as

objectives minimizing the overall cost of using the cloud infrastructure while meeting a user-defined deadline. The algorithm is dynamic to a certain extent to respond to unexpected delays and environmental dynamics common in cloud computing. It also has a static component that allows it to find the optimal schedule for a group of workflow tasks, consequently improving the quality of the schedules it generates. By reducing the workflow into bags homogeneous tasks and pipelines that share a deadline, we are able to model their scheduling as a variation of the unbounded knapsack problem and solve it in pseudo-polynomial time using dynamic programming.

The simulation experiments show that our solution has an overall better performance than the state-of-the-art algorithms. It is successful in meeting deadlines under unpredictable situations involving performance variation, network congestion and inaccurate task size estimations. It achieves this at low costs, even lower than the fully static approaches which have the ability of using the entire workflow structure, comparing various solutions, and choosing the best one before the workflow execution.

This chapter presented a strategy that was successful in meeting deadlines and minimizing the infrastructure cost. While this is the most commonly explored scheduling objective in cloud workflow scheduling, the next chapter focuses in a less common yet important requirement, meeting a budget constraint while minimizing the total execution time of the workflow.

Chapter 5

A Mixed Integer Linear Programming Based Scheduling Algorithm

The majority of workflow scheduling algorithms tailored for clouds focus on coarse-grained billing periods that are much larger than the average execution time of individual tasks. Instead, this work focuses on emerging finer-grained pricing schemes (e.g. per-minute billing) that provide users with more flexibility and the ability to reduce the inherent wastage that results from coarser-grained ones. This chapter proposes a scheduling algorithm whose objective is to optimize a workflow's execution time under a budget constraint; QoS requirement that has been overlooked in favor of optimizing cost under a deadline constraint. This strategy addresses fundamental challenges of clouds such as resource elasticity, abundance, and heterogeneity, as well as performance variation and virtual machine provisioning latency. The simulation results demonstrate the algorithms responsiveness and ability to generate high-quality schedules that comply with the budget constraint while achieving lower makespans when compared to state-of-the-art algorithms.

5.1 Introduction

The adoption of cloud computing for scientific workflow deployment has led to extensive research on designing efficient scheduling algorithms capable of elastically utilizing VMs. This ability to modify the underlying execution environment is a powerful tool that allows algorithms to scale the number of resources to achieve both, performance and cost efficiency. However, this flexibility is limited when coarse-grained billing periods, such as hourly billing, are enforced by providers. As the average execution time of workflow

This chapter is derived from: **Maria A. Rodriguez** and Rajkumar Buyya. "Budget-Driven Resource Provisioning and Scheduling of Scientific Workflow in IaaS Clouds with Fine-Grained Billing Periods." *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2016 (under second review).

tasks is considerably smaller than a billing cycle, algorithms are obliged to focus on maximizing the usage of time slots in leased VMs as a cost-controlling mechanism. This not only restricts the degree of scalability in terms of resources but also leads to inevitable wastage as idle time slots will naturally occur due to performance restrictions and dependencies between tasks.

This coarse-grained billing period model is assumed by the majority of existing algorithms dealing with resource provisioning and scheduling in clouds. Instead, this work targets emerging pricing models that are designed to give users more flexibility and reduce wastage by offering fine-grained charging periods, such as per-minute billing. Under this model, algorithms can more freely take advantage of the cloud's resource abundance and as a result, more aggressive dynamic scaling policies are needed. The potential of using a different VM for each workflow task emphasizes the importance of making accurate resource provisioning decisions that are not only guided by the scheduling objectives, but also by characteristics inherent to clouds such as resource performance variation and a non-negligible VM start-up (i.e. provisioning) delay.

The utility-based pricing model offered by cloud providers means that finding a trade-off between cost and performance is a common denominator for scheduling algorithms. This is done mostly by trying to minimize the total infrastructure cost while meeting a time constraint, or deadline. Only a small fraction of techniques focus on scheduling under budget constraints. Most of them are based on computationally intensive meta-heuristic techniques that do not scale well with the number of tasks in the workflow and that produce a static schedule unable to adapt to the inherent dynamicity of cloud environments. Others include a deadline constraint which guides the optimization process and the budget is only taken into consideration when deciding the feasibility of a potential schedule. Contrary to this, we propose a budget-driven algorithm whose objective is to optimize the way in which the budget is spent so that the makespan of the application is minimized. It includes a budget distribution strategy that guides the individual expenditure on tasks and makes dynamic resource provisioning and scheduling decisions to adapt to changes in the environment. Also, to improve the quality of the optimization decisions made, two different mathematical models are proposed to estimate the opti-

mal resource capacity for parallel tasks derived from data distribution structures. Our simulation results demonstrate that our algorithm is scalable, adaptable and capable of generating efficient schedules with high quality in terms of meeting the budget constraint with lower makespans when compared to state-of-the-art algorithms.

The rest of this chapter is organized as follows. Section 5.2 presents the related work followed by the application and resource models in Section 5.3. Section 5.4 explains the proposed resource provisioning and scheduling algorithm. Section 5.5 presents the experimental setup and the evaluation of our solution. Finally, a summary is outlined in Section 5.6.

5.2 Related Work

Our work is related to algorithms for workflow scheduling in IaaS clouds capable of elastically scaling resources. The Partitioned Balanced Time Scheduling (PBTS) algorithm [35] divides the execution of the workflow into time partitions the size of the billing period. Then, it optimizes the schedule of the tasks in each partition by estimating the minimum number of homogeneous VMs required to finish them on time. It differs from our solution as we do not assume tasks can finish within one billing period and we consider VMs with different characteristics and prices. SCS [77] and Dyna [124] are other algorithms with an auto-scaling mechanism to dynamically allocate and deallocate VMs based on the current status of tasks. They differ from our proposal as they consider dynamic and unpredictable workloads of workflows and assume an hourly billing period. Designed to schedule a single workflow while dynamically making resource provisioning decisions are the heuristics proposed by Poola et al. [93] and Wang et al. [110], however, they also assume a pricing model based on an hourly rate. Furthermore, all of the mentioned algorithms have different objectives to our solution as they aim to minimize the execution cost while meeting a deadline constraint.

The Dynamic Provisioning Dynamic Scheduling (DPDS) algorithm [76] is another strategy that dynamically scales the VM pool and was designed to schedule a group of interrelated workflows (i.e. ensembles) under budget and deadline constraints. It does

so by creating an initial pool of homogeneous VMs with as many resources as allowed by the budget and updating it at runtime based on a utilization measure estimated using the number of busy and idle VMs. DPDS is different from our work in several aspects; mainly, its provisioning strategy is suited for coarse grained periods and we focus on scheduling a single workflow without considering a deadline constraint.

Only a few algorithms targeting IaaS clouds consider a budget constraint as part of their objectives. The Static Provisioning Static Scheduling (SPSS) algorithm [76] considers the scheduling of workflow ensembles under deadline and budget constraints. The deadline guides the scheduling process of individual workflows by assigning sub-deadlines to individual tasks. These are then assigned to VMs that can complete their execution on time with minimum cost. This process is repeated until all the workflows have been scheduled or the budget has been reached. Pietri et al. [91] proposed SPSS-EB, an algorithm based on SPSS and concerned with meeting energy and budget constraints. The execution of the workflow is planned by scheduling each task so that the total energy consumed is minimum, a plan is then accepted an executed only if the energy and budget constraints are met. Our work is different from these approaches in two aspects. Firstly, they consider a second constraint as part of the scheduling objectives. Moreover, they make static provisioning and scheduling decisions and do not account for VM provisioning and deprovisioning delays or performance degradation.

A dynamic budget-aware algorithm capable of making auto-scaling and scheduling decisions to minimize the application's makespan is presented by Mao and Humphrey [79]. However, they consider an hourly budget instead of a budget constraint for the entire workflow execution and aim to optimize the execution of a continuous workload of workflows. Similar to our work, the Critical-Greedy [111] algorithm considers a financial constraint while minimizing the end-to-end delay of the workflow execution. However, it does not include billing periods on its cost estimates and hence considers VMs priced per unit of time. Also, the output of the algorithm is a task to VM type mapping and the authors do not propose a strategy to assign the tasks to actual VMs while considering their startup time and performance degradation.

The Revised Discrete Particle Swarm Optimization (RDPSO) algorithm [112] uses a

technique based on particle swarm optimization to produce a near-optimal schedule that minimizes either cost or time and meets constraints such as deadline and budget. In contrast to our approach, the algorithm is based on a computationally intensive meta-heuristic technique that produces a globally optimized schedule. ScaleStar [121] is another algorithm that considers a budget constraint. Similarly to our approach, it aims to minimize the makespan of the workflow. However, although it explicitly considers billing periods of one hour, their total execution cost calculation does not consider the fact that partial utilization of VMs is charged as full time utilization. These algorithms also differ from our solution in that they produce static schedules and assume a finite set of VMs is available as input.

Malawski et al. [75] present a mathematical model that optimizes the cost of scheduling workflows under a deadline constraint. As opposed to our algorithm, it considers a multi-cloud environment where each provider offers a limited number of VMs billed per hour. They group tasks on each level based on their computational cost and input/output data and schedule these groups instead of single tasks. They achieve this by modeling the problem as a mixed integer program, which differs from ours as it generates a static schedule for the entire workflow as opposed to a resource provisioning plan for a subset of the workflow tasks. Genez et al. [52] also formulate the problem of scheduling a workflow on a set of subscription-based and on-demand instances as an integer program. However, the output of their model is a static schedule indicating the mapping of tasks to VMs as well as the time when they are meant to start their execution. This limits the scalability of the algorithm as the number of variables and constraints in the formulation increases rapidly with the number of cloud providers, maximum number of VMs that can be leased from each provider, and the number of tasks in the DAG.

5.3 Application and Resource Models

This work is designed to schedule scientific workflows with a large number of tasks that are computationally and data intensive. Specifically, it considers workflows modeled as Directed Acyclic Graphs (DAGs) as described in Section 1.1.2 of Chapter 1. The amount of

input data required by task t is defined as d_t^{in} and the amount of output data it produces as d_t^{out}.

We define the sharing of data between tasks to take place via a global storage system such as Amazon S3 [2]. In this way, tasks store their output in the global storage and retrieve their inputs from the same. As stated in Section 2.2.3 of Chapter 2, this model has two main advantages. Firstly, the data is persisted and hence, can be used for recovery in case of failures. Moreover, unlike a peer to peer model where VMs need to remain active until all of the child tasks have received the corresponding data, a shared storage enables asynchronous computation as the VM running the parent task can be released as soon as the data is persisted in the storage system.

We assume a pay-as-you go model where VMs are leased on-demand and are charged per billing period τ. We acknowledge that any partial utilization results in the VM usage being rounded up to the nearest billing period. Nonetheless, we focus on fine-grained billing periods such as one minute as offered by providers such as Google Compute Engine [9] and Microsoft Azure [84]. We consider a single cloud provider and a single data center or availability zone. In this way, network delays are reduced and intermediate data transfer fees eliminated. Finally, we impose no limit on the number of VMs that can be leased from the provider.

The IaaS provider offers a range of VM types $VMT = \{vmt_1, vmt_2, \ldots, vmt_n\}$ with different prices and configurations. The execution time, E_t^{vmt}, of each task on every VM type is available to the scheduler. Different performance estimation methods can be used to obtain this value. The simplest approach is to calculate it based on an estimate of the size of the task and the CPU capacity of the VM type. Another valid method could be based on the results obtained after profiling the tasks on a baseline machine. This topic is out of the scope of this chapter, however, note that the proposed solution acknowledges that this value is simply an estimate and does not rely on it being one hundred percent accurate to achieve its objectives.

VM types are also defined in terms of their cost per billing period c_{vmt} and bandwidth capacity b_{vmt}. An average measure of their provisioning $prov_{vmt}$ delay is also included as part of their definition. We assume a global storage system with an unlimited storage

capacity. The rates at which it is capable of reading and writing are GS_r and GS_w respectively. The time it takes to transfer and write d output data from a VM of type vmt into the storage is defined as

$$N^{out}_{d,vmt} = (d/b_{vmt}) + (d/GS_w). \tag{5.1}$$

Similarly, the time it takes to transfer and read a task's output data from the storage to a VM of type vmt is defined as

$$N^{in}_{d,vmt} = (d/b_{vmt}) + (d/GS_r). \tag{5.2}$$

The total processing time of a task t on a VM of type vmt, PT^{vmt}_t, is calculated as the sum of its execution time and the time it takes to read the required n_{in} input files from the storage and write n_{out} output files to it. Notice that there is no need to read an input file whenever it is already available in the VM were the task will execute. This occurs when parent and child tasks run on the same machine.

$$P^{vmt}_t = E^{vmt}_t + (\sum_{i=1}^{n_{in}} N^{in}_{d_i,vmt}) + (\sum_{i=1}^{n_{out}} N^{out}_{d_i,vmt}) \tag{5.3}$$

The cost of using a resource r_{vmt} of type vmt for $lease_r$ time units is defined as

$$C_{r_{vmt}} = \lceil (prov_{vmt} + lease_r)/\tau \rceil * c_{vmt} \tag{5.4}$$

Finally, we assume data transfers in and out of the global storage system are free of charge, as is the case for products like Amazon S3 [2], Google Cloud Storage [8] and Rackspace Block Storage [16]. As for the actual data storage, most cloud providers charge based on the amount of data being stored. We do not include this cost in the total cost calculation of neither our implementation nor the implementation of the algorithms used for comparison in the experiments. The reason for this is to be able to compare our approach with others designed to transfer files in a P2P fashion. Furthermore, regardless of the algorithm, the amount of stored data for a given workflow is most likely the same in every case or it is similar enough that it does not result in a difference in cost.

The scheduling problem addressed in this chapter can then be defined as dynamically scaling a set of resources R and allocating each task to a given resource so that the total cost,

$$C_{total} = \sum_{i=1}^{|R|} C_{r_i^{vmt}}, \tag{5.5}$$

is less than or equal to the workflow's budget β while minimizing the makespan of the application.

5.4 Proposed Approach

We propose a budget-driven algorithm, called BAGS, in which different resource provisioning and scheduling strategies are used for different topological structures. This is done by partitioning the DAG into bags of tasks (BoTs) containing a group of parallel homogeneous tasks, parallel heterogeneous tasks, or a single task. This strategy derives from the observation that large groups of parallel tasks is a common occurrence in scientific workflows and as a result, we aim to optimize their execution as an attempt to generate higher quality schedules while maintaining the dynamicity and adaptability of the algorithm to the underlying cloud environment.

More specifically, our strategy identifies sets of tasks that are at the same level in the DAG and are guaranteed to be ready for execution at the same time. This may happen when they are at the entry level of the workflow and have no parent tasks dictating the time of their execution or when they share a single parent task which distributes data to them. Figure 5.1 shows examples of these BoTs in five well-known scientific workflows. Any task that does not meet any of the above requirements is categorized as a single task, or as we will refer to from now on, a bag with a single task. Each BoT is then scheduled using different strategies tailored for its particular characteristics.

Our algorithm consists of four main stages. The first one is an offline strategy that partitions the DAG into BoTs prior to its execution. The second one is an online budget distribution phase repeated throughout the execution of the workflow. It assigns a portion of the remaining budget to the tasks that have not been scheduled yet. The third

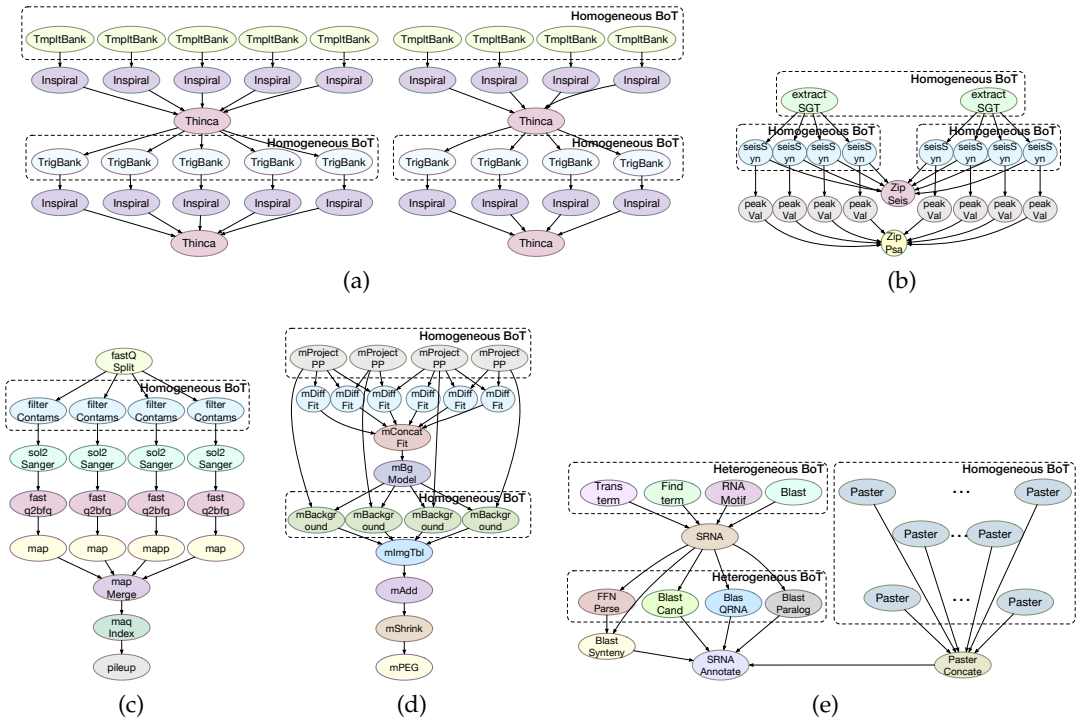


Figure 5.1: Examples of BoTs in five well-known scientific workflows. Tasks not belonging to a homogeneous or heterogeneous BoT are classified as a single-task BoT (a) LIGO (b) CyberShake (c) Epigenomics (d) Montage (e) SIPHT

stage is responsible for creating a resource provisioning plan for BoTs as their tasks become available for execution. Finally, ready tasks are scheduled and executed based on their corresponding provisioning plan. Each of these phases is explained in detail in the following sections.

