

UNIVERSIDADE FEDERAL FLUMINENSE

LUAN TEYLO GOUVEIA LIMA

**Scheduling Deadline Constrained Bag-of-Tasks in
Cloud Environments using Hibernation prone Spot
Instances**

NITERÓI

2021

UNIVERSIDADE FEDERAL FLUMINENSE

LUAN TEYLO GOUVEIA LIMA

**Scheduling Deadline Constrained Bag-of-Tasks in
Cloud Environments using Hibernation prone Spot
Instances**

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Doctor of Science. Topic Area: Computer Science.

Advisor:

LÚCIA MARIA DE ASSUMPÇÃO DRUMMOND

NITERÓI

2021

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

L732s Lima, Luan Teylo Gouveia
Scheduling Deadline Constrained Bag-of-Tasks in Cloud
Environments using Hibernation prone Spot Instances / Luan
Teylo Gouveia Lima ; Lúcia Maria de Assumpção Drummond,
orientadora. Niterói, 2021.
112 f.

Tese (doutorado)-Universidade Federal Fluminense, Niterói,
2021.

DOI: <http://dx.doi.org/10.22409/PGC.2021.d.02852177129>

1. Escalonamento de tarefa. 2. Computação em nuvem. 3.
Produção intelectual. I. Drummond, Lúcia Maria de
Assumpção, orientadora. II. Universidade Federal Fluminense.
Instituto de Computação. III. Título.

CDD -

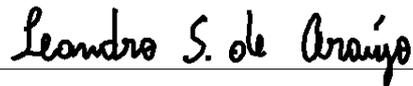
LUAN TEYLO GOUVEIA LIMA

Scheduling Deadline Constrained Bag-of-Tasks in Cloud Environments using
Hibernation prone Spot Instances

Approved on March 26th, 2021 by:



Dr. Lúcia M. A. Drummond, D.Sc. / IC / Universidade
Federal Fluminense (President)



Dr. Leandro Santiago de Araújo , D.Sc. / IC / Universidade
Federal Fluminense



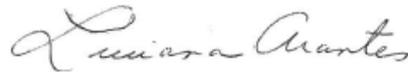
Dr. Maria Cristina Silva Boeres, Ph.D. / IC / Universidade
Federal Fluminense



Dr. Alba Cristina M. A. de Melo, Ph.D. / CIC /
Universidade de Brasília



Dr. Laurent Lefèvre, Ph.D. / Inria / Université de Lyon



Dr. Luciana Arantes, Ph.D. / LiP6 / Sorbonne Université



Dr. Pierre Sens, Ph.D. / LiP6 / Sorbonne Université

Niterói

2021

“We are at the very beginning of time for the human race. It is not unreasonable that we grapple with problems. But there are tens of thousands of years in the future. Our responsibility is to do what we can, learn what we can, improve the solutions, and pass them on.”

Richard P. Feynman

To my wife, Pamela.

Agradecimentos

À professora Lúcia Drummond, pela orientação e por toda a confiança depositada na elaboração deste trabalho. Sou grato pela oportunidade de trabalhar com uma pessoa tão comprometida com o crescimento intelectual de seus alunos.

À professora Luciana Arantes, pelo acolhimento durante o meu período de doutorado sanduíche e pelas várias ideias que fizeram deste um trabalho do qual me orgulho muito. Sem a sua ajuda o meu desempenho e os resultados não seriam os mesmos.

To Professor Pierre Sens for all the help and patience in his explanations. I am grateful for the dedication you had with me during the period I was in LiP6 and for you always have time to clarify my doubts.

Aos meus pais, Miguel e Santiaga, por toda dedicação, carinho e paciência. Agradeço pelo apoio e pela alegria com que vocês recebem as notícias que tenho para dar, mesmo quando significam que haverá uma distância física ainda maior entre a gente.

Ao meu irmão Miguel Júnior, que não cansa de me surpreender com a sua maturidade e inteligência. Você ainda continua sendo o melhor presente que meus pais me deram.

À Pamela, com quem compartilho as minhas conquistas ao longo de 12 anos. Obrigado pelo relacionamento duradouro e construtivo.

Ao Elcio, Betânia, Tatiana e dona Dalvina, pelo acolhimento e por todo o apoio. Sem vocês tudo seria mais difícil. Sou imensamente grato por fazer parte da família.

Aos meus companheiros de laboratório Pablo, Maicon, Rodrigo e Rafaela, que entre cafés e conversas, me proporcionaram um ambiente de trabalho rico e prazeroso. A todos os professores e colegas que de alguma forma colaboraram para a minha formação.

Ao CNPq, pela bolsa concedida nos dois primeiros anos do meu doutorado. À FAPERJ pela bolsa Aluno Nota 10, concedida nos dois últimos anos do meu doutorado. À CAPES, pela bolsa de doutorado-sanduíche, concedida por meio do seu Programa Institucional de Internacionalização (CAPES/PrInt). Ao CNPq e AWS pelo projeto BioCloud, que disponibilizou verbas para a utilização de recursos de nuvem.

Resumo

Os principais provedores de nuvens computacionais oferecem diversos tipos de máquinas virtuais (MVs) em distintos modelos de contratação, com diferentes garantias e graus de confiabilidade, dos quais os mais populares são o sob demanda (*on-demand*) e o *spot*. As MVs *on-demand* são alocadas por um custo fixo por unidade de tempo, e sua disponibilidade é garantida pelo provedor durante toda a execução da aplicação. Já as instâncias *spot* são oferecidas com um grande desconto monetário, quando comparadas com as *on-demand*. Contudo, tanto a disponibilidade quanto a confiabilidade variam de acordo com a demanda de recursos enfrentada pelo provedor. Ademais, as instâncias *spot* podem ser encerradas ou hibernadas a qualquer momento, caso o provedor necessite de seus recursos computacionais.

Em novembro de 2017, a *Amazon Web Services* (AWS) introduziu o recurso de hibernação em MVs *spot*. Agora, em vez de encerrar as MVs, o provedor pode hiberná-las temporariamente. Nesta tese, estamos interessados na execução aplicações *Bag-of-Tasks* sujeitas a um *deadline*. Assim, propomos o *Hibernation Aware Dynamic Scheduler* (HADS), um *framework* modular e leve que pode ser facilmente atualizado para atender a novos requisitos. O HADS inclui uma série de funções integradas, como balanceamento de cargas (por meio de um procedimento de *work-stealing*), *checkpoint* nativo e procedimentos de migração e recuperação de tarefas. Ao contrário de outros *frameworks* que lidam com a rescisão/revogação de MVs *spot*, o HADS explora o recurso de hibernação dessas máquinas para minimizar os custos monetários de execução, gerenciando todo o ciclo de vida da aplicação.

Além dos modelos de contratação, os provedores de nuvem introduziram recentemente o conceito de MVs *burstable*. Essas MVs lidam com as variações da carga de trabalho das aplicações, aumentando a sua capacidade de processamento durante um período limitado de tempo. As MVs *burstable* são oferecidas no mercado *on-demand* com um grande desconto em comparação a MVs com poder de processamento equivalente. Assim, neste trabalho, também apresentamos o Burst-HADS, uma extensão do HADS que explora instâncias *burstable* para minimizar o tempo total de execução da aplicação. O Burst-HADS é um *framework* multi-objetivo que minimiza o custo monetário da execução e o tempo de execução das aplicações. Portanto, esta tese apresenta o HADS e Burst-HADS e todos os estudos realizados ao longo do seu desenvolvimento.