5.4.1 DAG Preprocessing

This stage is responsible for identifying and partitioning the DAG into BoTs. Tasks are grouped together if they belong to the same data distribution structure and share the same single parent, or if they are entry tasks and have no parent tasks associated with them. If a task does not meet any of these requirements then it is placed on its own, single-task bag. BoTs with multiple tasks are further categorized into two different classes. The first category is groups of parallel homogeneous tasks, that is, all tasks in the bag are of the same type in terms of the computations they perform. The second one is comprised

of groups of heterogeneous tasks.

Hence, the preprocessing stage leads to the identification of the following sets:

- $BoT_{hom} = \{bot_1, bot_2, \ldots, bot_n\}$: Set of bags of homogeneous tasks,
- $BoT_{het} = \{bot_1, bot_2, \ldots, bot_m\}$: Set of bags of heterogeneous tasks,
- $BoT_{sin} = \{bot_1, bot_2, \ldots, bot_s\}$: Set of bags containing a single task.

5.4.2 Budget Distribution

The budget distribution phase assigns each individual task a portion of the budget and ultimately determines how fast a task can be processed. Although we propose a strategy here, it is worthwhile mentioning that this method can be easily interchanged without altering the methodology of the algorithm. During this stage, the cost of a task on a given VM type is estimated using the following equation,

$$C_t^{vmt} = \lceil P_t^{vmt} / \tau \rceil * c_{vmt}. \tag{5.6}$$

This definition relies on our assumption of fine-grained billing periods as a task's execution time is likely to be close to a multiple of the billing period, and if there is spare time, the additional cost incurred in paying for it is not significant. We do not include the VM provisioning delay here as the number of VMs that can be afforded to launch will be determined by the amount of spare, or leftover, budget after this distribution.

Relying on the assumption that the more expensive the VM type the faster it is capable of processing tasks, the first step consists in finding the most expensive (or fastest) VM type (vmt_{ex}) that if assigned to all tasks, their combined cost would be equal to or less than the budget. If no such type exist, then vmt_{ex} is defined to be the cheapest (or slowest) available VM type. If this is the case, although the estimated cost of running the workflow tasks on the cheapest VM type is higher than the budget, we do not conclude the budget is insufficient to run the workflow as at this stage we are overestimating the cost by assuming that VMs are not reused. This does however mean that there will be no spare budget to lease VMs and the algorithm will be forced to re-use existing ones.

Algorithm 5 Budget Distribution

```
1: procedure DISTRIBUTEBUDGET(β,T)
2:     levels = DAG levels
3:     Find fastest VM type vmt_ex such that IC = Σ_{i=1}^{|T|} C_{t_i}^{vmt_ex} ≤ β
4:     if No suitable IC ≤ β is found then
5:         vmt_ex = vmt_cheapest
6:     end if
7:     For every l ∈ levels do l.vmtype = vmt_ex
8:     For every t ∈ T do t.budget = C_t^{vmt_ex}
9:     if IC < β then
10:        spare = β − IC
11:        while spare > 0 and at least one level can be upgraded do
12:            for each level l ∈ levels with T_l ⊂ T tasks do
13:                vmt_up = next fastest vm than l.vmtype
14:                Previous level cost PLC = Σ_{i=1}^{|T_l|} C_{t_i}^{vmt_{l.vmtype}}
15:                New level cost NLC = Σ_{i=1}^{|T_l|} C_{t_i}^{vmt_up}
16:                if NLC − PLC ≤ spare then
17:                    l.vmtype = vmt_up
18:                    For every t ∈ T_l do t.budget = C_t^{vmt_up}
19:                    Update remaining spare budget
20:                end if
21:            end for
22:        end while
23:        if spare > 0 then
24:            for each level l ∈ levels do
25:                β_l = (|T_l|/|T|) ∗ spare
26:                l.provisioningBudget = β_l
27:            end for
28:        end if
29:    end if
30: end procedure
```

Additionally, before accepting a budget plan that is higher than the actual budget, the algorithm checks that the available money is at least enough to run all of the remaining tasks in a single VM of the cheapest type, denoted as the minimum cost plan. Further details of this heuristic are explained in section 5.4.4.

After determining vmt_{ex}, each task is assigned an initial budget corresponding to $C_t^{vmt_{ex}}$. BAGS then proceeds to distribute any spare or leftover budget by upgrading all tasks in a level using the following top-down strategy. Iteratively and starting at the top level of the DAG, all of the level's tasks are assigned additional budget corresponding to their execution on the next fastest VM type to vmt_{ex} if the total additional cost of running all the level's tasks on such VM type does not exceed the spare budget. This process is repeated until no more levels can be upgraded or the spare budget is exhausted.

Finally, any spare money left is distributed to each level for provisioning purposes. When a task is being scheduled, this provisioning budget will determine if a new VM can be launched, or if an existing one has to be reused. The distribution is proportional to the number of tasks in the level. Algorithm 5 depicts an overview of the budget assignment process.

5.4.3 Resource Provisioning

This section explains the strategies used to create the resource provisioning plans for each of the BoT categories. A high-level overview of the process is depicted in Algorithm 6.

Bags of Homogenous Tasks

The resource capacity for bags of homogenous tasks is estimated using mixed integer linear programming (MILP). The MILP model was designed to provide an estimate of the number and types of VMs that can be afforded with the given budget so that the tasks are processed with minimum makespan. The simplicity of the model was a main design goal as a solution for large bags needs to be provided in a reasonable amount of time.

We recognize that although tasks are homogenous, their computation time may differ as the size of their input and output data may vary. For this reason, and to keep the MILP model simple, we assume all tasks in the bag take as long to process as the most data intensive task. That is, the task that uses and produces the most amount of data out of all the ones in the bag.

The following notation is used to represent some basic parameters used in the model:

- n: number of tasks in the bag,
- β: available budget to spend on the bag. The budget for a multi-task BoT is defined as the sum of the budgets of the individual tasks contained in the bag. If there is any spare budget assigned to the DAG level to which the tasks belong to, then this is added to the BoT budget as well,

Algorithm 6 Resource Provisioning

```
1: procedure CREATERESOURCEPROVISIONINGPLAN(bot)
2:     if bot ∈ BoT_hom then
3:         solve MILP for homogeneous bot
4:         for each vmt that had at least one task assigned do
5:             numTasks = number of tasks assigned to a VM of type vmt
6:             numVMs = number of VMs of type vmt used
7:             RP_vmt = (numTasks, numVMs)
8:             RP_bot ∪ RP_vmt
9:         end for
10:    else if bot ∈ BoT_het then
11:        solve MILP for heterogeneous bot
12:        for each vm that had at least one task assigned do
13:            tasks = tasks assigned to vm
14:            RP_vm = (tasks, vm)
15:            RP_bot ∪ RP_vm
16:        end for
17:    else if bot ∈ BoT_sin then
18:        t = bot.task
19:        vmt_fast = find fastest VM that can finish the task within bot.budget
20:        if vmt_fast does not exist then
21:            vmt_fast = vmt_cheapest
22:        end if
23:        RP_bot = (vmt_fast)
24:    end if
25: return RP_bot
26: end procedure
```

- $IntTol$: refers to the MILP solver integrality tolerance. It specifies the amount by which an integer variable in the MILP can be different than an integer and still be considered feasible.

The following data sets representing the cloud resources are used as an input to the program:

- VMT: set of available VM types,
- VM_{vmt}: set of possible VM indexes for type vmt. Represents the number of VMs of the given type that can be potentially leased from the provider and ranges from 1 to n.

Each VM type is defined by the following characteristics:

- c_{vmt}: cost per billing period of vm type $vmt \in VMT$,
- $prov_{vmt}$: provisioning delay of a vm of type $vmt \in VMT$,

- $p_{t_{di}}^{vmt}$: processing time, as calculated in Equation 5.3 of the most data intensive task $t_{di} \in BoT$ in vm type $vmt \in VMT$.

To following variables are used to solve the problem:

- M: Makespan,
- $N_{vmt,k}$: integer variable representing the number of tasks assigned to the k^{th} VM ($k \in VM_{vmt}$) of type VMT,
- $L_{vmt,k}$: binary variable taking the value of 1 if and only if the k^{th} VM ($k \in VM_{vmt}$) of type VMT is to be leased, 0 otherwise,
- $P_{vmt,k}$ integer variable indicating the number of billing periods the k^{th} VM ($k \in VM_{vmt}$) of type VMT is used for,

The total number of time units the k^{th} VM ($k \in VM_{vmt}$) of type VMT is used for is defined as

$$U_{vmt,k}^{hom} = (p_{t_{di}}^{vmt} * N_{vmt,k}) + (prov_{vmt} * L_{vmt,k}), \tag{5.7}$$

and the total execution cost as

$$C_{botl} = \sum_{j \in VMT} \sum_{k \in VM_j} P_{vmt,k} * c_{vmt}. \tag{5.8}$$

The MILP is formulated as follows,

Minimize M Subject to:

$$\begin{aligned} M - U_{vmt,k} \geq 0 \\ \forall\ vmt \in VMT, \quad \forall\ k \in VM_{vmt}, \end{aligned} \tag{C1}$$

$$\sum_{j \in VMT} \sum_{k \in VMT_j} N_{vmt,k} = n, \tag{C2}$$

$$N_{vmt,k} \geq L_{vmt,k} \quad \forall\, vmt \in VMT, \quad \forall\, k \in VM_{vmt}, \tag{C3}$$

$$N_{vmt,k} \leq n * L_{vmt,k} \quad \forall\, vmt \in VMT, \quad \forall\, k \in VM_{vmt}, \tag{C4}$$

$$U_{vmt,k}/\tau \leq P_{vmt,k} \quad \forall\, vmt \in VMT, \quad \forall\, k \in VM_{vmt}, \tag{C5}$$

$$(U_{vmt,k}/\tau) + (1 - IntTol) \geq P_{vmt,k} \quad \forall\, vmt \in VMT, \quad \forall\, k \in VM_{vmt}, \tag{C6}$$

$$C_{bot} \leq \beta. \tag{C7}$$

Constraint C1 defines the BoT makespan as the longest time any of the leased VMs is used for. Constraint C2 ensures all the tasks are processed. Constraints C3 and C4 defines if a VM is leased based on the number of tasks assigned to it. Constraints C5 and C6 define the number of billing periods a VM is used by rounding up to the nearest integer the amount of time units the VM is used for. Finally, Constraint C7 ensures the total cost of does not exceed the budget.

After solving the problem, the variable $N_{vmt,k}$ is transformed into a resource provisioning plan of the form $RP^{hom}_{vmt} = (NumVms, NumTasks)$ for each VM type that has at least one task assigned to it. $NumVms$ indicates the number of VMs of type vmt to lease and $NumTasks$ the number of tasks that need to be processed by these VMs.

Bags of Heterogeneous Tasks

The strategy used to plan the resource provisioning of this type of bags is also based on MILP. The model is similar to that of homogenous tasks. An additional set $T_{bot} = \{t_1, t_2, \ldots, t_n\}$ representing the tasks in the bag is included. The processing time of each task $t \in bot$ on each VM type vmt is represented by the parameter p_t^{vmt}. The binary variable $A_{t,vmt,k}$ is used to solve the problem in addition to M, $L_{vmt,k}$, and $P_{vmt,k}$. $A_{t,vmt,k}$ takes de value of 1, if and only if, task t is allocated to the k_{th} VM of type vmt.

The total number of time units the k_{th} VM ($k \in VM_{vmt}$) of type VMT is used for is defined as

$$U_{vmt,k}^{het} = \sum_{tinT_{bot}} (p_t^{vmt} * A_{t,vmt,k}) + (prov_{vmt} * L_{vmt,k}), \tag{5.9}$$

and the total execution cost is defined by Equation 5.8.

The MILP is formulated in the same way as in section 5.4.3 with the following differences,

- Constraint C2 is reformulated to ensure that all the tasks are processed and that each task is assigned to a VM only once,

$$\sum_{j \in VMT} \sum_{k \in VMT_j} A_{t,vmt,k} = 1 \quad \forall\, t \in T_{bot}, \tag{C2}$$

- Constraints C3 and C4 are reformulated in terms of the variable $A_{t,vmt,k}$,

$$\sum_{t \in T_{bot}} A_{t,vmt,k} \geq L_{vmt,k} \quad \forall\, vmt \in VMT, \quad \forall\, k \in VM_{vmt}, \tag{C3}$$

$$\sum_{t \in T_{bot}} A_{t,vmt,k} \leq n * L_{vmt,k} \quad \forall\, vmt \in VMT, \quad \forall\, k \in VM_{vmt}. \tag{C4}$$

After solving the problem, the variable $A_{t,vmt,k}$ is transformed into a resource provisioning and scheduling plan of the form $RP^{het}_{vm} = (vm, T_{vm} \subset T_{bot})$ for each VM that had at least one task assigned to it. Notice that this provisioning plan determines the actual machines to use and the tasks that they are required to run, as opposed to just indicating the number and type of VMs to use. Due to the complexity of the MILP, heterogeneous BoTs are limited in size to a constant N^{het}_{bot}. This constant is provided as a parameter to the algorithm and ensures the proposed MILP is solved in a reasonable amount of time. Bags larger than this parameter are split so that they contain at most N^{het}_{bot} tasks.

Bags with a Single Task

This heuristic finds the fastest VM type that can be afforded with the budget assigned to the task. This is done by estimating the runtime of the task and its associated cost using Equation 5.6 on each available VM type. The one that can finish the task with minimum time and within the budget is selected and a resource provisioning plan of the form $RP^{sin} = (vmt_{fastest})$ assigned to the task.

5.4.4 Scheduling

The scheduling is done by processing tasks that are in the scheduling queue and ready for execution. Each time the queue is processed, a max-min strategy is used and tasks are sorted in ascending order based on their predicted runtime on the slowest VM type. In this way we ensure larger tasks from multi-task bags are scheduled first. A high-level view of the algorithm is shown in Algorithm 7.

For each ready task, the first step is to identify the bag *bot* to which it belongs to. Afterwards, the algorithm determines if the bag *bot* has an active resource provisioning plan associated to it. If such plan has not been created yet, then the budget distribution is updated based on the remaining tasks and budget. A provisioning plan is then created considering the type of the bag, its budget, and the spare budget assigned to the corresponding DAG level. The latter value will determine the number of VMs that can be launched to process the bag. Once this plan is created, all the other tasks belonging

Algorithm 7 Scheduling

```
1: procedure SCHEDULEQUEUE(Q)
2:     while Q is not empty do
3:         t = Q.poll()
4:         bot = getBoT(t)
5:         if no provisioning plan RP exists for bot then
6:             β_r = remaining budget
7:             T_r = remaining tasks
8:             distributeBudget(β_r, T_r)
9:             update bot budget
10:            update t.level spare budget
11:            create provisioning plan RP_bot
12:        end if
13:        if there is an idle VM vm_bot^idle ∈ VM_bot^idle then
14:            schedule(t, vm_bot^free)
15:        else if there is a suitable general purpose idle VM vm_gp^idle ∈ VM_gp^idle then
16:            schedule(t, vm_gp^idle)
17:        else
18:            if bot ∈ BoT_hom and bot.hasRemainingVMQuota() then
19:                vmType = RP_bot.nextVMTypeToLease()
20:                vm^new = provisionVM(vmType)
21:                schedule(t, vm^new)
22:            else if bot ∈ BoT_het then
23:                vm = RP_bot.getVmForTask(t)
24:                if vm is not leased then
25:                    vm = provisionVM(vm.vmType)
26:                end if
27:                schedule(t, vm)
28:            else if bot ∈ BoT_sin and t.level spare budget is enough to lease RP_bot.VMType then
29:                vm^new = provisionVM(RP_bot.VMType)
30:                schedule(t, vm^new)
31:            else
32:                place t back in queue
33:            end if
34:        end if
35:    end while
36: end procedure
```

to the bag (i.e. $\forall\ t_i \in bot$) will be scheduled based on it. In this way, the mathematical models only need to be solved once for each bag, when the first task of the bag is found in the scheduling queue.

Each active provisioning plan has a set of VMs, VM_{bot}, that were leased to serve the corresponding *bot*. This set is composed of busy VMs (VM_{bot}^{busy}) that are running tasks and idle VMs (VM_{bot}^{idle}) that can, and should, be reused by tasks in the bag. Once a VM is not required to process more tasks in the bag, it is removed from VM_{bot} and placed in a general purpose VM set. This set, VM_{gp}^{idle}, contains idle VMs that can be reused by

any task from any bag. VMs in this set that are approaching their next billing cycle are shutdown to avoid incurring in additional billing periods.

Once *bot* has an associated resource provisioning plan RP_{bot}, then the task $t \in bot$ being processed can be scheduled. For bags with a single task, the algorithm first tries to reuse an existing VM from VM_{gp}^{idle}. The purpose is to avoid the cost and time overhead of provisioning delays, to reduce cost by using idle time slots, and to reduce the number of data transfers to be storage by assigning tasks to VMs which contain all or some of their input data. An idle VM is chosen if it can finish the task at least as fast as it was expected by its provisioning plan and with a cost less than or equal to its budget. If multiple free VMs fulfill these conditions, then the one that can finish the task the fastest is selected. In this way, tasks are encouraged to run on the same resources as their parent tasks, as they are expected to have smaller runtimes in VMs where their input data is readily available. If no idle VM is found, a new one of the type specified by the plan is leased if the level's spare budget allows for it. If not, then the task is put back in the queue to be scheduled later on an existing VM that becomes available.

Tasks belonging to bags of homogenous tasks are processed in a similar way. The first step is to try to map the task to a free VM in VM_{bot}^{idle}. VMs in this set are sorted in ascending CPU capacity order, in this way, the most powerful VM is always reused first. This in conjunction with the max-min strategy used to sort tasks ensures that the largest tasks get assigned to the fastest VMs when possible. If there are no VMs in VM_{bot}^{idle}, then the algorithm tries to schedule the task on a free VM from VM_{gp}^{idle} that can finish the task for the same or a cheaper price than the most expensive VM type in the provisioning plan. If no suitable idle VM is found then the provisioning plan is followed in the following way. If the number of VMs leased for the provisioning plan is less than the specified one, then a new VM can be leased to run the task. The fastest VM type of those still available is chosen. If the VM quota has been reached and all the necessary VMs have been leased, then the task is put back in the queue so that it can be mapped to an existing VM assigned to the *bot* provisioning plan during the next scheduling cycle.

Tasks from heterogeneous bags are simply scheduled onto the VM specified by their provisioning plan. If the VM has already been leased then the task is added to the queue

of jobs waiting to be processed by the VM. If it has not been leased, then it is provisioned and the task assigned to it.

Finally, we define the minimum cost required to run a set of tasks T as the cost of running all the tasks sequentially on a single VM of the cheapest type,

$$C_T^{min} = \lceil (\sum_{i=1}^{|T|} P_{t_i}^{vmt_{cheapest}}) / \tau \rceil * c_{vmt_{cheapest}}. \tag{5.10}$$

Whenever $C_T^{min} > \beta$ or there is no feasible solution to a MILP problem, then the minimum cost plan is put in place. This plan consists on assigning every task to a single VM of the cheapest available type. This is done with the aim of reducing cost as much as possible until the algorithm recovers or to finish the execution of the workflow with a cost as close to the budget as possible.

5.5 Performance Evaluation and Results

The performance of BAGS was evaluated using the five well-known workflows from different scientific areas depicted in Section 1.1.2 (Montage, LIGO, SIPHT, Epigenomics, and CyberShake), each with approximately 1000 tasks.

An IaaS provider offering a single data center and four types of VMs was modeled using CloudSim [37]. The VM type configurations are based on those offered by Google Compute Engine and are shown in Table 5.1. A VM billing period of 60 seconds was used. For all VM types, the provisioning delay was set to 60 seconds. CPU performance variation was modeled after the findings by Schad et al. [99]. The performance of a VM was degraded by at most 24% based on a normal distribution with a 12% mean and a 10% standard deviation. Based on the same study, the bandwidth available for each data transfer within the data center was subject to a degradation of at most 19% based on a normal distribution with a mean of 9.5% and a standard deviation of 5%. The described CPU degradation configuration was used in all of the experiments except those in section 5.5.3 while the specified data transfer degradation was used throughout all of the experiment sets. A global shared storage with a maximum reading and writing speeds

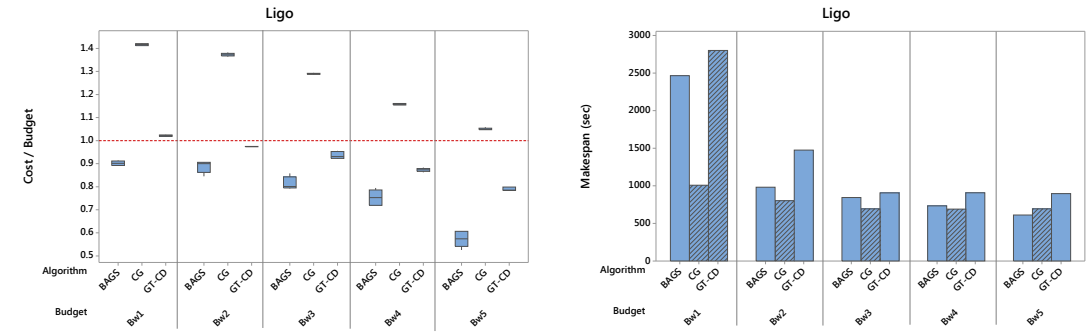
Table 5.1: VM types based on Google Compute Engine's offerings.

Name	Memory	Google Compute Engine Units	Price per Minute
n1-standard-1	3.75GB	2.75	$0.00105
n1-standard-2	7.5GB	5.50	$0.0021
n1-standard-4	15GB	11	$0.0042
n1-standard-8	30GB	22	$0.0084

was also modeled. The reading speed achievable by a given transfer is determined by the number of processes currently reading from the storage, the same rule applies for the writing speed. In this way, we simulate congestion when trying to access the storage system.

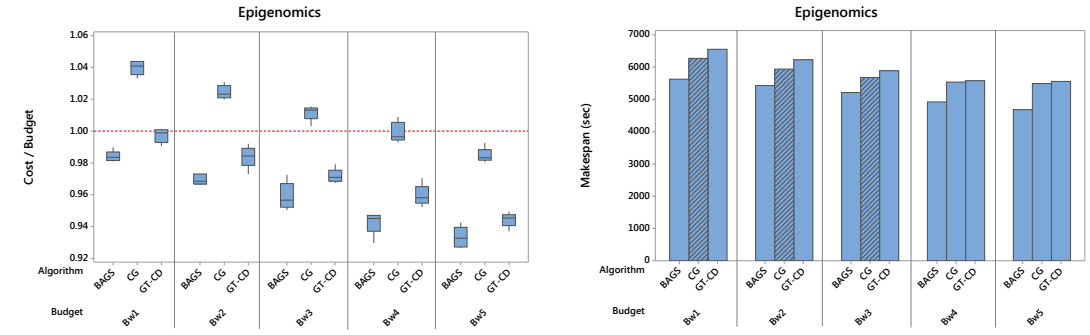
The experiments were conducted using five different budgets, β_{W1} being the strictest one and β_{W5} being the most relaxed one. For each workflow, β_{W1} is equal to the cost of running all the tasks in a single VM of the cheapest type. β_{W5} is the cost of running each workflow task on a different VM of the most expensive type available. An interval size of $\beta_{int} = \beta_{W5} - \beta_{W1}/4$ is then defined and used to estimate the remaining budgets: $\beta_{W2} = beta_{W1} + \beta_{int}$, $\beta_{W3} = beta_{W2} + \beta_{int}$, and $\beta_{W4} = beta_{W3} + \beta_{int}$.

Two algorithms were used when evaluating the performance of BAGS. The first one is called GreedyTime-CD [120] (GT-CD) and was developed for utility grids. It distributes the budget to tasks based on their average execution times. At runtime, VMs that can finish the tasks with minimum time within their budget are selected. We adapted GT-CD to dynamically lease VMs based on a task's assigned budget. This auto-scaling mechanism was designed so that leased VMs are reused when possible without impacting the original schedule produced by the algorithm. The second one is the Critical-Greedy [111] (CG) budget-constrained algorithm. It was developed for IaaS cloud environments and makes an initial estimate of cost boundaries for each task based on the available budget and VM types. Any additional budget is distributed to each task based on a time and cost difference ratio. CG ignores billing periods when calculating the cost of using a VM type and it does not specify how to allocate tasks to actual VMs. We adapted the algorithm to consider billing periods when estimating the task's cost boundaries and introduced the same VM auto-scaling mechanism implemented for GT-CD.



(a) (b)

Figure 5.2: Makespan and cost experiment results for the LIGO workflow.

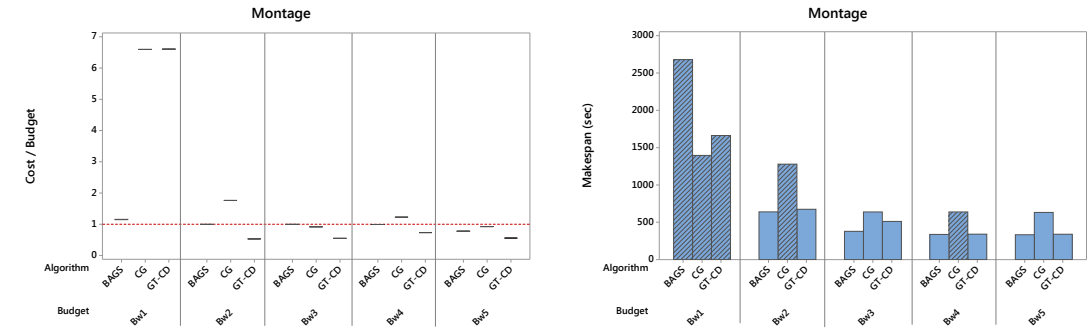


(a) (b)

Figure 5.3: Makespan and cost experiment results for the Epigenomics workflow.

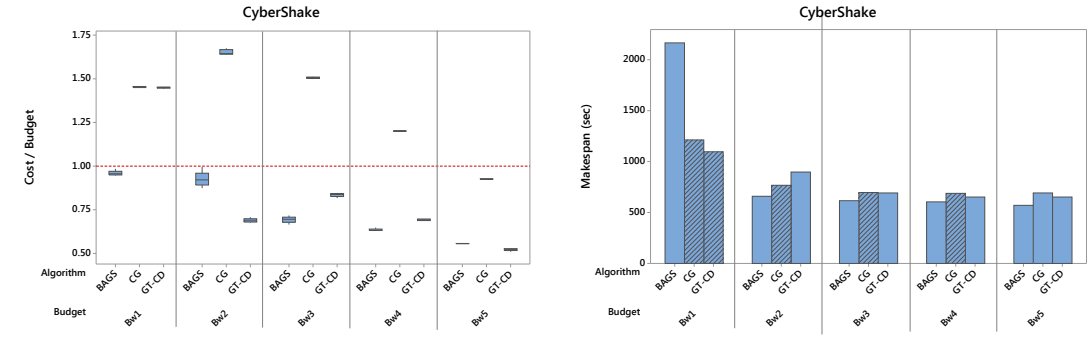
5.5.1 Algorithm Performance

The goal of these experiments is to evaluate the performance of the algorithms in terms of cost and makespan. The cost performance is determined by an algorithm's ability to meet the specified budget constraint, this is evaluated by using the workflow's cost to budget ratio. In this way, ratio values greater than one indicate a cost larger than the budget, values equal to one a cost equal to the budget and, values smaller than one a cost smaller than the budget. The experiments for each budget interval, workflow, and algorithm were run 20 times. The box plots displaying the cost to budget ratios



(a) (b)

Figure 5.4: Makespan and cost experiment results for the Montage workflow.

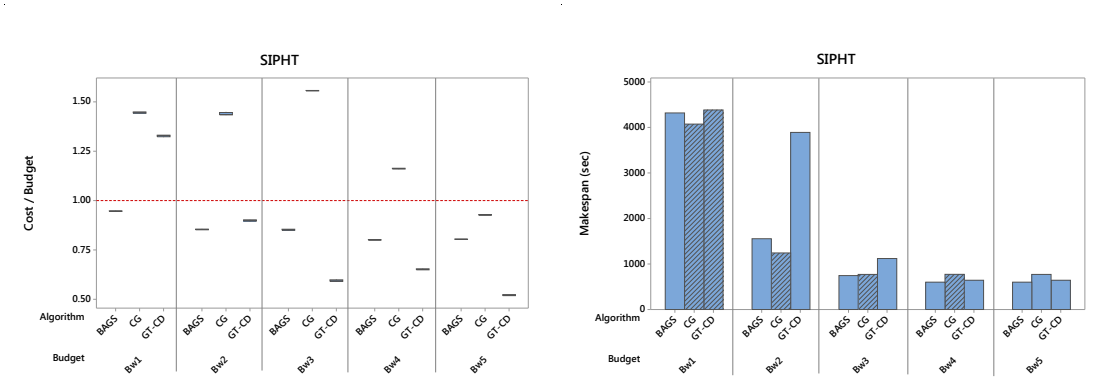


(a) (b)

Figure 5.5: Makespan and cost experiment results for the CyberShake workflow.

summarize these data while the bar charts depicting the workflow's makespan show the mean value obtained from the data. The dashed bars in the makespan bar charts indicate that the mean cost obtained by the algorithm exceeded the corresponding budget.

The results obtained for the LIGO workflow are shown in Figure 5.2. BAGS is the only algorithm capable of achieving a ratio smaller than one for all of the five budget intervals. The mean ratio obtained by GT-CD is below one from the second to the fifth budget intervals, while CG fails to meet the budget in all of the five cases. In every scenario in which BAGS and GT-CD meet the budget, BAGS achieves a lower makespan, demonstrating its ability to generate high-quality schedules.



(a) (b)

Figure 5.6: Makespan and cost experiment results for the SIPHT workflow.

Figure 5.3 depicts the results obtained for the Epigenomics application. Both BAGS and GT-CD are successful in meeting the five budget constraints, while CG meets the last three. BAGS always achieves the lowest makespan out of those algorithms that complete the execution within budget. These results demonstrate once again the efficiency of the makespan-minimizing heuristics used in BAGS.

The results for the Montage application are shown in Figure 5.4. The first budget constraint proves too tight for any of the algorithms to meet it. However, the ratio obtained by BAGS is considerably smaller than the ratio obtained by the other algorithms. For the second and third budget intervals, BAGS outperforms those algorithms capable of meeting the budget by obtaining lower makespans. The fourth budget interval sees GT-CD and BAGS obtain very similar average makespans, and in this case, GT-CD obtains a lower ratio when compared to BAGS. All of the algorithms are successful in meeting the final budget interval, with BAGS and GT-CD obtaining once again very similar makespans that are considerably smaller than the ones obtained by CG.

The CyberShake workflow results are shown in Figure 5.5. The first budget constraint is too strict for either GT-CD or CG meet it. BAGS demonstrates its ability to deal with unexpected delays by being the only algorithm capable of staying within this budget. For the rest of the budget intervals, BAGS outperforms in every case the other algorithms in terms of makespan. In the cases of B_{w3} and B_{w4} BAGS not only achieves the fastest time

but also the cheapest cost.

Figure 5.6 shows the results obtained for the SIPHT workflow. BAGS succeeds in meeting the budget in every case, GT-CD meets the four most relaxed constraints, and CG meets only the last budget interval. In all of the five scenarios, SIPHT outperforms the other algorithms by generating lower makespan schedules.

Overall, BAGS is the most successful algorithm in meeting the budget constraints by achieving its goal in all of the scenarios except one, the first budget interval of the Montage workflow. Even in this case, it performs better than the other algorithms by having a ratio value approximately six times smaller than that of GT-CD and CG. This demonstrates the importance of tailoring an algorithm to consider the underlying cloud characteristics in order to take advantages of the features offered by the platform and meet the QoS requirements. The experiments also demonstrate the efficiency of BAGS in generating higher-quality schedules by achieving a lower makespan values in every case except one (Montage workflow, β_{W4}). These results highlight the efficiency of the time optimization strategies used by BAGS. Another desirable characteristic of BAGS that can be observed from the results is its ability to consistently decrease the time it takes to run the workflow as the budget increases. The importance of this relies in the fact that many users are willing to trade-off execution time for lower costs while others are willing to pay higher costs for faster executions. The algorithm needs to behave within this logic in order for the budget value given by users to be meaningful.

5.5.2 Provisioning Delay Sensitivity

Fine-grained billing periods encourage frequent VM provisioning operations and therefore, it is important to evaluate the ability of BAGS to finish the workflow execution with a cost no greater than the given budget under different VM provisioning delays. The delays were varied from 0 to 9 billing periods (540 seconds). Figure 5.7 shows the ratios of cost to budget obtained for the each of the workflow applications across all five budgets. Identical outlier data points are displayed as a single symbol.

For the LIGO application, the mean and median ratio values remained under one for all of the provisioning delays. However, for the last two values, 420 and 540 seconds, the

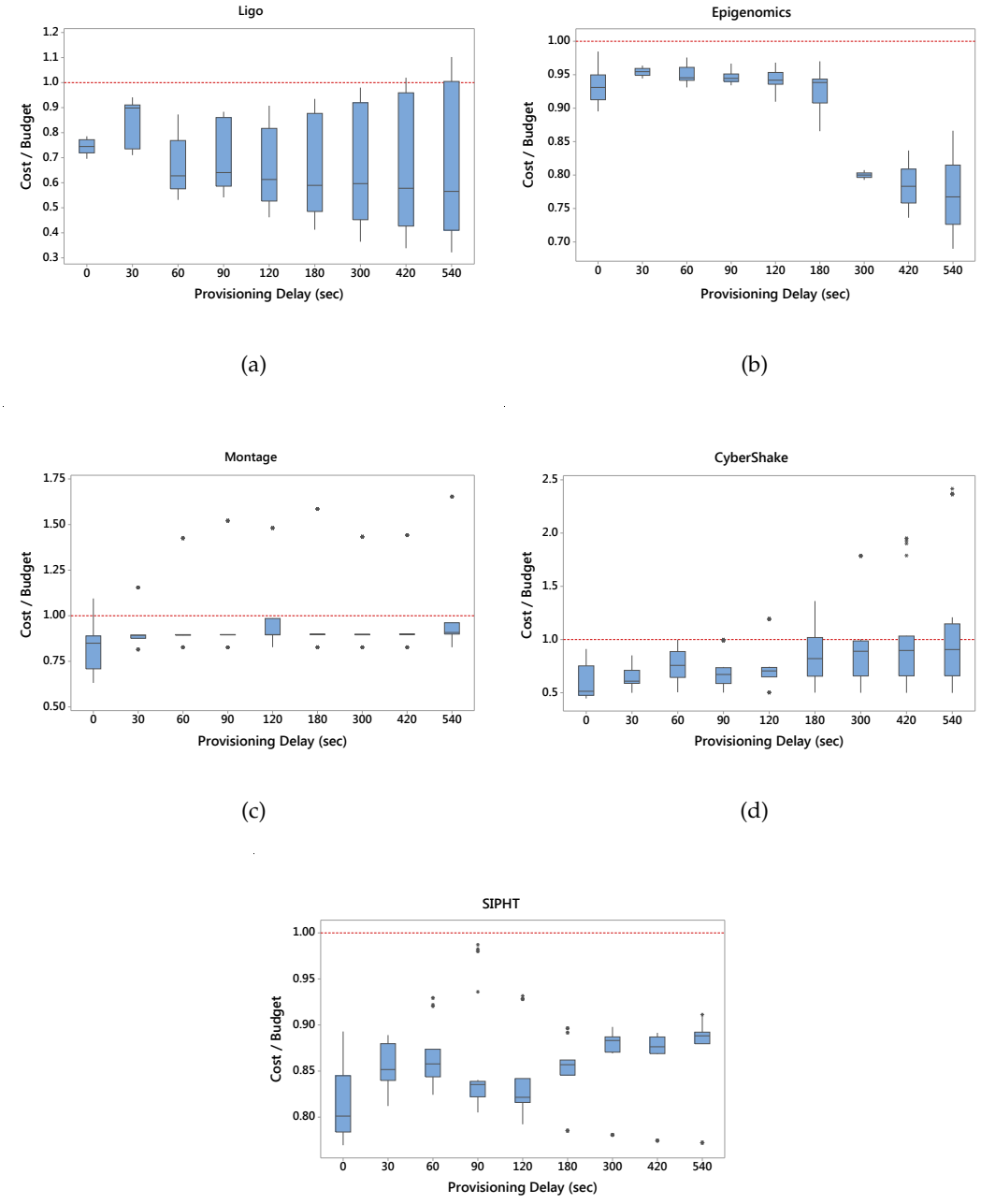


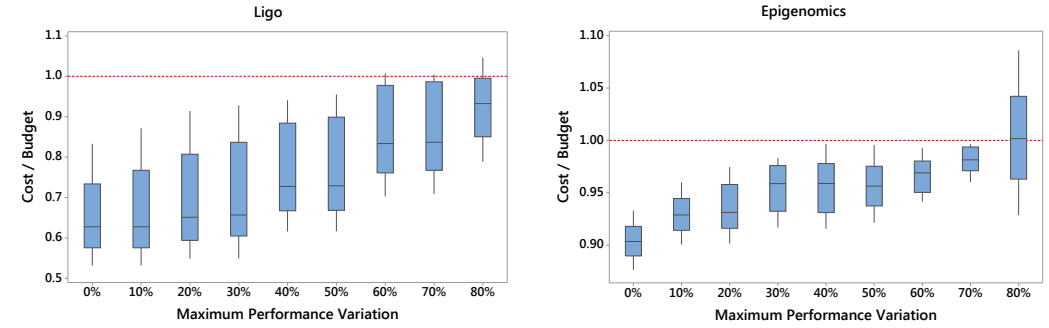
Figure 5.7: Cost to budget ratios obtained for each of the workflows with varying VM provisioning delays.

maximum ratio value obtained is slightly higher than one. This is due to the algorithm being unable to meet the first budget interval, as it becomes too strict for it to be achievable with such high provisioning delays. The results for the Montage workflow display maximum or outlier values greater than one in every case, this is inline with what was found when analyzing the performance of the algorithms, the first budget is too strict for BAGS to finish on time regardless of the provisioning delay. The mean and median values however remain well below one in every case. For the CyberShake application, outliers greater than one start to appear from a provisioning delay value of 120 seconds onwards. Once again, these ratios correspond to the strictest budget and they increase in value as the delay increases. In the Epigenomics and SIPHT cases, all of the ratio data points are below one, demonstrating the ability of BAGS to adapt to increasing provisioning delays as long as the budget allows for it.