Palavras-chave: Computação em Nuvem, Escalonamento, Máquinas *spot*, Checkpoint.

Abstract

Leading cloud providers offer several types of Virtual Machines (VMs) with different guarantees in terms of availability and volatility. Moreover, the same resource is provided through multiple pricing models, also called markets. Among them, the most popular markets are on-demand and the spot. In the on-demand market, the VMs are allocated for a fixed cost per time unit, and the provider ensures their availability during the whole execution. On the other hand, in the spot market, VMs are offered with a huge discount when compared to the on-demand ones, but their availability fluctuates according to the cloud's current demand. Different from the on-demand VMs, the cloud provider can terminate spot VMs at any time.

In November 2017, Amazon Web Services (AWS) introduced the hibernation feature on VMs from the spot market. Now, instead of terminating the spot VMs, the provider can hibernate them temporarily. In this thesis, we are interested in executing Bag-of-Task applications subject to a deadline on cloud environments. Thus, we propose the Hibernation Aware Dynamic Scheduler (HADS), a modular and lightweight framework that can be easily upgraded to meet new requirements. HADS includes a series of built-in functions such as load balance (through a work-stealing procedure), native checkpoint, and recovery procedures. Unlike other frameworks that cope with the termination/revocation of spot VMs, HADS explores the hibernation feature of spot VMs to minimize execution monetary costs. As a matter of fact, HADS manages all life-cycle of BoT applications and schedules the tasks on the given VM instances.

Besides the hibernation feature, cloud providers have also introduced in recent years the concept of burstable VMs. Those VMs can burst up their respective baseline CPU performance during a limited period of time. Burstable VMs are offered on the on-demand market with a huge discount when compared to an equivalent non-burstable on-demand VM. In this work, we also present the Burst-HADS, an extension of HADS that explores burstable instances to minimize the application's total execution time. Burst-HADS is a multi-objective framework that minimizes the monetary cost and the execution time of the applications. Thus, this thesis presents HADS and Burst-HADS and all studies conducted during their development.

Keywords: Cloud Computing, Scheduling, Spot VMs, Burstable VMs, Checkpoint.

List of Figures

4.1	Execution without hibernation	25
4.2	Execution with hibernation	26
4.3	General architecture of the Frameworks	27
4.4	Frameworks' execution steps	28
4.5	Examples of the <i>env.json</i> and <i>job.json</i> input files	28
5.1	Dump time in S3, EBS and EFS for different sizes of memory footprint	32
5.2	Dump time of concurrent checkpoints in S3 and EFS storage services	34
5.3	Overheads of launching a spot VM and recording checkpoints	35
5.4	Recovery Procedure Times	36
5.5	Monetary Costs of Storage and VM hiring with Services S3, EBS and EFS	38
6.1	D_{spot} definition	41
6.2	Execution where v_j hibernates at p , does not resume, and its tasks are migrated at mtt	49
6.3	A scheduling with two logical Allocation Cycles (AC)	49
6.4	Diagram with the events and actions handle by the Dynamic Scheduling Module	51
6.5	Spot vm_j state diagram	52
6.6	Evaluating the migration of task 6 to a spot VM	55
6.7	Migration of task 6 to a new on-demand VM	56
6.8	Example of tasks in an on-demand VM that can be stolen by the work-stealing procedure	60
7.1	Average duration of hibernation in different scenarios	66

7.2	Percentage distribution of the procedures used by HADS during the execution of jobs J60 and J80	68
7.3	Percentage distribution of the procedures used by HADS during the execution of jobs J100 and ED200	69
7.4	Impact of variation of k_h in the execution costs of job ED200	70
7.5	Work-stealing distribution of Job ED200	71
7.6	Tasks progress of spot VM c4.large during the execution of Job ED200 in scenario sc_2	72
7.7	Tasks progress of spot VM c3.large during the execution of Job ED200 in scenario sc_2	72
7.8	Dump time variation	73
9.1	Monetary cost of a practical execution of Burst-HADS without hibernations and on-demand only strategy	96

List of Tables

3.1	Related Literature of Scheduling Using Burstable and/or Spot and Regular On-demand Instances	23
4.1	Variables and parameters of the problem.	26
5.1	Monetary Costs of Services S3, EBS and EFS in a Long-running Application	37
6.1	Functions and Procedures called by Primary Scheduling Module Algorithms	46
6.2	Variables and parameters used on Primary Scheduling Module Algorithms	47
6.3	Variables and parameters used by the Dynamic Scheduling Module algorithms	61
6.4	Functions and Procedures called by the Dynamic Scheduling Module algorithms	61
7.1	VMs attributes	63
7.2	Jobs characteristics	63
7.3	Different execution scenarios generated by varying parameters λ_h and λ_r .	65
7.4	Baseline executions	66
7.5	Execution of HADS in scenarios sc_1 to sc_7 . The table shows the probabilistic mass function of the hibernation (λ_h) and the resume events (λ_r) for each scenario, the average number of hibernations, the number of used on-demand VMs, the average makespan, and the average monetary cost . .	68
7.6	Average number of checkpoints, migrations, recoveries, and CPU save time from checkpoint in of job ED200 in the seven scenarios	73
8.1	Notation and Variables used in the Mathematical Formulation.	77
9.1	VMs attributes	88
9.2	Different execution scenarios generated by varying parameters λ_h and λ_r .	88

9.3	Results of the ILS-based Primary Scheduling Heuristic and the Exact Approach	89
9.4	Results of the ILS-based Primary Scheduler, MinMin, MaxMin and Greedy Heuristics.	90
9.5	Cost and Makespan of Burst-HADS and HADS, without hibernation; and ILS On-demand only.	91
9.6	Comparison between Burst-HADS and HADS in terms of monetary cost and makespan in scenarios s_1 to s_5	92
9.7	Attributes of Burst-HADS' input set VMs	95
9.8	Different execution scenarios generated by varying the parameters λ_h and λ_r	96
9.9	Comparison between Burst-HADS and On-demand strategy in terms of monetary cost and makespan in scenarios c_1 , c_2 and c_3	97

Acronyms and Abbreviations

AC	: Allocation Cycle
AWS	: Amazon Web Services
BoT	: Bag-of-Task
BP	: Base Pairs
Burst-HADS	: Burst Hibernation Aware Dynamic Scheduler
CRIU	: Checkpoint Restore In Userspace tool
EBS	: EC2 Block Storage
EC2	: Amazon Elastic Compute Cloud
EFS	: Elastic File System
HADS	: Hibernation-Aware Dynamic Scheduler
IaaS	: Infrastructure-as-a-Service
ILS	: Iterated Local Search
PaaS	: Platform as a Service
S3	: Simple Storage Service
SaaS	: Software as a Service
SDK	: Software Development Kit
vCPU	: Virtual CPU
VM	: Virtual Machine

Contents

1	Introduction	1
1.1	Objective	4
1.2	Contributions	5
1.3	Thesis Outline	6
2	Background	7
2.1	Cloud Computing	7
2.2	Amazon Web Services	9
2.2.1	Markets of the Amazon Elastic Compute Cloud	10
2.2.2	Storage Services	11
2.3	The Scheduling Problem on Clouds	13
3	Related Work	16
3.1	Bag-of-Tasks Scheduling on the Cloud	16
3.2	Scheduling using Spot and On-demand Instances	18
3.3	Burstable Instances Related Literature	21
4	Proposed Models and Framework Architecture	24
4.1	System and Application Models	24
4.2	Architecture of HADS and Burst-HADS Frameworks	26
5	Evaluating AWS Storage services for Checkpointing and Recovering	31
5.1	Dump Time Evaluation	32