5.5.3 Performance Degradation Sensitivity

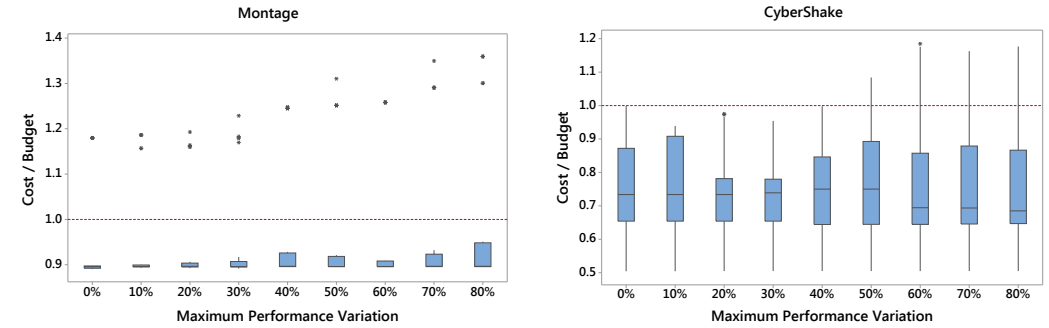
The sensitivity of the algorithm to the VM CPU performance variation was studied by analyzing the cost to budget ratio under different degradation values. The performance variation was modeled using a normal distribution with a variance of 1% and different average and maximum values. The average values were defined as half of the maximum CPU performance degradation which range from 0% to 80%.

The results obtained are depicted in Figure 5.8, identical outlier data points are displayed as a single symbol. The mean and median ratio values are under one for all of the degradation values for the LIGO application. With an 80% maximum degradation however, the maximum ratio obtained is just over one and corresponds to the strictest budget value. The results are similar for the Epigenomics application, but in this case, a greater sensitivity to the unexpected delays is seen in the case of 80% maximum degradation, with the median being slightly higher than one. The outliers displayed in the Montage box plot correspond once again to the first budget, which is too strict to be met regardless of the provisioning delay or performance variation. All the other ratios obtained remained under one for this application. The CyberShake workflow is more sensitive to degradation with the maximum ratio values exceeding one from 50% onwards. These



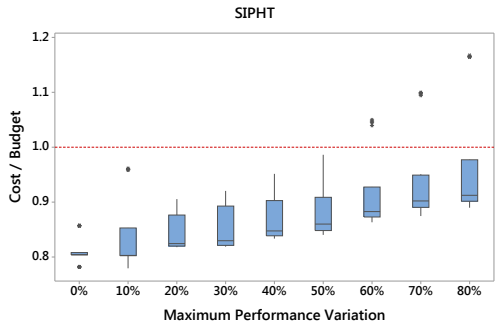
(a)

(b)



(c)

(d)



(e)

Figure 5.8: Cost to budget ratios obtained for each of the workflows with different CPU performance variation values.

belong to the strictest budget as the mean and median values are well below one in all of the cases. Finally, the results obtained for SIPHT demonstrate the algorithm is capable of finishing within budget in most of the cases, except for some outlier data points for the three greater performance variation values.

Another potential cause for exceeding the budget constraint is the fact that BAGS creates a static provisioning plan for BoTs with multiple tasks. Although this enables the algorithm to make better optimization decisions to minimize the makespan of workflows, it also affects its responsiveness to changes in the environment. These results demonstrate however, that despite this, BAGS is still successful in achieving it's budget goal in the vast majority of cases. As a future work, a rescheduling strategy for multi-task BoTs will be explored with the aim of further reducing the impact of unexpected delays.

5.5.4 Mathematical Models Solve Time

The time taken to solve the MILP models for homogeneous and heterogeneous bags was also studied. The number of tasks used as input to the homogeneous BoT model was varied from 10 to 1000 while the number of tasks for the heterogeneous BoT model was varied from 10 to 100. For each of these values, experiments using 10 different budget values ranging from stricter to more relaxed ones were performed. Figures 5.9 and 5.10 summarize the results obtained. Both models were formulated in AMPL and solved using the default configuration of CPLEX.

The results obtained for the homogeneous BoT case demonstrate the scalability of the proposed model with the maximum time taken to solve the problem being approximately 4.5 minutes for 800 tasks and the largest median value being 40 seconds for 1000 tasks. The performance of the heterogeneous BoT model however is greatly affected by the number of tasks being scheduled. For 100 tasks, the maximum solve time obtained is in the order of 14 minutes, this value is to high and unpractical for our scheduling scenario. Based on these results, the maximum number of tasks, N_{bot}^{het}, allowed in an heterogeneous bag was defined as 50, for which we obtained a maximum solve time of approximately 4.5 minutes.

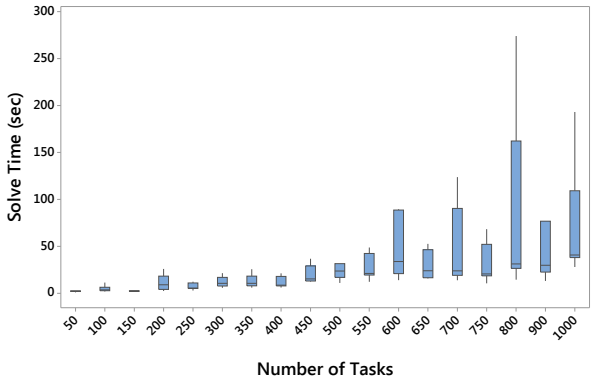


Figure 5.9: Solve time for the homogeneous BoT MILP model. The results display the time for solving the MILP with 4 different VM types and across 10 different budgets, ranging from stricter to more relaxed ones,

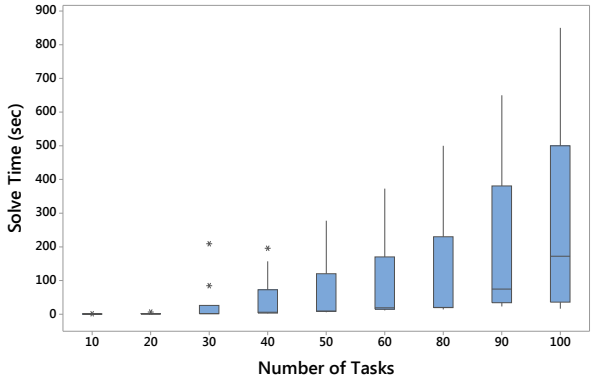


Figure 5.10: Solve time for the heterogeneous BoT MILP model. The results display the time for solving the MILP with 4 different VM types and across 10 different budgets, ranging from stricter to more relaxed ones.

5.6 Summary

BAGS, an adaptive resource provisioning and scheduling algorithm for scientific workflows in clouds capable of generating high quality schedules was presented in this chapter. It has as objective minimizing the overall workflow makespan while meeting a user-defined budget constraint. The algorithm is dynamic to respond to unexpected delays and environmental dynamics common in cloud computing. It also has a static component to schedule groups of tasks that allows it to find the optimal schedule for a set of workflow tasks improving the quality of the schedules it generates.

The simulation experiments show that the proposed algorithm has an overall better performance than other state-of-the-art algorithms. It is successful in meeting the strictest budgets under unpredictable situations involving CPU and network performance variation as well as VM provisioning delays.

This chapter presented the last of the algorithms proposed in this thesis. Next, the implementation of a cloud workflow management system used to deploy workflows in real-cloud environments is explained. We also implement the WRPS algorithm in this system and deploy a real-life Montage workflow to validate our simulation results.

Chapter 6

The Cloudbus Workflow Management System

This chapter presents the design of a workflow management system used to manage the execution of workflows in distributed environments. Its fundamental components are described focusing on the workflow engine, the core component responsible for making resource provisioning and scheduling decisions. The system was extended to support the dynamic and on-demand acquisition of cloud resources and to support the WRPS algorithm explained in Chapter 4, the implementation details are presented here. Finally, a case study using a real world Montage workflow is described to demonstrate not only the newly added functionality but also that the experiment results obtained for WRPS in a real cloud environment are consistent with those obtained using a simulator.

6.1 Introduction

The execution of workflows in clouds is done via a cloud Workflow Management System (WMS). It enables the creation, monitoring and execution of scientific workflows and has the capability of transparently managing tasks and data by hiding the orchestration and integration details among the distributed resources [116]. A reference architecture is shown in Figure 6.1. The depicted components are common to most cloud WMS implementations, however, not all of them have to be implemented in order to have a fully functional system.

This chapter is derived from: **Maria A. Rodriguez** and Rajkumar Buyya. "Scientific Workflow Management System for Clouds" *Software Architecture for Big Data and the Cloud*, Elsevier - Morgan Kaufmann, 2017 (in press).

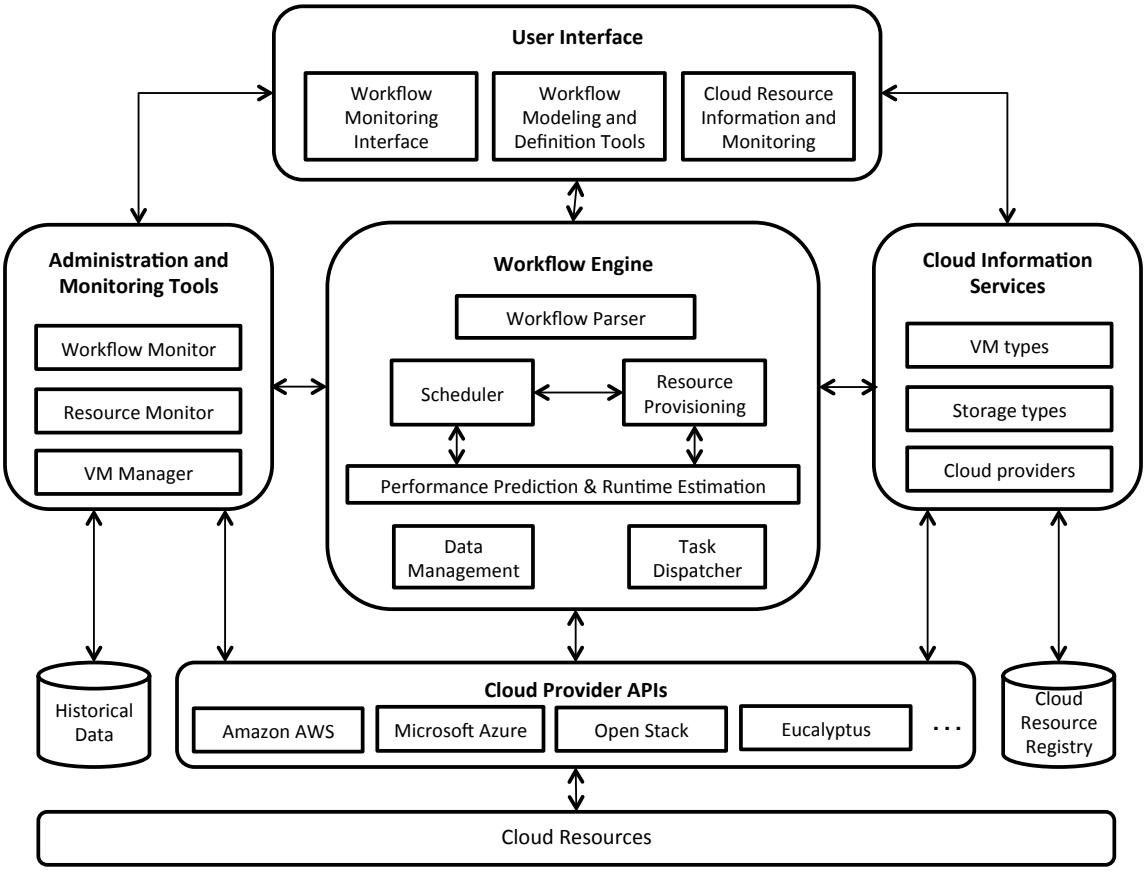


Figure 6.1: Reference architecture of a Workflow Management System.

User Interface. The user interface allows for users to create, edit, submit, and monitor their applications.

Workflow Engine. The workflow engine is the core of the system and is responsible for managing the actual execution of the workflow. The parser module within the engine interprets a workflow depicted in a high level language such as XML and creates the corresponding internal workflow representation such as task and data objects. The scheduler and resource provisioning modules work together in planning the execution of the workflow. The resource provisioning module is responsible of selecting and provisioning the cloud resources and the scheduling component applies specific policies that map tasks to available resources, both processes are based on the QoS requirements and scheduling objectives. The performance prediction and runtime estimation module use

historical data, data provenance, or time series prediction models, among other methods, to estimate the performance of cloud resources and the amount of time tasks will take to execute on different VMs. This data is used by the resource provisioning and scheduling modules to make accurate and efficient decisions regarding the allocation of tasks. The data management component of the workflow engine manages the movement, placement, and storage of data as required for the workflow execution. Finally, the task dispatcher has the responsibility of interacting with the cloud APIs to dispatch tasks ready for execution onto the available VMs.

Administration and Monitoring tools. The administration and monitoring tools of the WMS architecture include modules that enable the dynamic and continuous monitoring of workflow tasks and resource performance as well as the management of leased resources, such as VMs. The data collected by these tools can be used by fault tolerance mechanisms or can be stored in a historical database and used by performance prediction methods, for example.

Cloud Information Services. Another component of the architecture is the cloud information services. This component provides the workflow engine with information about different cloud providers, the resources they offer including their characteristics and prices, location, and any other information required by the engine to make the resource selection and mapping decisions.

Cloud Provider APIs These APIs enable the integration of applications with cloud services. For the scheduling problem described in this paper, they enable the on-demand provisioning and deprovisioning of VMs, the monitoring of resource usage within a specific VM, access to storage services to save and retrieve data, transferring data in or out of their facilities, and configuring security and network settings, among others. The majority of IaaS APIs are exposed as REST (Representational State Transfer) and SOAP (Simple Object Access Protocol) services, but protocols such as XML-RPC and JavaScript are also used. For instance CloudSigma, Rackspace, Windows Azure, and Amazon EC2 all offer REST-based APIs. As opposed to providing services for a specific platform, other

solutions such as Apache JClouds [3] aim to create a cross-platform cloud environment by providing and API to access services from different cloud providers in a transparent manner. Cross-platform interfaces have the advantage of allowing applications to access services from multiple providers without having to rewrite any code, but may have less functionality or other limitations when compared to vendor-specific solutions.

In this chapter we present the architecture of the Cloudbus Workflow Management System [88] (Cloudbus WMS), a WMS developed in the CLOUDS Laboratory at the University of Melbourne. The discussion focuses on the components that are most relevant to the scheduling problem addressed in this thesis. More importantly, we present the details of an extension made to the software to support dynamic, on-demand access to cloud resources. The WRPS algorithm presented in Chapter 4 was also implemented in the system and a case study using the Montage application is presented.

6.2 Cloudbus Workflow Management System

The Cloudbus WMS is a platform that allows scientist to express their applications as workflows and execute them on distributed resources. It enables the creation, monitoring and execution of large scale scientific workflows on distributed environments by transparently managing the computational processes and data. Its architecture consists of a subset of the components depicted in Figure 6.1 and is presented in Figure 6.2.

The *Workflow Portal* is the entry point to the system. It provides a web-based user interface for scientists to create, edit, submit and monitor their applications. It provides access to a *Workflow Deployment* page that allows users to upload any necessary data and configuration input files needed to run a workflow. A *Workflow Editor* is also embedded in this component and it provides a GUI that enables users to create or modify a workflow using drag and drop facilities. The workflow is modeled as a DAG with nodes and links that represent tasks and dependencies between tasks. The editor converts the graphical model designed by the users into an XML based workflow language called xWFL which is the format understood by the underlying workflow engine.