5.1.1	Dump Time evaluation of Concurrent Checkpoints	33
5.2	Overall Overhead Analysis	34
5.3	Monetary Cost Estimation	37
6	Hibernation Aware Dynamic Scheduler	40
6.1	Primary Scheduling Module	40
6.1.1	Estimation of D_{spot}	40
6.1.2	Checkpoint Intervals	42
6.1.3	Primary Scheduling Heuristic Algorithm	43
6.2	Dynamic Scheduling Module	47
6.2.1	Preliminary Concepts	47
6.2.2	VM states and Allocation Cycle concept	48
6.2.3	Migration Time Limit (mtt)	49
6.2.4	Event Handler	51
6.2.5	Migration Procedure	54
6.2.6	Work-Stealing Procedure	59
7	Experimental Results of HADS	62
7.1	Experimental Environment	62
7.1.1	Emulation of the Hibernation and Resume Events	64
7.1.2	Parameters Setting and Generated Scenarios	64
7.2	Baseline Executions	66
7.3	Performance Results	67
7.3.1	Impact of hibernation and resuming	70
7.3.2	Built-in Functions Evaluation	71
8	Burst Hibernation Aware Dynamic Scheduler	74
8.1	Burst Primary Scheduling Module	74

8.1.1	Mathematical Formulation	74
8.1.2	Iterated Local Search Heuristic	77
8.2	Burst Dynamic Scheduling Module	81
8.2.1	Migration Procedure	81
8.2.2	Work-Stealing Procedure	84
9	Burst-HADS Experimental Results	87
9.1	Experimental Environment	87
9.2	Evaluation of the ILS Primary Task Scheduling	88
9.3	Baseline executions	90
9.4	Performance Results	91
9.5	Case study: A Sequence Alignment Problem	93
10	Conclusions and Future Work	98
10.1	Concluding Remarks	98
10.2	Future Work	100
	Bibliography	102
	Appendix A - Published Papers	111

Chapter 1

Introduction

In the past few years, cloud computing has emerged as an attractive option to run different applications due to several advantages over other platforms, such as clusters and grids. Infrastructure-as-a-Service (IaaS) existing cloud platforms (e.g., Amazon Web Services, Microsoft Azure, Google Cloud, etc.) enable users to dynamically acquire resources, usually as virtual machines (VMs), according to their application requirements, without upfront capital investments, and in a pay-per-use model where users only pay for which was actually used. Those platforms offer different classes of VMs with distinct guarantees in terms of availability and volatility, provisioning the same resource through multiple pricing models and markets. For instance, in Amazon Elastic Compute Cloud (EC2), there are three main markets: i) *reserved* market, where the user pays an upfront price, guaranteeing long-term availability; ii) *on-demand* market which is allocated for specific periods of time, and incurs a fixed cost per unit time of use, ensuring the availability of the instance during this period; iii) *spot* market in which unused resources are available up to 90% discount when compared to the on-demand model.

In the three markets, there is a wide range of VM types that suit different user requirements. According to Amazon Web Service (AWS), “Instance types comprise varying combinations of virtual CPUs (vCPUs), memory, storage, and networking capacity and give you the flexibility to choose the appropriate mix of resources for your applications. Each instance type includes one or more instance sizes, allowing you to scale your resources to the requirements of your target workload” [16]. Instance types are grouped into families based on their use case. For example, the compute-optimized instances (c3, c4, and c5) offered in EC2 are ideal for compute-bound applications that require high-performance processors.

In the spot market, the availability of its VMs fluctuates according to the cloud’s

current demand. If there are not enough resources to meet clients' requests, the cloud provider can interrupt a spot VM (temporarily or definitively). Despite the risk of unavailability, the main advantage of spot VMs is that their costs are much lower than on-demand VMs since the user requests unused instances at steep discounts. An interrupted spot VM instance can either terminate or hibernate. If the VM will be terminated, the cloud provider warns the user two minutes before its interruption. On the other hand, hibernated VM instances are frozen immediately after notifying the user. In this case, EC2 saves the VM instance memory and context in the root of EC2 Block Storage (EBS) volume, and during the VM's interruption period, the user is only charged for the EBS storage use. EC2 resumes the hibernated spot instance, reloading the saved memory and context, only when there is enough available resource whose price is lower than the maximum one, which the user agreed to be charged.

Besides the markets, all leading cloud providers introduced in the last years the concept of burstable VMs. Those VMs can sprint their performance during a limited period of time to cope with sudden workload variations. By operating on a CPU credit regime that controls the processing power offered to users, burstable VM instances can use 100% of the VM's processing power or only a fraction of its power depending on such credits. Burstable on-demand instances have two main advantages: i) they are offered with an up to 20% discount compared to non-burstable on-demand instances with equivalent computational resources, and ii) contrarily to spot VMs, they are not prone to revocation. On the other hand, to obtain monetary advantages of burstable instances, the user has to control their respective CPU credit usage by monitoring their baseline performance and defining bursting periods.

In this thesis, we are particularly interested in Bag-of-Task (BoT) applications executing in cloud environments. This type of application is composed of independent tasks which can be executed in any order and in parallel. In addition, we consider that the BoT applications may require deadline-bounds where the correctness of the computation also depends on the time for executing all tasks. It is worth pointing out that, although simple, the BoT approach is used by several well-known applications such as parameter sweep, chromosome mapping, Monte Carlo simulation, and computer imaging applications. Moreover, the task scheduling to a heterogeneous environment is a well-known NP-hard problem which makes that an even more challenging problem [34, 23].

Thus, we propose the Hibernation-Aware Dynamic Scheduler (HADS). A framework that explores hibernate-prone spot instances and on-demand instances to minimize the

applications' monetary cost. To this end, HADS tries to execute the application tasks in spot VMs as much as possible. However, it must also ensure that the application deadline constraints will be satisfied even if allocated spot VMs hibernate multiple times, otherwise, a temporal failure will take place. The latter happens whenever one or more spot VMs hibernate, not resuming in time to satisfy application's deadline. The framework is event-driven and was built in a modular way with two main scheduling modules: i) the **Primary Scheduling Module** that defines an initial scheduling map of tasks to VMs, and ii) a **Dynamic Scheduling Module** responsible for task migration or task rollback recovering to/in idle VMs.

HADS' first version was presented in [76]. In this version, we used hibernation-prone spot instances to minimize only the monetary cost of BoT applications execution, respecting their deadline constraints. To meet such a deadline, even in the presence of multiple hibernations, new on-demand VMs, not allocated in the initial mapping would be dynamically launched. In this case, tasks of the hibernated spot instances and those not executed yet would be migrated to on-demand instances. Although the strategy significantly reduced the monetary costs, always respecting the application deadline, the application's total execution time could considerably increase if VM spots hibernate. To tackle such a problem, we investigate how burstable instances could be used to reduce the impact on the application's execution time and the corresponding monetary costs. Thus, in [77], we proposed Burst-HADS, an extension of HADS that manages the execution of BoT applications with deadline constraints by scheduling them to spot and burstable on-demand VMs, aiming to minimize both the monetary costs and the BoT total execution time (makespan).