The *Workflow Monitor Interface* is also accessed through the portal and it provides a

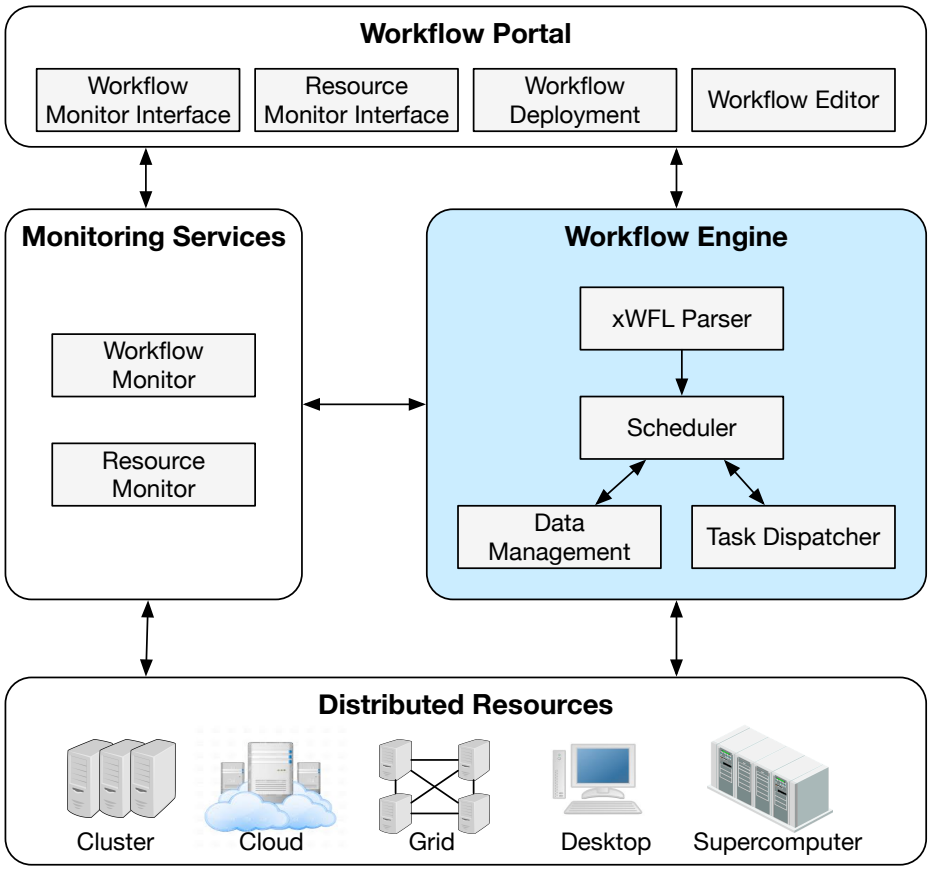


Figure 6.2: Key architectural components of the CloudbusWMS.

GUI to the *Workflow Monitor* module which is part of the Monitoring Services component. It allows users to observe the execution progress of multiple workflows and to view the final output of an application. Users can monitor the status of every task in a specific workflow, for instance, tasks can be on a ready, executing, stage in, or completed status. Additionally, users have access to information such as the host in which a task is running, the number of jobs being executed, and the failure history of each task. The Workflow Monitor relies on the information produced by the Workflow Engine, the interaction between these two components takes place via an event mechanism using tuple spaces. In broad terms, whenever the state of a task changes, the monitor is notified and as a response to the event, it retrieves the new state and any relevant task metadata from a central database. Finally, the portal offers users access to a *Resource Monitor Interface* which displays the information of all the current available computing resources. The *Re-*

source Monitor module in the Monitoring Services component is responsible for collection this information.

The *Workflow Engine* is the core of the Cloudbus workflow management system; its main responsibilities include scheduling, dispatching, monitoring, and managing the execution of tasks on remote resources. As shown in figure 6.2, the workflow engine has four main subsystems: workflow language parser, scheduler, task dispatcher, and data manager.

The workflow portal or any other client application submits a workflow for execution to the engine. The submitted workflow must be specified in the XML-based language, xWFL. This language enables users to define all the characteristics of a workflow such as tasks and their dependencies among others. Aside from the xWFL file, the engine also requires a service and a credential XML-based description files. The service file describes the resources available for processing tasks while the credentials one defines the security credentials needed to access these resources. The existence of these two files demonstrates the type of distributed platforms en engine was originally designed to work with, platforms where the resources are readily available and their type and number remains static throughout the execution of the workflow. Once the system is upgraded to support clouds, the use of these files will be obsolete as resources will be created and destroyed dynamically.

The xWFL file is then processed and interpreted by a subsystem called the workflow language parser. This subsystem creates objects representing tasks, parameters, data constraints and conditions based on the information contained on the XML file. From this point, these objects will constitute the base of the workflow engine as they are the ones containing all the information regarding the workflow that needs to be executed. Once this information is available, the workflow is scheduled and its tasks are mapped onto resources based on a specific scheduling policy. Next, the engine uses the Cloudbus Broker as a task dispatcher.

The Cloudbus Broker [107] provides a set of services that enable the interaction of the workflow engine with remote resources. It mediates access to the distributed resources by discovering them, deploying and monitoring tasks on specific resources, accessing the

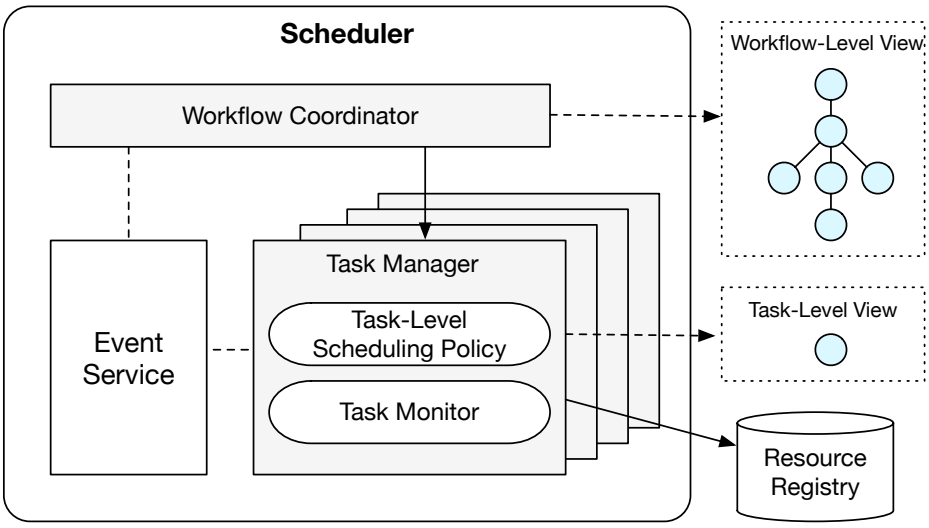


Figure 6.3: Key architectural components of the CloudbusWMS Scheduler.

required data during task execution and consolidating results. An additional component that aids in the execution of the workflow is the data movement service which enables the transfer of data between the engine and remote resources based on protocols such as FTP and GridFTP.

The workflow engine has a decentralized scheduling system that supports just-in-time planning and allows resource allocation to be determined at runtime. Each task has its own scheduler called Task Manager (TM). The TM may implement any scheduling heuristic and is responsible for managing the task processing, resource selection and negotiation, task dispatching and failure handling. At the same time, a Workflow Coordinator (WCO) is responsible for managing the lifetime of every TM as well as the overall workflow execution.

Figure 6.3 shows the interaction between the different components involved in the scheduling process. The WCO creates and starts a TM based on the task's dependencies and any other specific scheduling heuristic being used. Each TM has a task monitor that continuously checks the status of the remote task and a pool of available resources to which the task can be assigned. The communication between the WCO and the TMs takes place via events registered in a central event service.

Each TM is independent and may have its own scheduling policy, this means that

several task managers may run in parallel. Additionally, the behavior of a TM can be influenced by the status of other task managers. For instance, a task manager may need to put its task execution on hold until its parent task finishes running in order for the required input data to be available. For this reason, TMs need to interact with each other just as the WCO needs to interact with every TM; once again this is achieved through events using a tuple space environment.

6.3 Cloud-based Extensions to the Workflow Engine

Several extensions and changes were made to the workflow engine component of the Cloudbus WMS in order to support the execution of workflows in IaaS clouds. These extensions allow for scheduling algorithms and resource provisioning strategies to leverage the elastic and on-demand nature of cloud resources, in particular of VMs. The overall architecture of the system and the interaction between the main components remains the same, the extended architecture is shown in Figure 6.4, where the shaded components are the newly included ones. Each of these components is explained next and a class diagram depicting their implementation is presented in Figure 6.5.

VM Lifecycle Manager. A module providing an interface to access VM lifecycle management services offered by IaaS providers. These include leasing, shutting down, restarting, and terminating VMs. Access to a provider's VM management API is done using Apache JClouds[1], a java-based multi-cloud toolkit. It is an open source library that provides portable abstractions for cloud-specific features. It currently supports 30 providers and cloud software stacks such as, OpenStack, Amazon, Google, Rackspace, and Azure. The class diagram in Figure 6.5 shows the methods and IaaS providers currently supported by this module.

The realization of this module also included eliminating the need of having a set of compute services defined in an XML file previous to the execution to the workflow.

[1] Apache JClouds http:// jclouds.apache.org

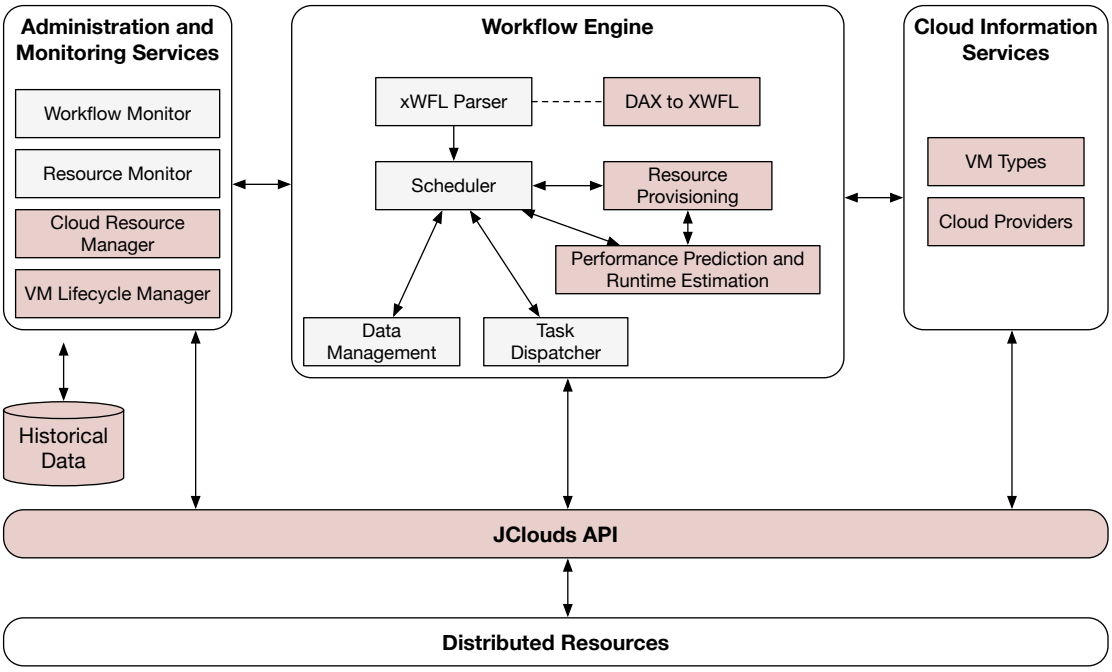


Figure 6.4: Key architectural components of the extended Cloudbus WMS.

Cloud Resource Manager. An entity responsible of managing the cloud resources used by the engine. It maintains information on the VMs leased from an IaaS provider. Its responsibilities include keeping track of leased, busy, and idle VMs, as well as recording data regarding the lease of VMs such as their lease start and end times.

The following are examples of data that can be accessed through the Cloud Resource Manager:

- Leased VMs: a list of all the VMs that have been leased throughout the lifecycle of the workflow execution.
- Terminated VMs: a list of all the VMs that have been terminated throughout the lifecycle of the workflow execution.
- Active VMs: a list of VMs that are currently leased and active.
- Busy VMs: a list of all VMs that are active and busy with the execution of one or more tasks.
- Idle VMs: a list of all VMs that are active and idle.

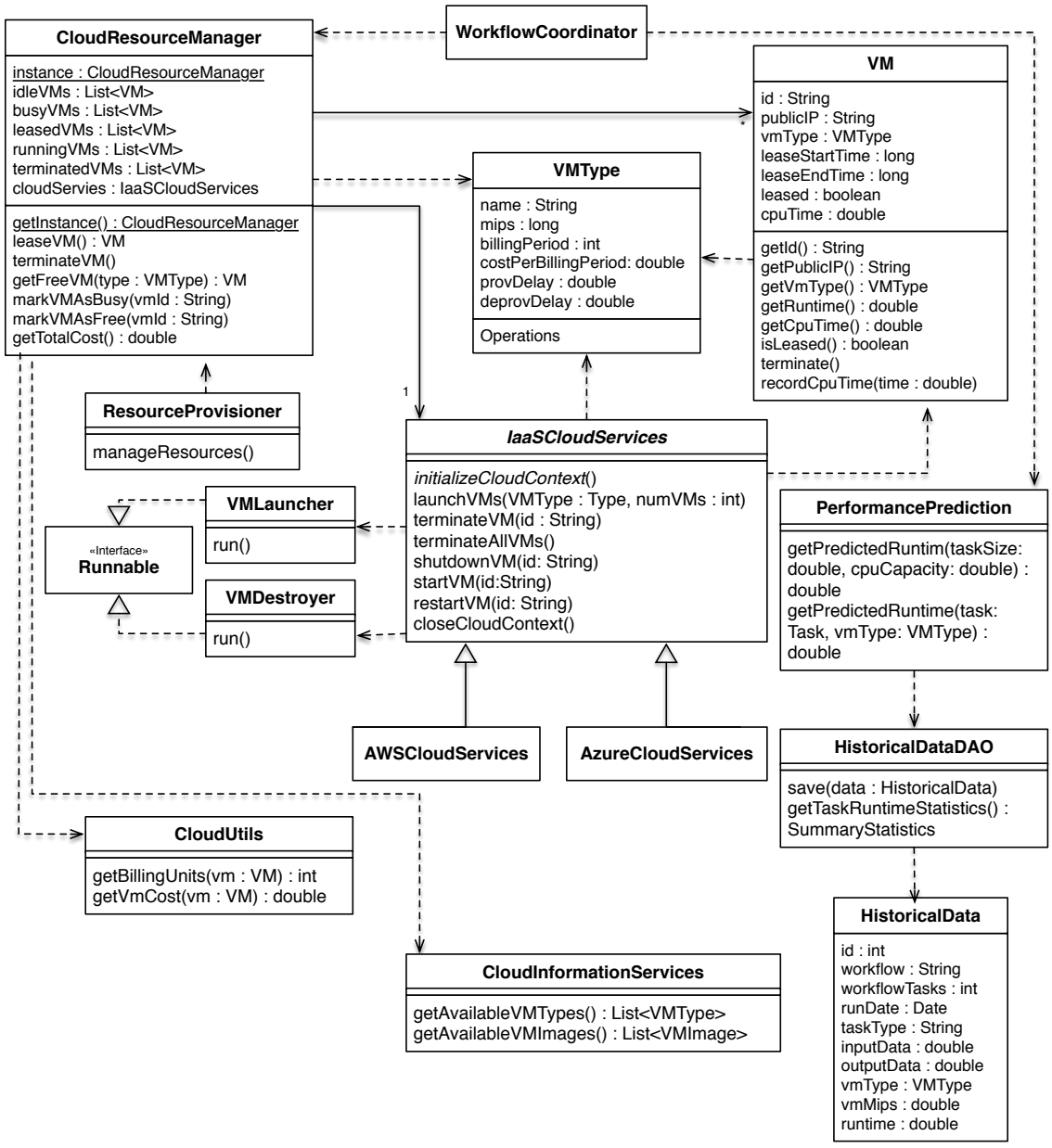


Figure 6.5: Class diagram architectural components of the CloudbusWMS Scheduler.

Resource Provisioning. An entity responsible of making resource provisioning decisions based on the scheduling objectives and QoS requirements. A basic provisioning strategy was implemented. It monitors the leased VMs every $PROV\_POLLING\_TIME$. The value for this polling interval is a configurable parameter that can be defined via a properties file. The provisioner then makes the decision to shut VMs down whenever

Algorithm 8 Resource Provisioning

```
1: procedure MANAGERESOURCES
2:     VM^idle = all leased VMs that are currently idle
3:     for each vm_idle in VM^idle do
4:         t_r = time remaining until next billing period
5:         t_d = Deprovisioningdelayestimate
6:         if (t_r − t_d ≥ 0) AND (t_r − t_d ≤ PROV_POLLING_TIME) then
7:             terminate vm_idle
8:         end if
9:     end for
10: end procedure
```

they are idle and approaching their next billing cycle. It does so by considering the time it takes for VMs to be deprovisioned, the time remaining until the VM reaches the next billing cycle, and the time when the next provisioning cycle will occur. If the deprovisioning delay is larger then the time remaining until the next billing cycle then there is no benefit on shutting down the VM as incurring in a new billing cycle is inevitable. Otherwise, the algorithm decides whether the VM can be left idle and be shutdown on later provisioning cycles without incurring in an additional billing period or if the VM should be deprovisioned in the current cycle to avoid incurring in additional costs. An overview of this strategy is depicted in Algorithm 8. The design provides the flexibility to plug-in different resource provisioning strategies without the need of modifying any other module. For instance, a provisioning strategy that not only decides when to shutdown VMs but also when to lease them based on a utilization metric could also be easily implemented.

Performance Prediction and Runtime Estimation Two different performance prediction strategies where implemented into a newly created Performance Prediction and Runtime Estimation Module. The first one is a straightforward strategy that allows for the runtime of tasks to be estimated using a measure of the size of a task and the CPU performance of the VM. For this purpose, the xWFL language as well as the existing parser were extended so that the definition of a task includes an optional element indicating its size. In practice, this size can be either the number of instructions (MI), the number of floating point operations (FLOP), or the time complexity of the tasks among others. Additionally, the definition of *compute service* within the engine was extended to include an

Table 6.1: Contents of the database table recording historical runtime data of tasks.