Although both frameworks were originally designed and evaluated using AWS, it is important to point out that both of them are easily adapted to other cloud services. In fact, except for the hibernation feature, which is currently supported only by AWS, the frameworks could be extended to support any cloud provider that offered a Software Development Kit (SDK) to request and manage the cloud resources and services. Thus, if in the future, other cloud providers like Google Cloud or Microsoft Azure implement a feature similar to Amazon's spot hibernation, the frameworks can be adopted in these environments too.

In this work, we present both versions of the framework and all the steps followed during their development. This work was conducted in collaboration with the LIP6 lab-

oratory from Sorbonne Université as part of the ReMatCH project¹. That collaboration started in 2019 and yielded a total of four papers: [74, 75, 78, 77], a 6-month sandwich doctorate in the LiP6 laboratory, and several events participation. Moreover, in 2021 we start a project called BioCloud, in partnership with CNPq and AWS². From this project, we present a case study published in [79], where 22,600 SARS-CoV-2 sequences were compared with the MASA-OpenMP tool [69] in Amazon EC2 using Burst-HADS, showing that clouds can play a fundamental role in ensuring the efficiency of the execution of such applications with reduced costs.

Thus, in this thesis, we present a compilation of all those publications.

1.1 Objective

In this work, the main objective is to explore hibernation prone spot VMs and Burstable VMs to reduce the monetary cost and execution time of the BoT applications subject to a deadline defined by the user.

Therefore, we propose HADS, a modular and lightweight framework. It includes a series of built-in functions such as load balance, checkpoint, and recovery procedures. Unlike other frameworks that cope with the termination/revocation of spot VMs, HADS explores the hibernation feature of spot VMs to minimize the monetary cost of the execution.

Moreover, we also propose Burst-HADS, an extension of HADS that also explores hibernation-prone spot VMs to minimize the monetary cost, but it also uses the burst capacity of the burstable on-demand instances to minimize the application's execution time. In fact, unlike HADS, Burst-HADS is a multi-objective framework and tries to minimize both the monetary cost and the execution time at the same time.

In addition, we present the formulation of the primary task scheduling problem, a multi-objective problem that aims to minimize, at the same time, (i) the monetary cost and (ii) the execution time, makespan, of the initial scheduling plan defined by Burst-HADS' Primary Scheduling Module. As we will show in Chapter 9, the execution time to solve the exact model is prohibitive even to BoT applications with few tasks. Thus, to find good solutions to the primary scheduling problem in an acceptable time, we propose an Iterated Local Search (ILS) based approach presented in Chapter 8. The ILS heuristic was

¹More information about the project can be found in <http://cloud.ic.uff.br/index.php/pt/capes-print/>

²More information about the project can be found in <http://cloud.ic.uff.br/index.php/project-cnpq-aws/>

validated by comparing it to the optimal solutions given by the mathematical formulation and to baseline algorithms.

Besides the scheduling frameworks and the mathematical formulation, we also show an extensive evaluation using several scenarios of spot VMs hibernation and resumes. Those tests were performed in a real cloud environment, using synthetic and real benchmark applications. The results showed that HADS and Bust-HADS can be very advantageous, guaranteeing the application's deadline, even in the presence of hibernations, and still optimizing the monetary cost.

1.2 Contributions

The main contributions of this work are the following:

- I The proposal of the HADS framework that explores hibernation-prone spot VMs to minimize the monetary cost of the execution;
- II The design of the Primary Scheduling Module that defines the initial scheduling strategy;
- III The design of the Dynamic Scheduling Module that reacts to events that may cause deadline violation;
- IV The proposal of Burst-HADS, a multi-objective framework, that explores hibernation-prone spot VMs and burstable VMs to minimize the monetary cost and the execution time;
- V The design of the Burst Primary Scheduling Module that uses an ILS-based heuristic to define the initial scheduling strategy;
- VI The design of the Burst Dynamic Scheduling Module that explores the burst capacity of the VMs to minimize the execution time if spot VMs hibernates; and
- VII Practical evaluations of the proposed scheduling frameworks in a real cloud provider environment.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces some background concepts. Chapter 3 presents related works and discusses our proposal's main contributions compared with them. Chapter 4 presents the system and application model considered in this thesis and the frameworks' general architecture. Chapter 5 discuss a study and evaluation of AWS storage services in the context of checkpointing and recovering procedures. In Chapter 6, we described HADS framework, while Chapter 7 presents evaluation results related to HADS. Next, in Chapter 8, we present the Burst-HADS, and its evaluation results are showed in Chapter 9. Finally, Chapter 10 concludes the thesis and introduces some future directions.

Chapter 2

Background

In this Chapter, we present some background concepts related to the work presented in this thesis. Firstly, in Section 2.1, we present some aspects of cloud computing. Next, Section 2.2 presents AWS, the cloud provider used in this work, and that introduces the hibernation feature. Finally, in Section 2.3, we present the scheduling problem in cloud environments.

2.1 Cloud Computing

Over the past years, different definitions of cloud computing have been presented [57, 27, 5, 86, 26]. These definitions vary from the technical aspects, which addresses concepts related to the architecture and the physical environment, to the aspects related to usability, whose focus is on the services provided and the charging models [46].

From the service perspective, cloud computing environments are a set of virtual servers working interconnected over the internet that can scale dynamically and be terminated at any time by the user [43]. These environments are similar to clusters in some technical aspects. However, differently from the latter, the cloud offers computational resources through virtualization, and the users pay only for what they actually used (pay-per-use) [46]. Cloud computing environments can be classified in the following models [57]:

- **Public cloud:** An organization or company owner of large data centers offer computational resources to the general public. Examples of public clouds are AWS, Google Cloud, and Microsoft Azure;
- **Private cloud:** an organization uses which owns infrastructure to deploy a cloud

solution. In this case, this organization is the only one responsible for managing and use the resources; and

- Hybrid cloud: it is a combination of public and private clouds. Normally, the company uses its private cloud resources for the most important or critical tasks, adding additional resources from the public cloud when necessary, for example, to handle workload peaks.

Virtualization is one of the main technologies behind the success of the clouds. It can be defined as a process of sharing computational resources (such as CPU, storage, and network) that isolate the physical hardware. According to [41], the virtualization process reduces the inefficiency of the allocating and distribution of computational resources. The most common way of virtualization in cloud computing is the use of virtual machines. VMs create hardware and software environments configured completely independently of the physical resource, allowing the simultaneous execution of independent VMs on top of the same physical hardware [43].

Compared with other distributed platforms, such as grids and clusters, the cloud computing services present several technical and economic advantages, as they combine virtualization and scalability in models of service economically viable for both the client and the provider [46]. Some of the main advantages of using the clouds are [41]: availability, monetary cost savings, reliability, and service integration. Moreover, cloud providers claim to present an unlimited pool of resources to the users. According to Hashem *et al.* [41], besides the minimization of the monetary cost related to automation and data processing, in the cloud, the reduction on the costs starts since the acquisition and maintenance until the management of the infrastructure.

Elasticity is one of the key advantages of cloud computing. In a scenario in which the allocated resources are not enough to meet the application's demand, either due to load variations or incorrect estimates, cloud environments allow the users to change the current computational power by allocating or removing resources. Thus, the use of elasticity eliminates the need to previously request a greater amount of computational power than the application really needs (over-provisioning), to meet possible future load increments.

The services offered by the cloud are divided into three categories [41]:

- *Platform as a Service* (PaaS), where different resources (operating systems, libraries, compilers, etc.) operates together to provide a platform for the end-user. AWS

Elastic Beanstalk [70], Google App Engine [33], and Heroku [42] are some examples of PaaS.

- *Software as a Service* (SaaS) consists of applications executing directly on a cloud infrastructure and offered as a service to the end-user. Examples of this model are Google Docs [38], Gmail [37] and Overleaf [62].
- *Infrastructure as a Service* (IaaS) refers to the acquisition of virtualized hardware equipment. In this model, the customer has full control of the configuration and installation of the software. Amazon EC2 [9], Google Cloud Platform [24], and Microsoft Azure [18] are examples of IaaS.

Usually, the use of cloud services for executing distributed applications is based on the IaaS model and is done by allocating a virtual cluster, which can be composed of VMs of the same type (homogeneous environment) or VMs with different configurations (heterogeneous environment). Thus, before the execution, it is necessary to define and request the necessary VMs and plan the application's execution.

2.2 Amazon Web Services

There are currently several public cloud providers, such as AWS [12], Microsoft Azure [18] and Google Cloud [24]. Among those companies, AWS is undoubtedly the market leader. According to [60], AWS is responsible for almost half the world's public cloud infrastructure market, reporting in 2020 a profit of \$13.5 billion. AWS provides a series of services and features, from infrastructure technologies like compute, storage, and databases to emerging technologies, such as machine learning and artificial intelligence. Amazon EC2 [9] is one of these services. EC2 allows users to contract resources packaged as VMs, organized in the following key features, such as the number of vCPUs, memory capacity, network bandwidth, and usage purpose [16].

In EC2, a price is established for each VM according to its resources. For example, while a general-purpose t3.large VM (2 vCPUs and 8 GB of memory) can cost up to 0.0832\$ for one hour of use; a CPU optimized c3.9xlarge VM (36 vCPUs and 72 GB of memory) costs up to 1.53\$ for the same time period. Currently, EC2 offers more than 30 VM types. Yet, users have limits on how many instances they can contract at the same time.

Since October 2017, EC2 uses a per-second billing model for Linux VMs, unlike the previous per-hour billing [21], providing an opportunity to pay for what was actually used. Other providers also use similar billing models. For example, in Google Compute Engine¹, when a new VM is initiated, users pay for its first minute (even if it was used by only 30 seconds). After this minute, instances are charged in one-second increments.

2.2.1 Markets of the Amazon Elastic Compute Cloud

In Amazon EC2, the same VM type can be hired in three different markets: on-demand, spot, and reserved. In the on-demand market, the users can request resources at any time and at a fixed price per time unit; the on-demand VMs have their availability and reliability ensured by the provider. In other words, the user has the guarantee that he/she will be able to request resources at any time, and it will be active during the time the user needs. However, the on-demand market has the most expensive resources.

In the reserved market, users need to reserve a set of VMs for a specific and, normally, long period of time. Thus, the environment's total monetary cost is known beforehand, and the provider also guarantees the total availability of the resources. That kind of market is suitable for a user who knows their application's demand and requirements for a long period of time (one or more years). According to AWS, the price reduction of the reserved market in relation to the on-demand one can be up to 72%².

EC2's spot VMs are unused resources offered by the provider with a huge discount (according to Amazon, the discount can be up to 90% when compared to on-demand prices). However, spot VMs can be revoked, i.e., terminated by Amazon whenever the provider needs its resources. Furthermore, in November 2017, EC2 introduced the spot VM hibernation feature [7]. Now, instead of terminated, VMs can be hibernated by the provider [7]. When a VM hibernates, its memory and context are saved, and therefore, at its resumption, the respective context is restored. The tasks interrupted can be restarted from the hibernation point. Yet, while a VM is hibernated, the user is not charged for that VM.

Another interesting feature offered in EC2 on-demand market is the burstable capacity of some VMs types. Those instances can sprint their performance (burst mode) during a limited period of time following the user application demands. The burst capacity operates according to a CPU credit regime that controls the processing power offered to

¹<https://cloud.google.com/compute/vm-instance-pricing>

²<https://aws.amazon.com/ec2/pricing/reserved-instances>

users. Basically, if the instance has CPU credits, the user can use 100% of the VM's processing power (burst mode). Otherwise, only a fraction of that power is available (baseline mode).

Burstable instances accumulate CPU credits per hour, whose amount depends on the instance type. If a burstable instance uses fewer CPU resources than required for baseline performance (for example, when it is idle), the unspent CPU credits are accrued in the CPU credit balance of the instance. If a burstable VM needs to burst above the baseline performance level, it spends the accrued credits. The more credits that a burstable performance instance has accrued, the longer it can burst beyond its baseline when higher performance is required. In EC2, burstable VMs are the ones of type family t2 and t3 [14].

2.2.2 Storage Services

AWS also offers several storage services, as can be seen in their portfolio [15]. Each service is optimized for different storage needs. In this work, we evaluated and used three popular, simple, and cheap general-purpose storage services offered by the provider: (i) the Amazon Simple Storage Service (S3) [11], (ii) the Amazon Elastic Block Store (EBS) [8], and (iii) the Amazon Elastic File System (EFS) [10].

Amazon S3 is probably the most known storage service used in AWS. According to the provider, this service can be utilized to store and recover any amount of data and was developed to answer a minimum quantity of characteristics focused on simplicity and system robustness. S3 provides storage to a wide range of object sizes, from 0 Bytes to 5TB, kept in a two-level organization [11]. In the superior level, there are the buckets, structures that are similar to folders that have a unique global name. As reported by [63], the buckets can be used for different purposes such as allowing data organization by the users, identifying the users to be billed accordingly, data transferring, and as an aggregation to audition reports. Each user of S3 can create up to 100 buckets associated with one unique account, and an unlimited number of objects can be stored in a bucket.

In the inferior level, there are the objects. They contain the data stored by the user and some metadata. The S3 objects' metadata is a key-value pair, in which each data stocked in a bucket is represented by a name and a unique key [17]. The user can create, change and read the objects within a bucket. However, renaming and moving them to another bucket requires the download from the original bucket to a local system.

The charges in S3 are based on three factors: the size of stored objects, storage time, and S3 class used. Each S3 class is specialized for each type of memory access, such as random access, dynamic access, or less frequent one. This work utilized the S3 Standard class, which has low latency, high throughput and is optimized for general access usage. In the *us-east-1* region, localized in North Virginia, checked in February of 2021, the price of each GB per month of storage in S3 is US\$0.023, and for every 1000 requests of PUT, COPY, POST, or LIST type, it is charged US\$0.005. Besides, the data transfer from S3 to the Internet is charged when the user exceeds 1 GB of data within one month.

EBS is another option of storage inside EC2. In this service, users can create storage volumes that are attached to a directory inside the VM [68]. The capacity of these volumes can vary from 1 GB to 16 TB and is defined by the user in the solicitation moment [8]. EBS volumes are persistent and can be kept even without any VM associated with them. However, the disk can only be mounted in one VM at a time. Thus, access to the disk is limited to this single VM. The two types of disk volumes that can be created are the Solid State Drives (SSDs), disks used for applications that need low latency, and the Hard Disk Drives (HDDs), mostly used for applications that need higher throughput.