Property	Description
Workflow	Name of the workflow application
Number of Tasks	Total number of tasks in the workflow
Run Date	Date the workflow was deployed
Algorithm	Name of the scheduling algorithm managing the workflow execution
Task Type	Name or type of the workflow task for which the runtime is being recorded
Transferred Input Data	Amount of input data transferred to the task's VM
Transferred Output Data	Amount of output data transferred out of the task's VM
VM Type	Name of the VM type used to run the task
VM CPU Capacity	CPU capacity of the VM type
VM Memory	Memory available for the VM type
VM Bandwidth	Bandwidth of the VM type
Task Runtime	Time it took to complete the task's execution (including input transfer, computations, and output transfer)

optional property indicating a measure of the resource's CPU capacity. For this purpose the schema and parsers of the XML-based service file were modified to include the new property as was the *ComputeService* class.

The second strategy is based on the analysis of historical task runtime data. For this purpose, task runtimes are recorded on a historical database which can be later used to estimate the runtime of tasks on particular VM types using statistical tools. The data recorded for each task executed by the engine are depicted in Table 6.1. The current strategy calculates the 95% confidence interval of a task runtime given the task name or type, the workflow it belongs to, the number of tasks in the workflow, the amount of input and output data generated by the task, and the name of the VM type for which the prediction is being made for.

In the future, different prediction algorithms can be seamlessly implemented into this module and used by scheduling algorithms to guide their decisions.

Cloud Information Services. Through the cloud providers APIs, this module enables the workflow engine to query information regarding the types of services offered by a given provider. Specifically, the implementation leveraged the JClouds API to query the

types of VMs available from a given provider as well as the VM images available for use for a given user.

DAX to XWFL Tool The DAX [5] format is a description of an abstract workflow in XML that is used as the primary input into the Pegasus Workflow Management System [45], a tool developed at the Information Sciences Institute (ISI), University of Southern California. The extensive research done by this organization in workflows as well as their collaboration with the scientific community makes of the DAX format a popular and commonly one used. For instance, the Pegasus Project [15] has developed a tool in conjunction with the NASA/IPAC project that generates the specification of different Montage workflows in a DAX format. Hence, to take advantage of the existence of these tools as well as workflows described in the DAX format, a DAX to xWFL (the workflow description language supported by the CloudbusWMS) tool was developed as part of this thesis. In this way, the CloudbusWMS now has the ability to interpret workflows expressed in the DAX format.

Scheduler Extension. The workflow coordinator has been extended to have the ability to make scheduling decisions in terms of task to resource mappings. The previous version of the workflow engine limited its responsibilities to enforcing the dependency requirements of the workflow. That is, it was responsible for monitoring the status of tasks and releasing those ready for execution by launching their task manager, entity which was then responsible for deciding the resource where the task would be executed. The extended version allows for the workflow coordinator to make all of the scheduling and resource provisioning decisions if required based on its global view of the workflow. Additionally, the WRPS algorithm explained in Chapter 4 was implemented and integrated into the workflow coordinator.

6.4 Case Study: Montage

This section details the deployment of the Montage application on the CloudbusWMS. The workflow was scheduled using the WRPS algorithm and Microsoft Azure resources

that were dynamically provisioned using the cloud-enabled version of the CloudbusWMS.

6.4.1 Montage

The Montage application is designed to compute mosaics of the sky based on a set of input images. These input images are taken from image archives such as the Two Micron All Sky Survey (2MASS)[2], the Sloan Digital Sky Survey (SDSS)[3], and the Digitized Sky Surveys at the Space Telescope Science Institute[4]. They are first reprojected to the coordinate space of the output mosaic, the background of these reprojected images is then rectified, and finally they are merged together to create the final output mosaic [43].

Figure 1.4 depicts the structure of the Montage workflow as well as the different computational tasks it performs. The size of the workflow depends on the number of input images used and its structure changes to reflect an increase in the number of inputs, which results in an increase in the number of tasks. For this particular workflow, same-level tasks are of the same type, that is, they perform the same computations but on different sets of data.

The mProjectPP tasks are at the top level of the workflow and hence are the first ones to be executed. They process Flexible Image Transport System (FITS) input images by re-projecting them. There is one mProjectPP task for every FITS input image. In the next level are the mDiffFit tasks. They are responsible for computing the difference between each pair of overlapping images and as a result, their number is determined by the number of overlapping input images. Next is the mConcatFit task, it takes all of the different images as input and fits them using a least squares algorithm. This is a compute-intensive task as a result of its data aggregation nature. The next task is mBgModel which determines a background correction to be made to all the images. This correction is the applied to each individual image by the mBackground tasks in the next level of the workflow. Then, the mImgTbl task aggregates metadata from all the images and is followed by the mAdd job. This task is the most computationally intensive and is responsible for the actual aggregation of the images and the creation of the final mosaic. Finally, the size of

[2]Two Micron All Sky Survey: http://www.ipac.caltech.edu/2mass
[3]Sloan Digital Sky Survey: http://www.sdss.org
[4]Digitized Sky Surveys: http://www.stsci.edu/resources/

Table 6.2: Tasks in a 0.5 degree Montage workflow.

Task	Level	Count	Mean Runtime (sec)	Mean Input (MB)	Mean Output (MB)
mProjectPP	1	32	35.94	1.66	8.30
mDiffFit	2	73	31.72	16.6	1.02
mConcatFit	3	1	82.99	0.02	0.01
mBgModel	4	1	43.57	0.02	0.001
mBackground	5	32	30.43	8.31	8.30
mImgTbl	6	1	93.92	129.28	0.009
mAdd	7	1	241.50	265.79	51.73
mShrink	8	1	46.43	25.86	6.47
mJPEG	9	1	88.54	6.47	0.20

the final mosaic is reduced by the mShrink task and the output converted to JPEG format by the last workflow task, mJPEG [64].

For this case study, a Montage workflow constructing a 0.5 degree mosaic of the sky was used. This particular instance of the workflow consists of 143 tasks, their type, number, and level are depicted in Table 6.2.

The following are the specific characteristics of the Montage workflow used in this case study:

- Survey: 2mass
- Band: j
- Center: M17
- Width: 0.5
- Height: 0.5

6.4.2 Infrastructure Configuration

There are three types of components involved in the execution of a workflow using the CloudbusWMS. Each of these is deployed on its own compute resource or node. The first component is the actual workflow engine, or *master node*, which is responsible for

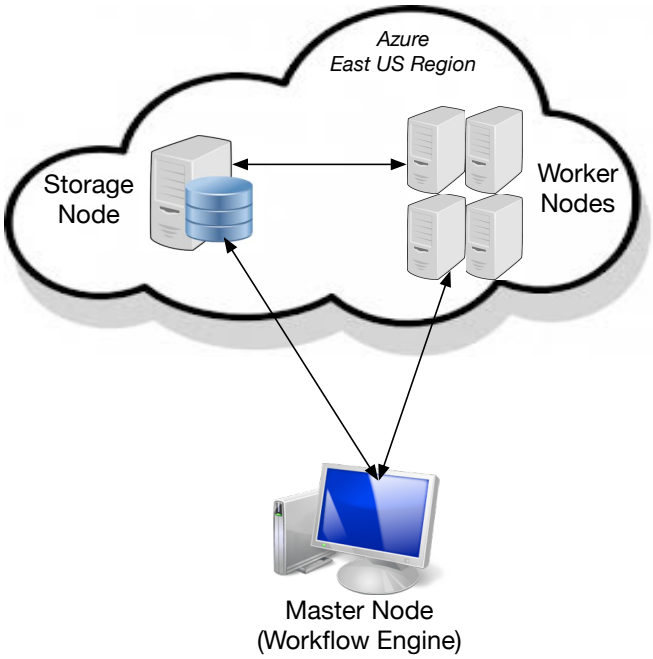


Figure 6.6: CloudbusWMS component deployment.

orchestrating the execution of tasks on *worker nodes*. The lifecycle of these worker nodes is managed by the engine and they contain the actual routines invoked by the workflow tasks. Finally, the *storage node* acts as a central file repository where worker nodes retrieve their input data from and store their output data to. Figure 6.6 depicts this deployment.

For the experiments performed in this chapter, the VM configuration and location used for each of these components is as follows:

- Master node: Ubuntu 14.4 LTS virtual machine running locally on a MacBook Pro with a 2.9 GHz Intel Core i7 processor and 8 GB RAM. The virtual machine was launched using Virtual Box and had a memory of 2.2GB and 125.6 GB disk.

- Storage node: Basic A2 Microsoft Azure virtual machine (2 cores, 3.5GB RAM) with Ubuntu 14.4 LTS installed deployed on the US East region.

- Worker nodes: Dynamically provisioned on Microsoft Azure's US East region using a custom VM image with Montage installed (see Section 6.4.3). The types of VMs where worker nodes could be deployed are depicted in Table 6.3. The A-series are general purpose compute instances while the D-series VMs feature solid state

Table 6.3: Types of VMs used to deploy a 0.5 degree Montage workflow.

VM Name	Cores	RAM	Disk Size	Price per Minute
A0 (extrasmall)	1	0.75 GB	20 GB	$0.000425
A1 (small)	2	1.75 GB	70 GB	$0.001275
A2 (medium)	2	3.5 GB	135 GB	$0.002548
D1	1	3.5 GB	50 GB	$0.001635
D2	2	7 GB	100 GB	$0.003270
D11	2	14 GB	100 GB	$0.004140

drives (SSDs) and have 60% faster processors than the A-series.

6.4.3 Montage Set Up

This section describes how the Montage routines were setup in the worker nodes VM image. It also explains how the input image files were obtained and how the workflow description XML file was generated.

The Pegasus Project has developed various tools that aid in the deployment of Montage workflows in distributed environments. The installation of Montage on the worker VM image as well as the generation of an XML file describing the workflow were done using these tools.

The first step was to download and install the Montage application, which includes the routines (mProjectPP, mDiffFit, mConcatFit, mBgModel, mBackground, mImgTbl, mAdd, mShrink, and mJPEG) corresponding to each workflow task. For this case study, version 3.3 was installed on a VM running Ubuntu 14.4 LST. In addition to the task routines, the installation of Montage also includes tools used to generate the DAG XML file and download the input image files. Namely, the mDAG and mArchiveExec tools.

The mDAG command generates a DAX XML file containing the description of the workflow in terms of the input files it uses, the tasks, the data dependencies, and the output files produced. This DAX file was then transformed to a xWFL-based one by using the DAX to XWFL tool.

The mArchiveExec command was used to download the input images which were placed in the storage node so that they could be accessed by worker nodes when required.

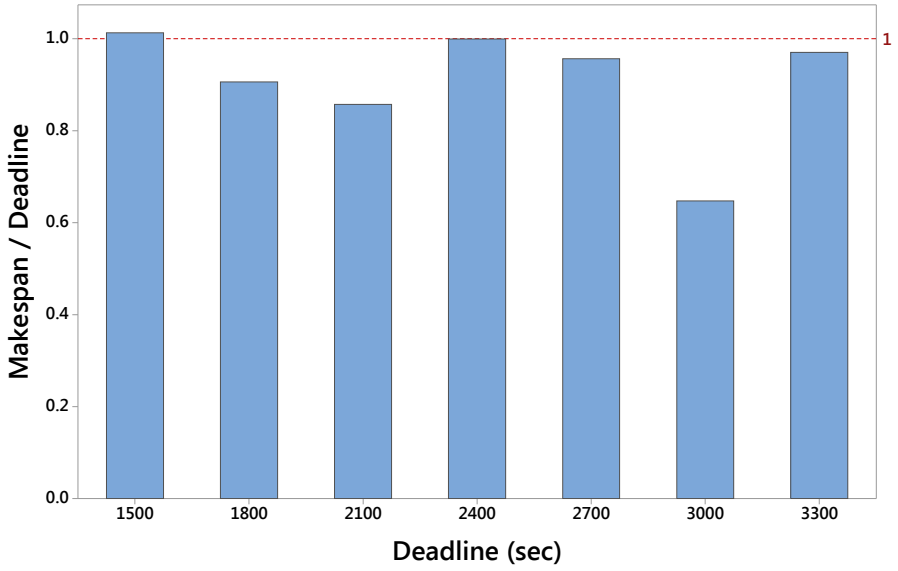


Figure 6.7: Makespan to deadline ratios obtained for the 0.5 degree Montage execution.

6.5 Results

This section presents the results obtained after executing the 0.5 degree Montage workflow on the CloudbusWMS under seven different deadlines.

Figure 6.7 presents the results in terms of the makespan to deadline ratio obtained. The makespan of a workflow is defined as the time it takes for the workflow execution to complete. Ratio values greater than one indicate a makespan larger than the deadline, values equal to one a makespan equal to the deadline, and values smaller than one a makespan smaller than the deadline. Figure 6.8 depicts the actual makespan values obtained for each deadline. The results presented are the average obtained after running the experiments for each deadline 10 times.

The first deadline of 1500 seconds is too strict for the workflow execution to be completed on time. On average, it takes approximately 1520 seconds for the workflow to complete, leading to a ratio of 1.01. This difference between makespan and deadline however is marginal and a 20 second difference is unlikely to have a significant impact

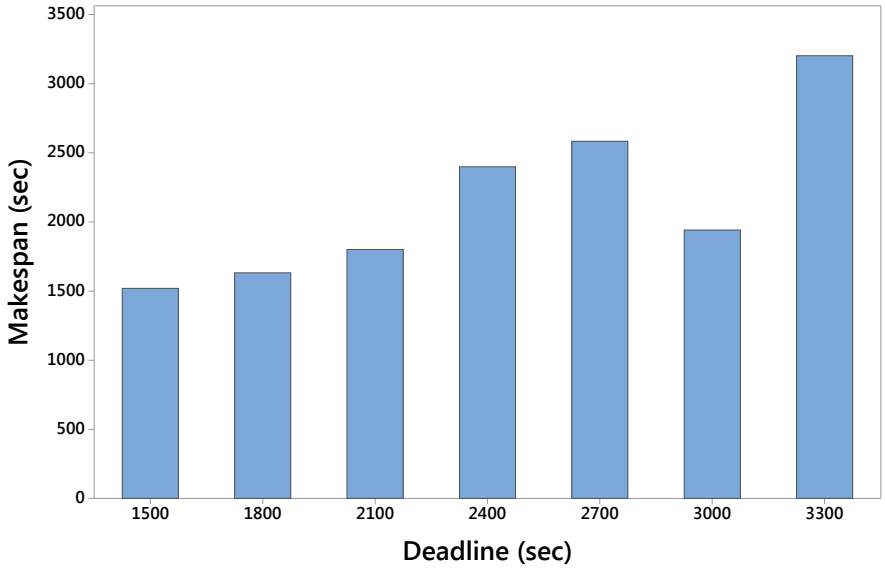


Figure 6.8: Makespan results obtained for the 0.5 degree Montage execution.

on either cost or the usability of the obtained workflow results. The choice of VMs for each deadline interval are presented in Table 6.4. The fact that all of the VMs leased for this deadline interval are of the most powerful VM type (D11), reflects the urgency of the algorithm to complete the workflow execution as fast as possible. The decision to limit the number of VMs to 9 is a direct result of the length of VM provisioning delays. The algorithm recognizes that in some cases it is faster and more efficient to reuse existing VMs rather than leasing new ones.

All of the remaining ratios for the deadlines ranging from 1800 to 3300 seconds are under one. Clearly, 1800 seconds is sufficient for the execution of the workflow to complete. This is achieved by leasing 7 D2 VMs and 2 D11 ones. Once again, the deadline is too strict to lease a larger number of VMs but relaxed enough to not have to lease them all of the most powerful type.

As the deadlines becomes more relaxed, WRPS decides it is more efficient to lease a larger number of VMs of less powerful and cheaper types. For a deadline of 2100 seconds, 13 A1 (small) and 5 D1 VMs are sufficient for the workflow execution to finish well under

Table 6.4: Number of VMs per type leased for the deployment of a 0.5 degree Montage workflow with different deadlines.

Deadline (sec)	A0	A1	A2	D1	D2	D11
1500	-	-	-	-	-	9
1800	-	-	-	-	7	2
2100	-	13	-	5	-	-
2400	17	2	-	1	-	-
2700	17	2	-	-	-	-
3000	17	-	-	2	-	-
3300	18	-	-	-	-	-

the deadline with an average ratio of 0.85. From this deadline onwards, the algorithm can finish the workflow execution on time with minimum cost by taking advantage of the cheapest and least powerful VM, the A0 or extrasmall. By combining this VM type with more powerful ones when necessary, all of the remaining deadlines are met.

Figure 6.9 shows the costs of the execution of the workflow for each of the deadlines. As expected, the most expensive scenario occurs when the deadline is the tightest, that is 1500 seconds. This is a direct result of the provisioning decision to lease the most expensive, and powerful, VM types to finish the execution on time. Overall, except for the deadline of 2400 seconds, the infrastructure cost consistently decreases as the deadlines become more relaxed. The fact that the cost of running the workflow with a deadline of 2400 seconds is cheaper than doing it with a deadline of 2700 can be explained by performance and VM provisioning delay variations.