In this thesis, we used *gp2* type SSDs as they have a latency of fewer than 10 milliseconds [8]. These SSDs also offer throughput up to 250MB/s. In terms of values, checked in February of 2021, EBS charges US\$0.10 per GB of the allocated disk, unlike S3, which charges by the size of each stored object. Moreover, the read and write operations in EBS are already included in the price.

The EFS is the storage service that provides a simple and scalable file system. This file system is also manageable to the cloud and local storage usage [10]. During its usage, EFS itself increases and decreases the allocated size of the file system automatically as files are inserted and deleted in the system.

The EFS files are stored in many availability zones (AZs) of the same region, and they can be accessed in parallel by as many VMs as the user want. However, there is a limit of 35000 accesses to the EFS files per second, and each file has a size limit of 52.67 TB. The EFS compatible file system is the Network File System (NFS) version 4 (NFSv4.0 or NFSv4.1). Regarding costs, the EFS charges a bigger fee to files that are more frequently accessed by the VMs and a smaller fee to files that are infrequently accessed, as they stay in a cheaper EFS class. The management between both classes can be done automatically by EFS if the user enables the Lifecycle Management in the creation of the EFS. So, in the *us-east-1* region, localized in North Virginia, the fee charged in the most frequently

accessed files is \$0.30 per GB per month, and the fee charged for the infrequent ones is \$0.025 per GB per month, checked in April of 2020.

2.3 The Scheduling Problem on Clouds

Yu *et al.* [91] define scheduling as the process of mapping tasks and manage the execution of the entire applications in a computational environment. Therefore, the scheduler's main function is to define where and when the tasks should be executed. Usually, this decision is bounded by restrictions imposed in the application or in the environment, such as a deadline or memory limits. According to Topcuoglu *et al.* [83], in distributed systems, an efficient scheduling approach is a key factor to achieving high computational performance.

In formal terms, the scheduler goal is to allocate a set of tasks $B = \{t_1, t_2, \dots, t_m\}$ to a set of machines $M = \{vm_1, vm_2, \dots, vm_n\}$. During that process, the scheduler is guided by an objective function. Moreover, the scheduler is subject to restrictions that, if not satisfied, make the solution unfeasible [84]. The most common objectives for scheduling applications are: minimize the total execution time (makespan), maximize the load balance, and minimize the failure rate. Besides, in cloud environments, the execution's monetary cost is one of the main users' concerns. In addition to the objectives, the most common restrictions faced by the scheduler are budget limits and deadlines [56].

The problem of scheduling tasks in a distributed system is part of the so-called NP-hard problems [85]. This means that no known algorithm can find the optimal solution in a polynomial time (unless $P = NP$). Consequently, scheduling approaches usually implement algorithms that generate approximate results, called scheduling heuristics [83, 56, 91]. These approaches do not guarantee that the solution found is optimal, i.e., the best possible solution to the problem. However, they can find solutions with acceptable quality and in a viable execution time [47].

Scheduling algorithms can be either static or dynamic [31]. In static algorithms, the entire scheduling plan is defined before the execution. To do that, the algorithm may consider the environment's initial conditions (such as processing capacities, number of available VMs, and bandwidth). Moreover, the heuristic is not able to take into account possible changes that can occur during the application execution, and all information related to the application and the environment, such as execution time, memory capacity, processing power, etc., must be known *a priori*. Therefore, the information's accuracy directly influences the scheduling result since inaccurate information induces the scheduler

to generate low-quality solutions. The most common ways of obtaining tasks and environment information are from historical records of previous executions or benchmark tests. Furthermore, some information can be obtained from estimations using mathematical models or statistical tools.

In the case of dynamic algorithms, the scheduling plan is created or updated during the execution of the application. Usually, in this approach, tasks are allocated in stages according to the computational resources' availability. Thus, the scheduler monitors the execution, including the VMs and tasks' status and, if there are tasks ready to be scheduled and idle VMs, it allocates those tasks following parameters that meet the desired objective. Generally, the dynamic scheduler uses a local approach, where each task's assignment is decided independently, using only the information of the task and the available resource, without considering other tasks in the decision [31, 47]. Moreover, unlike static scheduling, the dynamic scheduler can update from time to time all information related to the application and with the environment, which improves accuracy. Also, any change in the environment can be easily detected and handled by adapting the allocation plans. Although dynamic scheduling is more flexible and requires fewer input parameters, its execution overhead and implementation complexity are usually greater than the static one [31].

Differently from other environments such as grids and clusters, some additional aspects of the cloud environments need to be taken into account by the scheduling strategy. Firstly, since clouds adopted a pay-per-use pricing model, every decision made in those environments results in monetary cost. Thus, although cloud resources are virtually unlimited, a user's budget is not. Hence, the scheduling decisions need to consider a trade-off between the desired performance and the monetary cost.

Secondly, there is a wide variability of instance types offered by the providers. It can be a challenge to the users to define what type of instance is suitable for their application. Those issues can be further complicated when considering other services and aspects of the cloud, such as different markets and features. Finally, in the cloud, the heterogeneity of the physical hardware can result in unexpected performance variation. For instance, a VM running in a brand new physical machine probably will have better performance than a VM of the same type running in an old physical machine [81].

Thus, unlike the advertising of cloud providers who advocate usability as one of the main advantages of cloud environments, when we consider all the necessary variables to be defined to execute a given application efficiently, a cloud can become a complex envi-

ronment where any decision directly impacts the final execution and respective monetary costs.

Chapter 3

Related Work

Elastic environments, such as clouds, where computational resources can be added and removed based on the application's needs, are extremely suitable for applications composed of independent tasks. More recently, with the introduction of new hiring models, as the spot and reserved markets, several works that explore the characteristics of those models have also been proposed [58]. In this Chapter, we discuss some of those works, including the burstable instances related literature.

This Chapter is divided into three sections. Firstly, in Section 3.1, we introduce works that deal with the scheduling of BoT application in cloud environments. Next, in Section 3.2, we present scheduling works that propose solutions that use spot or/and on-demand instances. Finally, in Section 3.3 we present related literature about burstable instances.

3.1 Bag-of-Tasks Scheduling on the Cloud

BoT applications running in cloud environments are widely used not only for scientific applications but also for many commercial applications. In [35], Facebook reports that the jobs running in their own internal data centers are mostly independent tasks. Many works propose then scheduling independent tasks to both on homogeneous and heterogeneous cloud environments [82]. In the former, the performance and pricing of all available VMs are the same. In this case, authors usually consider either reserved VMs [89] or on-demand VMs [81]. For instance, Thai et al. [81] study scheduling of applications on on-demand VMs distributed across different datacenters, focusing on the trade-offs between performance and cost while Yao et al. [89] provide a solution that satisfies job deadlines while minimizing monetary cost. The proposed heuristics use both on-demand

and reserved VMs. Works on heterogeneous cloud consider different types of VMs. For instance, in [80] the authors present a heuristic algorithm for executing a bag-of-tasks application taking into account either budget or deadline constraints. In [82], Thai et al. present an extensive survey and taxonomy of existing research in scheduling of bag-of-task applications on clouds.

Although the spot market has received a lot of attention in the last years, few BoT schedulers exploit the use of spot VMs. Yao *et al.* [89], for example, propose a scheduler that satisfies a deadline, and that minimizes the monetary cost, by using only on-demand and reserved VMs. In [40], an agent-based strategy that uses different heuristics to schedule concurrent BoT applications to on-demand VMs is presented. Farahadaby *et al.* [32] propose FPRAS, a scheduler algorithm for BoT applications in multi-cloud environments whose objective is to minimize the monetary cost of the execution. In [49], Keshanchi *et al.* propose N-GA, a genetic algorithm-based scheduler for heterogeneous distributed environments such as clouds. N-GA is a static scheduler whose objective is to reduce the execution time of applications.

In Huang *et al.* [44], the authors aimed to minimize the total execution time of BoT applications executed in on-demand VMs, by developing a PSO-based scheduler. In [61], the authors present BaTs, a budget-constrained scheduler that uses on-demand VMs to execute BoT applications. In [22], Chakravarthi *et al.* present NBWS, a budget constraint dynamic scheduler, for scheduling workflows in on-demand VMs. Unlike the majority of the related works, NBWS considers CPU performance variation of on-demand VMs and the overhead (delay) of resource acquisition to make scheduling decisions. According to the authors, the simulated results showed that NBWS was able to over-performance baseline schedulers in monetary cost and execution time.

Unlike our approach, all the above works do not consider the use of spot VMs to minimize the monetary cost of the execution and do not apply any technique that guarantees the complete execution of the applications in case of VMs interruptions.

Besides the monetary costs and execution time of applications, energy consumption is also a popular objective of BoT scheduling problems. In [73], for example, Tang *et al.* propose a heuristic that defines where new jobs should be scheduled to reduce the number of active cloud data centers. The authors use a workload prediction approach that combines linear regression with neural network techniques. In [23], a multi-criteria meta-heuristic, whose objective was to minimize the makespan of the application and the energy consumption of the cloud resources, was proposed. In [54], Lu and Sun present an

energy-efficient resource scheduling algorithm, called CSRSA (Clonal Selection Resource Scheduling Algorithm). The algorithm deals with the problem of energy consumption by applying concepts and principles of load balancing techniques. According to the authors, their simulated results show that CSRSA has a close optimal ability to reduce energy consumption on data centers. In [6], the authors have proposed a multi-objective divisible scheduling heuristic whose aim is to minimize both energy consumption and execution time of BoT applications. Similarly to our work, applications are subject to a deadline. However, since the energy efficiency is related to data centers, those works are applied in the provider-side and not on the client-side, as is the case of HADS and Burst-HADS.

3.2 Scheduling using Spot and On-demand Instances

Spot and on-demand instances have received a lot of attention in applications scheduling. Likewise to our proposed framework, several works in the related literature [72, 66, 58, 71, 53, 87, 74, 75] propose, for monetary cost sake, the use, whenever possible, of spot VMs for scheduling applications. On the other hand, as these works were conceived before December 2017, they cope with the termination/revocation of spot VMs instead of their hibernation. The common objective of them is rather a tradeoff between monetary cost, reliability, and execution time. SpotOn [72] is a batch computing service platform that uses the price history of the spot market to select the fault-tolerance mechanism that can mitigate the impact of spot VMs revocations and minimize the expected monetary cost of the job execution. Loo *et al.* [53] proposed a hybrid approach that considers on-demand and spot VMs to execute tasks with different priorities. In their approach, on-demand VMs are used for high priority tasks while spot VMs are reserved for non-priority ones. Spot VMs interruptions are tackled by reserving a certain number of on-demand VMs as spare resources to execute backup tasks. Whenever a spot instance is terminated, the workload is immediately migrated to on-demand VMs.

Pham and Fahringer [66] propose an evolutionary multi-objective scheduling algorithm that minimizes the makespan and the cost of workflow applications running in the cloud. The strategy starts the execution using spot instances. In case of revocation, the spot instance is replaced by a similar on-demand VM keeping the same scheduling plan defined beforehand the execution. In [92], Zhou *et al.* proposed Dyna, a probabilistic scheduling system that minimizes a Workflow-as-a-Service provider's cost while satisfying the performance guarantees of individual workflows predefined by the user. According to the authors, the system uses a series of optimization techniques for monetary cost opti-

mizations, specifically designed for cloud dynamics, and adopt both spot and on-demand instances. The spot instances are adopted to reduce monetary cost, and on-demand instances are used as the last defense to meet deadline constraints.

In [71], Sharma *et al.* proposed SpotCheck, a framework that uses nested VMs within spot VMs to provide the illusion of a platform that offers always-available VMs. In order to cope with spot revocations, the nested VMs are migrated to an on-demand VM whenever a spot revocation occurs. Menache *et al.* [58], proposed an online learning algorithm, which selects spot and on-demand VMs to execute batch jobs that arrive over time. The algorithm adapts the resource allocation by learning from its performance on prior job executions and from the history of spot prices.

AutoBot, proposed in [87], uses both spot and on-demand VMs for scheduling tasks of BoT applications with a user-defined deadline. It applies task migration from spot to on-demand VMs to satisfy time constraints and uses checkpoint strategies for performance sake. AutoBot [87] is the closer work to HADS, because: (i) it uses both spot and on-demand VMs for scheduling tasks of BoT applications with a user-defined deadline; (ii) it applies task migration from spot to on-demand VMs to satisfy constraints; and (iii) it uses checkpoint strategies for performance sake. However, although AutoBot article was published in 2019, the authors still consider bid prices and the variation of the market to ensure reliability and to meet the application deadline. Furthermore, unlike HADS, where migration is a consequence of spot VM hibernations, AutoBot considers a critical point within the application execution when all tasks running in spot VMs should migrate to on-demand ones, even if no spot interruptions has happened. Consequently, AutoBot does not take full advantage of the available allocated spot VMs as HADS does.

All the above works cope with the termination/revocation of spot VMs and do not consider the hibernation feature. Moreover, most of them exploit the historical of spot VM price variation to predict spot VMs' revocations. In fact, the prediction based on spot price variation is a good approach. However, since 2017 that approach can be more challenging in AWS, as the provider has adopted prices more stable to the spot market. Moreover, some techniques explored the spot market's bid price to improve the reliability of the VMs. That approach is no longer support since the provider does not use the bid [64].

Several works from the related literature deal with the checkpointing problem of BoT applications in clouds [72, 36, 6, 87]. Yi *et al.* [90] propose an adaptive checkpoint that takes into account the price variation of the spot VM to predict the spot termination and

decide when a checkpoint shall be taken. On the other hand, SpotOn [72] implements a proactive mechanism, where the number of checkpoints is neither related to the market volatility nor the number of revocations, but on a specified checkpointing interval. In [87], three checkpoint strategies are proposed: i) optimistic checkpoint, where the state of the task is recorded just before the migration to an on-demand VM; ii) grace period checkpoint, where the two minutes between the notification of the interruption of a spot VM and the VM interruption itself are used to take the checkpoint; and iii) sliding checkpoint, where the checkpoint is taken in fixed intervals.