To demonstrate the auto-scaling features introduced into the Cloudbus WMS, Table 6.5 shows the number of VMs used to run the tasks on each level of the Montage workflow for the 2100 second deadline. For the first level mProjectPP tasks, only one A1 VM is used. WRPS estimates that this configuration will allow the mProjectPP tasks to finish by their assigned deadline. The second level of the workflow contains 73 mDiffFit tasks. Unlike the mProjectPP tasks, these tasks have different starting times, depending on when their mProjectPP parent tasks finished their execution. Based on this, WRPS makes the decision to scale the number of VMs out as mDiffFit tasks become ready for execution. At this point, 13 A1 VMs and 5 D1 VMs, 18 in total, are used to process all the 73 tasks

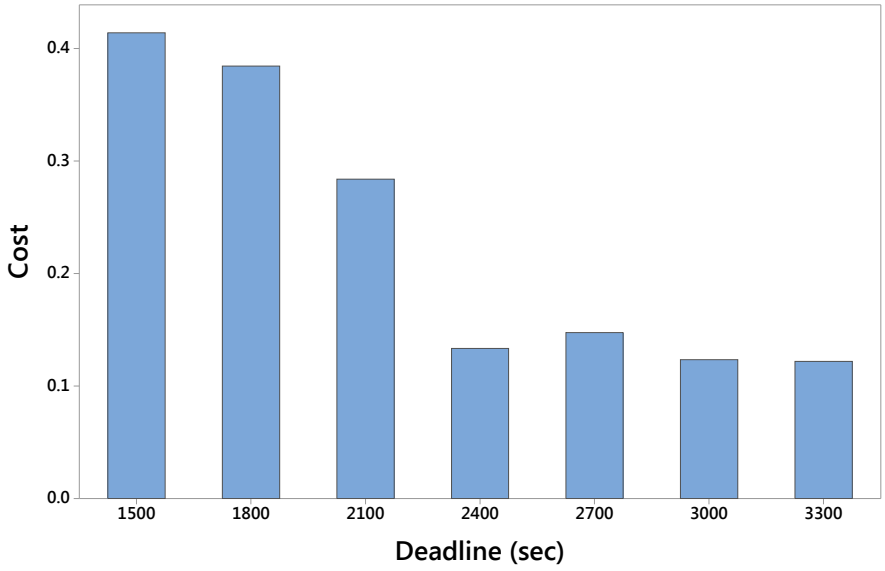


Figure 6.9: Cost results obtained for the 0.5 degree Montage execution.

in the level. Next, the parallelism of the workflow is reduced by a data aggregation task, mConcatFit, and as a result the resource pool is scaled in and only one VM of type A1 is left in the resource pool. The next level contains a single mBGModel task and the VM used in the previous level is reused. For the 32 mBackground tasks, WRPS decides they can finish on time by reusing the existing A1 VM. The remaining levels in the workflow contain a single task and hence there is no need to lease more VMs and the workflow finishes its execution with a single A1 VM.

The experiments presented in this section demonstrate how the elasticity and heterogeneity of cloud resources can be leveraged to meet the QoS requirements of workflow applications. In particular, they demonstrate how the performance of the workflow execution in terms of time as well as the cost of using the cloud infrastructure can be controlled by dynamically scaling the number of resources. This enables scientists to benefit from the flexibility, scalability, and pricing model offered by cloud computing. However, evaluating the performance of the Cloudbus WMS with different scheduling algorithms and with larger scientific workflows that have different data and computational require-

Table 6.5: Number and type of VMs used on the execution of each level of the 0.5 degree Montage workflow with a deadline of 2100 seconds.

Level	Task	Count	A0	A1	A2	D1	D2	D11
1	mProjectPP	32	-	1	-	-	-	-
2	mDiffFit	73	-	13	-	5	-	-
3	mConcatFit	1	-	1	-	-	-	-
4	mBgModel	1	-	1	-	-	-	-
5	mBackground	32	-	1	-	-	-	-
6	mImgTbl	1	-	1	-	-	-	-
7	mAdd	1	-	1	-	-	-	-
8	mShrink	1	-	1	-	-	-	-
9	mJPEG	1	-	1	-	-	-	-

ments and topological structures is an essential future task. In addition to this, as a future work, it is important to evaluate the performance of workflow executions deployed on different cloud providers with different billing periods, VM types, provisioning and deprovisioning delays, and resource performance variations.

6.6 Summary

This chapter presented a reference architecture for cloud WMSs and explained its key components which include a user interface with workflow modeling tools and submission services, a workflow engine capable of making resource provisioning and scheduling decisions, a set of task and resource monitoring tools, and a set of cloud information services that can be queried to retrieve the supported cloud providers and the type of resources they offer. We then introduced a concrete example of an existing system, the Cloudbus WMS, along with our efforts to extend its functionality to support the elastic cloud resource model. Finally, we demonstrated with a practical scenario the use of the enhanced Cloudbus WMS and validated the WRPS algorithm by deploying a Montage workflow on Microsoft Azure resources.

Chapter 7
Conclusions and Future Directions

7.1 Summary

Clouds provide an infrastructure for solving large-scale problems in science. They enable on-demand access to a virtually infinite pool of resources which can be easily configured to suit different application needs. Furthermore, resources can be scaled in and out at any point in time to suit the current application needs and QoS requirements. However, efficiently deploying scientific workflows on these dynamic and heterogeneous platforms is a challenging problem. This thesis addresses the problem of scheduling scientific workflows in cloud computing environments under QoS requirements. The algorithms presented aim to overcome several challenges derived from inherent features of the cloud resource model. These features include on-demand access, elasticity, heterogeneous and abundant resources, utility-based pricing, dynamicity, and uncertainty.

Chapter 1 described the thesis topic by defining the problem addressed as well as the challenges that need to be considered. It also presented key background concepts, motivation, and summarized the key contributions made. To provide a comprehensive understanding of the existing body of knowledge, Chapter 2 surveyed the related work in the area of scientific workflow scheduling in IaaS clouds. It reviewed, compared, and classified more than 24 algorithms. This was done by studying the scheduling, application, and resource models considered by state-of-the-art heuristics.

Chapter 3 presented a combined resource provisioning and scheduling strategy that is based on the meta-heuristic optimization technique, Particle Swarm Optimization. The algorithm aims to minimize the overall workflow execution cost while meeting a

deadline constraint and incorporates basic IaaS cloud principles such as a pay-as-you-go model and heterogeneity of the resources. Although static, the algorithm also incorporates policies to deal with dynamic aspects of cloud environments such as resource performance variation, VM provisioning delays, and elastic resources. The simulation experiments conducted with four well-known workflows demonstrated that the proposed strategy has an overall better performance than two state-of-the-art algorithms, SCS and IC-PCP.

Chapter 4 presented the Workflow Responsive resource Provisioning and Scheduling (WRPS) algorithm, a strategy that finds a balance between making dynamic decisions to respond to changes in the environment and planning ahead to produce better schedules. Its objectives are to minimize the overall cost of the utilized infrastructure while meeting a user-defined deadline. WRPS reduces the workflow into bags of homogeneous tasks and pipelines that share a deadline and models the scheduling of these components as a variation of the unbounded knapsack problem. This problem is then solved in pseudo-polynomial time using dynamic programming. The simulation results demonstrate it is scalable in terms of the number of tasks in the workflow, it is robust and responsive to the cloud performance variability and it is capable of generating better quality solutions than state-of-the-art algorithms. WRPS is successful in meeting deadlines under unpredictable situations involving performance variation, network congestion and inaccurate task size estimations. It achieves this at low costs, even lower than the fully static approaches which have the ability of using the entire workflow structure, comparing various solutions, and choosing the best one before the workflow execution.

Chapter 5 presented BAGS, an adaptive resource provisioning and scheduling algorithm for scientific workflows in clouds capable of generating high quality schedules. The algorithm focuses on emerging finer-grained pricing schemes (e.g. per-minute billing) that give users more flexibility and the ability to reduce the inherent wastage that results from coarser-grained ones. It's objective is to optimize a workflows makespan under a budget constraint; QoS requirement that has been overlooked in favor of optimizing cost under a deadline constraint. Once again, this strategy addressed fundamental challenges of clouds such as resource elasticity, abundance, and heterogeneity, as well as per-

formance variation and virtual machine start-up latency. The simulation results demonstrated the algorithm's responsiveness and ability to generate high-quality schedules that comply with the budget constraint while achieving lower makespans when compared to state-of-the-art algorithms.

Finally, Chapter 6 presented the CloudbusWMS, a tool designed to manage the deployment and execution of workflows in distributed environments. The key features of the system and its main architecture components, specifically those related to the scheduling problem addressed in this thesis, were described in this chapter. The implementation of various cloud-related modules added to the software was detailed and their functionality demonstrated with a case study using the Montage application and the WRPS algorithm.

7.2 Future Directions

This section gives an insight into promising unexplored pathways for future work in the area of scientific workflow scheduling in IaaS clouds.

7.2.1 WaaS Platforms

An interesting and emerging service model is Workflow as a Service (WaaS). This type of platforms offer to manage the execution of scientific workflows submitted by multiple users and hence are directly related to scheduling algorithms designed to process multiple workflows simultaneously. Out of the surveyed algorithms presented in Chapter 2, only a few target this application model. As the popularity and use of cloud computing becomes more widespread, so will services such as WaaS. Therefore, it is important to gain a better understanding and further investigate this type of algorithms. Multiple scenarios can be explored, for instance the WaaS system may acquire a pool of reserved, subscription-based, instances and hence algorithms may be concerned with maximizing their utilization, maximizing the profit of the WaaS provider, and supporting generic QoS requirements to suit the needs of multiple users.

7.2.2 Resource Abundance

The abundance of resources and flexibility to use only those that are required is a clear challenge particular to cloud computing. The difficulty of the provisioning problem under virtually unlimited resources calls for further research in this area. Efficiently utilizing VMs to reduce wastage and energy consumption [69] should be further studied. The Maximum Efficient Reduction [69] (MER) algorithm is a recent step towards this goal. It was proposed as a post-optimization resource efficiency solution and produces a consolidated schedule, based on the original schedule generated by any other algorithm, that optimizes the overall resource usage (minimizes resource wastage) with a minimal makespan increase. Further awareness and efficient techniques to deal with the provisioning and deprovisioning delays of VMs is also necessary. For example, pro-actively starting VMs earlier in the scheduling cycle so that tasks do not have to be delayed due to provisioning times may be a simple way of reducing their impact on the makespan.

7.2.3 Redefinition of QoS Requirements

The unstable performance of cloud resources makes it difficult for algorithms to estimate QoS requirements such as makespan and cost. Defining static constraints as part of QoS requirements has been widely used for other distributed computing paradigms such as parallel systems, clusters, and grids; however, this approach might not be well-suited to environments that are dynamic by nature as are clouds. This creates the need for a reformulation of the definition and meaning of QoS parameters. Instead of defining a static deadline or budget, a variance could be introduced to these values reflecting the resource performance variation and hence offering users more realistic QoS metrics. For example, instead of defining the deadline of an application to be 10 minutes, it could be defined as the range [8, 11] minutes to reflect the dynamic performance of resources. Another way of redefining these parameters could be by introducing a probability which indicates the chances of meeting a user-defined constraint. This could either be static and calculated once before the execution of the workflow or could be dynamically updated as the execution progresses and the state of the environment changes. Zhou et al. [124]

implement a similar approach on their framework Dyna. The clear challenge here is to define new ways of measuring QoS parameters that make sense to both, users and schedulers, so that it leads to more efficient schedules with higher accuracy in terms of meeting constraints while achieving other scheduling objectives.

7.2.4 Dynamic Performance Prediction and Runtime Estimation

Predicting the performance of resources allows for algorithms to estimate the runtime of tasks. These estimated runtimes are then used to make scheduling and provisioning decisions and the accuracy of the schedule greatly depends on the accuracy of the prediction. Current algorithms only consider a snapshot value of the performance of a resource and assume it is static through out the execution of the task. The variability and uncertainty surrounding the performance of cloud resources means that using a static value as an estimate for the performance of a resource is no longer valid. The dynamic nature of clouds suggests that performance prediction should also be dynamic in order to get more accurate runtime estimations. Performance models could be built based on past usage metrics, the dynamic monitoring of the resource performance, and time series analysis on future trends for example. Algorithms could benefit from either implementing their own strategies or making use of existing dynamic performance prediction frameworks. The ideal scenario would be to develop algorithms that are less reliant on runtime estimations and are capable of producing high-quality schedules even if the performance of the prediction or estimation techniques is poor. It is worthwhile noting that better estimating the performance of resources would imply an overhead and could possibly have a negative impact on the performance of the scheduling algorithm. For this reason, the tradeoff between performance and accuracy has to be considered when designing heuristics based on the application scenario being targeted.

7.2.5 Resource Elasticity and Performance Heterogeneity

A fundamental problem of cloud environments is that there is no guarantee on the performance of an application; it can change unpredictably and users have no control over

this. The ability to scale horizontally does not only provide aggregated performance, but also a way of dealing with the potential indeterminism of a workflow's execution. Algorithms should transparently provision additional resources to achieve the performance that would have been achieved if the application was running in isolation, with no performance interference; in other words, resource allocation has to be tuned in order to mitigate the effects of performance variation. But, allocating more resources to compensate for a lack of performance on others might not guarantee that the QoS requirements are achieved. Reinforcement learning [27] is another technique that could be explored in this area; it could allow algorithms to find a balance between exploitation (continued use of an instance) and exploration (launching new instances) [48]. For example, dynamic algorithms could launch an initial set of instances during the first scheduling cycle. For successive cycles, the performance of the already-leased machines could be analyzed and only those rated as high-performers would be kept while the low-performing ones would be shut down or re-launched if required on the next cycle. These algorithms would then be aiming to find and use high performing VMs for as long as possible to increase the overall performance of the system. This strategy would require a way of judging or classifying the performance of a VM as "high" or "low". One way to achieve this could be by exploiting the knowledge about the specific infrastructure of the IaaS provider or the historical performance of instances. Another solution could be based on the estimation of the average performance of the leased instances. In any case, algorithms would need to be extended to include heuristics to decide how to or when to exploit the scalability and elasticity of the cloud in order to mitigate the impact of performance variation and increase the accuracy in terms of meeting QoS requirements.

7.2.6 VM features

Most of the researched algorithms seem to target compute-intensive applications as they define VM types in terms of their CPU processing capacity. VM features such as memory and I/O performance are not considered. While this approach is valid for certain workflow applications, others with more demanding requirements need algorithms that ensure the provisioned VMs are capable of executing their tasks. For example, analytics

workflows that perform fit-in-memory machine learning algorithms might have specific RAM requirements that VMs need to meet in order for the tasks to run successfully. Exploring how the cost and execution times are affected when considering a more comprehensive definition of a VM is another topic of interest. Since most providers design VM types to suit certain types of applications and price them accordingly, algorithms might benefit from choosing the best type of VM for a given task based on its characteristics.

7.2.7 Big Data and Streaming Workflows

The growth in the development and adoption of sensor networks and ubiquitous computing sees a change in the data requirements of applications. Many workflows support the processing of real time data produced by sensor or distributed devices. Scheduling algorithms need to be able to handle these large amounts of data that are potentially arriving at a continuous rate for processing. This need to support big data and streaming applications calls for algorithms to develop heuristics that determine the best way of storing, transferring, accessing, and processing such data considering the characteristics of clouds. In this context for example, it is important to explore how the data is handled during the workflow execution. Given the bandwidth performance variation and cost of using the network infrastructure of a cloud provider, it is important to explore different ways of managing the input, intermediate, and output data of a workflow. The majority of state of the art algorithms transfer input and output data directly between VM instances as required, which may result in higher costs and delays by transferring large amounts of data that can some times be redundant. Other approaches better suited for clouds might consider the best place or way to store the data, where to place tasks so that the data transfer is minimized, and when and where to replicate data in order to achieve the QoS requirements.

7.2.8 Energy Efficient Algorithms

Individuals, organizations and governments worldwide have developed an increased concern to reduce carbon footprints in order to lessen the impact on the environment.

The cloud infrastructure relies on data centers that are not only expensive to maintain but also consume a vast amount of energy. A single 300W server consumes approximately 2628 kWh of energy per year [31] and data centers are generally composed of thousands of computing servers. In fact, data centers consume on average as much energy as 25,000 households [65] and in 2006 were accounted for 1.5% of the total US electricity consumption [32]. In addition, increased electricity prices mean that energy efficient data centers are also capable of considerably reducing costs and therefore becoming more profitable. For instance, a rough calculation done in 2009 estimated that Google consumes more than $38M worth of electricity per year [94]. Therefore, cloud providers need to have strategies in place in order to reduce their environmental impact and reduce energy-related costs.

There are various techniques such as sleep scheduling and resource virtualization that improve the energy efficiency in cloud computing [72]. However, algorithms managing the execution of compute, memory, and network intensive tasks can significantly aid in reducing the overall energy consumed by the data center if designed properly. Therefore, workflow scheduling algorithms deployed on clouds need to find a balance between energy efficiency and high service level performance instead of focusing solely on meeting performance targets. In particular, algorithms need to use resources and allocate tasks to them not only to satisfy the QoS requirements specified by users, but also to reduce the amount of energy consumed.

7.2.9 Security

Some scientific workflow applications may require that the input or output data are handled in a secure manner. Even more, some tasks might be composed of sensitive computations that need to be kept secure. Security in the cloud is a well-known challenge [98] but in recent years, many providers have put security monitoring systems and policies in place that are capable of protecting the users' data and applications as well as the facilities and infrastructure where these are processed and stored. On top of these basic security measures, many providers offer additional ones, for example, in the case of Amazon EC2, users have access to additional services such as secure access to the APIs through HTTPS,

built-in firewalls, encrypted data storage and access to security logs, among many others. In many cases, all of these security measures make cloud data centers more secure than privately owned ones. Although this is an advantage, applications need to use and configure the security service offerings properly in order to meet specific requirements. With this in mind, workflow scheduling algorithms could leverage different security policies, tools, and offerings in order to meet the application's security requirements. For instance, sensitive tasks and data can be managed in such a way that either resources or providers with a higher security ranking are used to execute and store them or on which additional security systems are installed. Another way of dealing with workflow applications with high security requirements is to deploy them in hybrid clouds (the combination of a private cloud owned and used by a single organization and one or more public clouds). In this way, data and tasks with specific security needs can be kept in the private cloud where a fine grained control of security measures can be achieved while less sensitive computations can be performed in public cloud resources.

Bibliography

[1] "Amazon EC2 Spot Instances," https://aws.amazon.com/ec2/purchasing-options/spot-instances/, accessed: October 2015.

[2] "Amazon Simple Storage Service," http://aws.amazon.com/s3/, accessed: October 2015.

[3] "Apache Jclouds," http:// jclouds.apache.org, accessed: October 2015.

[4] "Cloudsigma," https://www.cloudsigma.com, accessed: October 2015.

[5] "DAX," https://pegasus.isi.edu/documentation/creating_workflows.php, accessed: October 2015.

[6] "Elastic Compute Cloud EC2," http://aws.amazon.com/ec2, accessed: October 2015.

[7] "Google App Engine," https://cloud.google.com/appengine, accessed: October 2015.

[8] "Google Cloud Storage," https://cloud.google.com/storage/, accessed: October 2015.

[9] "Google Compute Engine," https://cloud.google.com/products/compute-engine/, accessed: October 2015.

[10] "Google Compute Engine Pricing," https://developers.google.com/compute/pricing, accessed: October 2015.

[11] "Gravitational Waves Detected 100 Years After Einstein's Prediction," https://www.ligo.caltech.edu/news/ligo20160211, accessed: October 2015.

[12] "LIGO Detection Portal," https://www.ligo.caltech.edu/detection, accessed: October 2015.

[13] "Microsoft Azure," https://azure.microsoft.com, accessed: October 2015.

[14] "National Centre for Biotechnology Information," http://www. ncbi.nlm.nih.gov, accessed: 2014-08-10.

[15] "Pegasus," https://pegasus.isi.edu/, accessed: October 2015.

[16] "Rackspace Block Storage," http://www.rackspace.com.au/cloud/block-storage, accessed: October 2015.

[17] "Salesforce.com," http://www.salesforce.com/au/saas, accessed: October 2015.

[18] "Southern California Earthquake Center," https://www.scec.org/, accessed: October 2015.

[19] "USC Epigenome Center," http://epigenome.usc.edu, accessed: October 2015.

[20] B. Abbott, R. Abbott, T. Abbott, M. Abernathy, F. Acernese, K. Ackley, C. Adams, T. Adams, P. Addesso, R. Adhikari *et al.*, "Observation of gravitational waves from a binary black hole merger," *Physical Review Letters*, vol. 116, no. 6, p. 061102, 2016.

[21] A. Abramovici, W. E. Althouse, R. W. Drever, Y. Gürsel, S. Kawamura, F. J. Raab, D. Shoemaker, L. Sievers, R. E. Spero, K. S. Thorne *et al.*, "Ligo: The laser interferometer gravitational-wave observatory," *Science*, vol. 256, no. 5055, pp. 325–333, 1992.

[22] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for Infrastructure as a Service Clouds," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 158–169, 2013.

[23] R. Andonov, V. Poirriez, and S. Rajopadhye", "Unbounded knapsack problem: Dynamic programming revisited," *European Journal of Operational Research*, vol. 123, no. 2, pp. 394 – 407, 2000.

[24] R. Andonov and S. Rajopadhye, "A sparse knapsack algo-tech-cuit and its synthesis," in *Proceedings of the IEEE International Conference on Application Specific Array Processors*, 1994.

[25] H. Arabnejad, J. G. Barbosa, and R. Prodan, "Low-time complexity budget–deadline constrained workflow scheduling on heterogeneous resources," *Future Generation Computer Systems*, vol. 55, pp. 29–40, 2016.

[26] A. Barker and J. Van Hemert, "Scientific workflow: a survey and research directions," in *Parallel Processing and Applied Mathematics*. Springer, 2008, pp. 746–753.

[27] A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 1998.

[28] F. Berman, G. Fox, and A. J. Hey, *Grid computing: making the global infrastructure a reality*. John Wiley and sons, 2003, vol. 2.

[29] G. Berriman, A. Laity, J. Good, J. Jacob, D. Katz, E. Deelman, G. Singh, M. Su, and T. Prince, "Montage: The architecture and scientific applications of a national virtual observatory service for computing astronomical image mosaics," in *Proceedings of Earth Sciences Technology Conference*, 2006.

[30] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Proceedings of the Third Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2008, pp. 1–10.

[31] R. Bianchini and R. Rajamony, "Power and energy management for server systems," *IEEE Computer*, vol. 37, no. 11, pp. 68–76, 2004.

[32] R. Brown, "Report to congress on server and data center energy efficiency: Public law 109-431," 2008.

[33] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud Computing: Principles and Paradigms*. Wiley. com, 2010, vol. 87.

[34] E.-K. Byun, Y.-S. Kee, J.-S. Kim, E. Deelman, and S. Maeng, "Bts: Resource capacity estimate for time-targeted science workflows," *Journal of Parallel and Distributed Computing*, vol. 71, no. 6, pp. 848–862, 2011.

[35] E.-K. Byun, Y.-S. Kee, J.-S. Kim, and S. Maeng, "Cost optimized provisioning of elastic resources for application workflows," *Future Generation Computer Systems*, vol. 27, no. 8, pp. 1011–1026, 2011.

[36] R. Calheiros and R. Buyya, "Meeting deadlines of scientific workflows in public clouds with tasks replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 7, pp. 1787–1796, July 2014.

[37] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[38] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[39] W.-N. Chen and J. Zhang, "An ant colony optimization approach to a grid workflow scheduling problem with various qos requirements," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 39, no. 1, pp. 29–43, 2009.

[40] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow management in condor," in *Workflows for e-Science*. Springer, 2007, pp. 357–375.

[41] D. de Oliveira, K. A. Ocaña, F. Baião, and M. Mattoso, "A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds," *Journal of Grid Computing*, vol. 10, no. 3, pp. 521–552, 2012.

[42] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[43] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*. IEEE Press, 2008, p. 50.

[44] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[45] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[46] J. J. Durillo and R. Prodan, "Multi-objective workflow scheduling in amazon ec2," *Cluster Computing*, vol. 17, no. 2, pp. 169–189, 2014.

[47] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek, "Askalon: A grid application development and computing environment," in *Proceedings of the Sixth IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2005, pp. 122–131.

[48] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, "More for your money: exploiting performance heterogeneity in public clouds," in *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC)*. ACM, 2012, p. 20.

[49] P. C. Fishburn, *Interval orders and interval graphs: A study of partially ordered sets*. John Wiley & Sons, 1985.

[50] M. E. Frincu, S. Genaud, J. Gossa *et al.*, "Comparing provisioning and scheduling strategies for workflows on clouds," in *Proceedings of the Third International Workshop on Workflow Models, Systems, Services and Applications in the Cloud (CloudFlow)*. IEEE, 2013.

[51] Y. Fukuyama and Y. Nakanishi, "A particle swarm optimization for reactive power and voltage control considering voltage stability," in *Proceedings of the eleventh IEEE International Conference on Intelligent System Applications to Power Systems*, 1999, pp. 117–121.

[52] T. A. Genez, L. F. Bittencourt, and E. R. Madeira, "Workflow scheduling for SaaS/PaaS cloud providers considering two sla levels," in *Proceedings of the IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2012, pp. 906–912.

[53] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers, "Examining the challenges of scientific workflows," *IEEE Computer*, vol. 40, no. 12, pp. 26–34, 2007.

[54] P. C. Gilmore and R. E. Gomory, "A linear programming approach to the cutting stock problem-part ii," *Operations Research*, vol. 11, no. 6, pp. 863–888, 1963.

[55] P. Gilmore and R. Gomory, "The theory and computation of knapsack functions," *Operations Research*, vol. 14, no. 6, pp. 1045–1074, 1966.

[56] R. Graves, T. H. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner *et al.*, "Cybershake: A physics-based seismic hazard model for southern california," *Pure and Applied Geophysics*, vol. 168, no. 3-4, pp. 367–381, 2011.

[57] A. Gupta and D. Milojicic, "Evaluation of hpc applications on cloud," in *Open Cirrus Summit (OCS), 2011 Sixth*, Oct 2011, pp. 22–26.

[58] J. O. Gutierrez-Garcia and K. M. Sim, "A family of heuristics for agent-based elastic cloud bag-of-tasks concurrent scheduling," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1682–1699, 2013.

[59] G. M. Harry, L. S. Collaboration *et al.*, "Advanced ligo: the next generation of gravitational wave detectors," *Classical and Quantum Gravity*, vol. 27, no. 8, p. 084006, 2010.

[60] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations research*, vol. 9, no. 6, pp. 841–848, 1961.

[61] T. T. Huu and J. Montagnat, "Virtual resources allocation for workflow-based applications distribution on a cloud infrastructure," in *Proceedings of the Tenth IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2010, pp. 612–617.

[62] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 931–945, June 2011.

[63] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Proceedings of the IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2010, pp. 159–168.

[64] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.

[65] J. M. Kaplan, W. Forrest, and N. Kindler, "Revolutionizing data center energy efficiency," *McKinsey & Company, Tech. Rep*, 2008.

[66] J. Kennedy, R. Eberhart *et al.*, "Particle swarm optimization," in *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, vol. 4. Perth, Australia, 1995, pp. 1942–1948.

[67] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[68] A. Lazinica, *Particle swarm optimization*. InTech Kirchengasse, 2009.

[69] Y. C. Lee, H. Han, A. Y. Zomaya, and M. Yousif, "Resource-efficient workflow scheduling in clouds," *Knowledge-Based Systems*, vol. 80, pp. 153–162, 2015.

[70] Y. C. Lee and A. Y. Zomaya, "Stretch out and compact: Workflow scheduling with resource abundance," in *Proceedings of the Thirteenth IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2013, pp. 219–226.

[71] H.-H. Li, Y.-W. Fu, Z.-H. Zhan, and J.-J. Li, "Renumber strategy enhanced particle swarm optimization for cloud computing resource scheduling," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2015, pp. 870–876.

[72] J. Liu, F. Zhao, X. Liu, and W. He, "Challenges towards elastic power management in internet data centers," in *Proceedings of the Twenty Ninth IEEE International Conference on Distributed Computing Systems Workshops (ICDCS Workshops)*. IEEE, 2009, pp. 65–72.

[73] J. Livny, H. Teonadi, M. Livny, and M. K. Waldor, "High-throughput, kingdom-wide prediction and annotation of bacterial non-coding rnas," *PloS one*, vol. 3, no. 9, p. e3197, 2008.

[74] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghan, D. Okaya *et al.*, "SCEC CyberShake workflowsautomating probabilistic seismic hazard analysis calculations," in *Workflows for e-Science*. Springer, 2007, pp. 143–163.

[75] M. Malawski, K. Figiela, M. Bubak, E. Deelman, and J. Nabrzyski, "Scheduling multilevel deadline-constrained scientific workflows on clouds based on cost optimization," *Scientific Programming*, vol. 2015, 2015.

[76] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, "Cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. EEE, 2012, p. 22.

[77] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2011, p. 49.

[78] ——, "A performance study on the VM startup time in the cloud," in *Proceedings of the Fifth IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2012, pp. 423–430.

[79] ——, "Scaling and scheduling to maximize application performance within budget constraints in cloud workflows," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2013, pp. 67–78.

[80] S. Martello and P. Toth, *Knapsack problems*. Wiley New York, 1990.

[81] E. W. Mayr and H. Stadtherr, "Optimal parallel algorithms for two processor scheduling with tree precedence constraints," 1995.

[82] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[83] E. Michon, J. Gossa, S. Genaud *et al.*, "Free elasticity and free CPU power for scientific workloads on IaaS clouds," in *Proceedings of the Eighteen IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2012, pp. 85–92.

[84] Microsoft, "Microsoft azure," Nov 2015. [Online]. Available: https://azure.microsoft.com

[85] J. Nabrzyski, J. M. Schopf, and J. Weglarz, *Grid resource management: state of the art and future trends*. Springer Science & Business Media, 2012, vol. 64.

[86] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of ec2 cloud computing services for scientific computing," in *Cloud computing*. Springer, 2009, pp. 115–131.

[87] C. O. Ourique, E. C. Biscaia, and J. C. Pinto, "The use of particle swarm optimization for dynamical analysis in chemical processes," *Computers & Chemical Engineering*, vol. 26, no. 12, pp. 1783–1793, 2002.

[88] S. Pandey, D. Karunamoorthy, and R. Buyya, "Workflow engine for clouds," *Cloud Computing: Principles and Paradigms*, pp. 321–344, 2011.

[89] S. Pandey, L. Wu, S. M. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *Proceedings of the Twenty Fourth IEEE International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2010, pp. 400–407.

[90] C. Pautasso and G. Alonso, "Parallel computing patterns for grid workflows," in *Proceedings of the Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2006, pp. 1–10.

[91] I. Pietri, M. Malawski, G. Juve, E. Deelman, J. Nabrzyski, and R. Sakellariou, "Energy-constrained provisioning for scientific workflow ensembles," in *Proceedings of the Third International Conference on Cloud and Green Computing (CGC)*. IEEE, 2013, pp. 34–41.

[92] D. Poola, S. K. Garg, R. Buyya, Y. Yang, and K. Ramamohanarao, "Robust scheduling of scientific workflows with deadline and budget constraints in clouds," in *Proceedings of the Twenty-eighth IEEE International Conference on Advanced Information Networking and Applications (AINA)*, 2014, pp. 1–8.

[93] D. Poola, K. Ramamohanarao, and R. Buyya, "Fault-tolerant workflow scheduling using spot instances on clouds," *Procedia Computer Science*, vol. 29, pp. 523–533, 2014.

[94] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs, "Cutting the electric bill for internet-scale systems," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 123–134, 2009.

[95] M. Rahman, S. Venugopal, and R. Buyya, "A dynamic critical path algorithm for scheduling scientific workflow applications on global grids," in *Proceedings of the IEEE International Conference on e-Science and Grid Computing*. IEEE, 2007, pp. 35–42.

[96] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE Transactions on Cloud Computing*, vol. 2, no. 2, pp. 222–235, April 2014.

[97] ——, "A responsive knapsack-based algorithm for resource provisioning and scheduling of scientific workflows in clouds," in *Proceedings of the Fourty-fourth International Conference on Parallel Processing (ICPP)*, vol. 1. IEEE, 2015, pp. 839–848.

[98] C. Rong, S. T. Nguyen, and M. G. Jaatun, "Beyond lightning: A survey on security challenges in cloud computing," *Computers & Electrical Engineering*, vol. 39, no. 1, pp. 47–54, 2013.

[99] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[100] T. Sousa, A. Silva, and A. Neves, "Particle swarm based data mining algorithms for classification tasks," *Parallel Computing*, vol. 30, no. 5, pp. 767–783, 2004.

[101] S. Stadill, "By the numbers: How google compute engine stacks up to amazon ec2," https://gigaom.com/2013/03/15/by-the-numbers-how-google-compute-engine-stacks-up-to-amazon-ec2/.

[102] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74.

[103] R. Tolosana-Calasanz, J. Á. BañAres, C. Pham, and O. F. Rana, "Enforcing qos in scientific workflow systems enacted over cloud infrastructures," *Journal of Computer and System Sciences*, vol. 78, no. 5, pp. 1300–1315, 2012.

[104] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[105] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[106] C. Valentin, D. Ciprian, S. Corina, P. Florin, and C. Alexandru, "Large-scale distributed computing and applications: Models and trends," 2010.

[107] S. Venugopal, K. Nadiminti, H. Gibbins, and R. Buyya, "Designing a resource broker for heterogeneous grids," *Software: Practice and Experience*, vol. 38, no. 8, pp. 793–825, 2008.

[108] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *Proceedings of the Twelfth IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2012, pp. 612–619.

[109] J.-S. Vöckler, G. Juve, E. Deelman, M. Rynge, and B. Berriman, "Experiences using cloud computing for a scientific workflow application," in *Proceedings of the 2nd international workshop on Scientific cloud computing*. ACM, 2011, pp. 15–24.

[110] J. Wang, P. Korambath, I. Altintas, J. Davis, and D. Crawl, "Workflow as a service in the cloud: architecture and scheduling algorithms," *Procedia Computer Science*, vol. 29, pp. 546–556, 2014.

[111] C. Q. Wu, X. Lin, D. Yu, W. Xu, and L. Li, "End-to-end delay minimization for scientific workflows in clouds under budget constraint," *IEEE Transactions on Cloud Computing*, vol. 3, no. 2, pp. 169–181, 2015.

[112] Z. Wu, Z. Ni, L. Gu, and X. Liu, "A revised discrete particle swarm optimization for cloud workflow scheduling," in *Proceedings of the International Conference on Computational Intelligence and Security (CIS)*. IEEE, 2010, pp. 184–188.

[113] M. Xu, L. Cui, H. Wang, and Y. Bi, "A multiple QoS constrained scheduling strategy of multiple workflows for cloud computing," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. IEEE, 2009, pp. 629–634.

[114] S. Yassa, R. Chelouah, H. Kadima, and B. Granado, "Multi-objective approach for energy-aware workflow scheduling in cloud computing environments," *The Scientific World Journal*, vol. 2013, 2013.

[115] U. Yildiz, A. Guabtni, and A. H. Ngu, "Business versus scientific workflows: A comparative study," in *Proceedings of the Fourth World Conference on Services (SERVICES)*. IEEE, 2009, pp. 340–343.

[116] J. Yu and R. Buyya, "A novel architecture for realizing grid workflow using tuple spaces," in *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*. IEEE, 2004, pp. 119–128.

[117] ——, "A budget constrained scheduling of workflow applications on utility grids using genetic algorithms," in *Proceedings of the Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2006, pp. 1–10.

[118] J. Yu, R. Buyya, and K. Ramamohanarao, "Workflow scheduling algorithms for grid computing," in *Metaheuristics for Scheduling in Distributed Computing Environments*. Springer, 2008, pp. 173–214.

[119] J. Yu, R. Buyya, and C. K. Tham, "Cost-based scheduling of scientific workflow applications on utility grids," in *Proceedings of the First International Conference on e-Science and Grid Computing*. IEEE, 2005, pp. 8–pp.

[120] J. Yu, K. Ramamohanarao, and R. Buyya, "Deadline/budget-based scheduling of workflows on utility grids," *Market-Oriented Grid and Utility Computing*, pp. 427–450, 2009.

[121] L. Zeng, B. Veeravalli, and X. Li, "Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud," in *Proceedings*

of the Twenty-sixth IEEE International Conference on Advanced Information Networking and Applications (AINA). IEEE, 2012, pp. 534–541.

[122] ——, "Saba: A security-aware and budget-aware workflow scheduling strategy in clouds," *Journal of Parallel and Distributed Computing*, vol. 75, pp. 141–151, 2015.

[123] F. Zhang and M. Sakr, "Performance variations in resource scaling for mapreduce applications on private and public clouds," in *Proceedings of the International Conference on Cloud Computing (CLOUD)*, 2014.

[124] A. Zhou, B. He, and C. Liu, "Monetary cost optimizations for hosting workflow-as-a-service in iaas clouds," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2015.

[125] A. C. Zhou and B. He, "Transformation-based monetary cost optimizations for workflows in the cloud," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2014.

[126] Q. Zhu, J. Zhu, and G. Agrawal, "Power-aware consolidation of scientific workflows in virtualized environments," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–12.

[127] Z. Zhu, G. Zhang, M. Li, and X. Liu, "Evolutionary multi-objective workflow scheduling in cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1344–1357, 2016.