As stated before, since AWS adopted a new price model, prices of VMs in the spot market are quite stable and defined exclusively by the supply and demand of spare capacity and no more by bid prices [64]. Therefore, checkpoint strategies based only on price variations could no be so efficient in EC2's VMs. Another remark is that the just described grace period checkpoint cannot be applied by HADS, because spot VMs are hibernated immediately without the two minutes notification. Therefore, uncoordinated checkpoints, periodically taken in constant intervals, akin to SpotOn checkpoint strategy, seem to be the most suitable technique for tasks running in hibernated-prone spot VMs. Instead of saving task states, SpotCheck [71] provides a checkpoint of the VM's memory state in an external disk by running a background process that continually flushes dirty memory pages to a backup machine. The VM may then resume from the saved memory state in a different machine.

To deal with spot VM revocations, we proposed in [74] a static heuristic that creates pre-defined backup maps before the execution of the job tasks themselves. It was the first attempt to cope with spot VMs hibernation, and results from simulation showed that the hibernation problem is better handled with a dynamic approach. Thus, in [76] we present a dynamic scheduler, denoted HADS that uses both spot and regular (non-burstable) on-demand VMs to execute BoT applications. It aims at minimizing the execution's monetary cost respecting application deadline. Finally, in [77], we present Burst-HADS, a extension of HADS that exploits hibernation-prone spot VMs and also burstable VMs. It also applies new heuristics to reduce not only the monetary cost of the execution but also the execution time of the application, while meeting the deadline defined by the user.

To the best of our knowledge, the hibernation mechanism of the spot VMs is only discussed in our previous works [74, 76] and in Fabra *et al.* [30]. In the latter, the authors consider a scenario where hibernation-prone spot VMs can be used and then they show that deadline constraints add complexity to the problem of resource provisioning.

However, no practical solution is presented neither discussed. Moreover, that work does not concern the task scheduling problem, but a resource provisioning one.

3.3 Burstable Instances Related Literature

Since AWS introduces the concept of burstable VM, many works exploring its features have been proposed. Many of them focus on evaluating the burstable approach and the improvement in computational performance that it can induce. In [51], Leitner and Scheuner presented a first empirical and analytical study about the second generation of AWS burstable instances (T2 family). They specifically considered T2.micro, T2.small, and T2.medium instances. Their article aimed at answering if, in terms of monetary cost and performance, these instance types are more efficient than other ones. The presented results show that compared to general-purpose and computed-optimized instances (2015 generation), the evaluated T2 instances provide a higher CPU performance-cost ratio as long as the average utilization of instances is below 40%.

To figure out the CPU usage limits on-the-fly, considering the dynamic variation of the workloads, Ali *et al.* proposed in [2] an autonomic framework that combines light-weight profiling and an analytical model. The objective was to maximize the amount of work done using the burstable capacity of T2 VM instances. The authors state that the framework extends the CPU credits depletion period. Similarly to Leitner and Scheuner's work [51], their results also confirmed the benefits from active CPU usage control when burstable instances are exploited. However, they do not discuss the impact on the total execution time when such an approach is applied. Jiang *et al.* [45] analytically modelled the performance of burstable VMs, considering their respective configuration, such as CPU, memory, and CPU credits parameters. They also showed that providers could maximize their total revenue by finding the optimal prices for burstable instances. Although their work, contrarily to ours, does not focus on application performance, its contribution is interesting since it states that providers can offer burstable instances for low prices without losing revenue while meeting QoS parameters.

Some scheduling and scaling works also take advantage of burstable instance features. In [19], for example, Baarzi *et al.* proposed an autoscale tool denoted BurScale, which uses burstable instances, together with on-demand instances, to handle transient queuing which arises due to traffic variability. They also presented how burstable instances can mask VM startup/warmup costs when autoscaling, to handle flash crowds, takes place.

Using two distinct workloads, a stateless web server cluster and a stateful Memcached caching cluster, the authors showed that a careful combination of burstable and regular instances ensure similar performance for applications as traditional autoscaling systems while reducing up to 50% of the monetary cost.

In [88], Wang *et al.* combined on-demand, spot, and burstable instances proposing an in-memory distributed storage solution. Burstable instances were used as a backup to overcome performance degradation resulting from spot instance revocations. According to the authors, those instances' burst capacity makes them ideal candidates for such a backup. Performance results show that the backup that uses burstable instances presents a latency, which is 25% lower than the latency of backup based on regular instances, inducing, therefore, significant monetary cost saving.

Table 3.1 summarizes the main characteristics of the related approaches presented in this Chapter. The following features are highlighted in the table: Spot, On-demand and Burstable, which indicates if the solution considers spot and/or on-demand markets and also burstable VMs; hibernate/resume, which shows if hibernation-prone VMs are used; the type of the scheduled application and the used fault tolerance technique; the objectives and constraints of the scheduling algorithm; and how the proposed scheduler was evaluated.

As we can see in Table 3.1, on-demand VMs are used in more than 95% of the related literature, while spot VMs comprise 50% and Burstable VMs only 12.50%. Moreover, more than 83% of the works considered BoT applications. In terms of fault tolerance, the migration procedure is used in more than 54% of the related works, while checkpoint and replication are used in 29.17% and 4.17%, respectively. In the table, we also see that the minimization of the monetary cost is the most common objective found in the related literature (more than 66% of the related works). Finally, the evaluation using simulation was adopted by more than 79% of the works, while the practical evaluation was used in only 33.33% of the approaches.

Table 3.1: Related Literature of Scheduling Using Burstable and/or Spot and Regular On-demand Instances

Article	Spot	Burstable	On-demand	hibernate/resume	Applications	Fault Tolerance	Objective (minimize)	Constraint	Evaluation Approach
Oprescu and Kielmann [61] (2010)	No	No	Yes	No	BoT	-	Execution Time	Budget	Simulation
Yi <i>et al.</i> [90] (2011)	Yes	No	No	No	BoT	Checkpoint and Migration	Monetary Cost	-	Simulation
Farahadaby <i>et al.</i> [32] (2012)	No	No	Yes	No	BoT	-	Monetary Cost	-	Simulation
Lu <i>et al.</i> [53] (2013)	Yes	No	Yes	No	BoT	Migration	Monetary Cost	-	Simulation
Gutierrez-Garcia <i>et al.</i> [40] (2013)	No	No	Yes	No	BoT	-	Monetary Cost	-	Simulation
Aupy <i>et al.</i> [6] (2013)	No	No	Yes	No	BoT	Checkpoint	Energy Consumption and Execution Time	Deadline	Simulation
Menache <i>et al.</i> [58] (2014)	Yes	No	Yes	No	BoT	Migration	Monetary Cost	Deadline	Simulation
Yao <i>et al.</i> [89] (2014)	No	No	Yes	No	BoT	-	Monetary Cost	Deadline	Simulation
Zhou <i>et al.</i> [92] (2015)	Yes	No	Yes	No	Workflows	Checkpoint and Migration	Monetary Cost	Deadline	Simulation and Prototype on EC2
Subramanya <i>et al.</i> [72] (2015)	Yes	No	Yes	No	BoT	Checkpoint, Migration and Replication	Monetary Cost	-	Simulation and Prototype on EC2
Sharma <i>et al.</i> [71] (2015)	Yes	No	Yes	No	BoT	Migration	Monetary Cost	-	Prototype on EC2
Wang <i>et al.</i> [88] (2017)	Yes	Yes	Yes	No	Memcached Application	Migration	Monetary Cost	-	Prototype on EC2
Keshanchi <i>et al.</i> [49] (2017)	No	No	Yes	No	BoT	-	Execution Time	-	Simulation
Tang <i>et al.</i> [73] (2018)	No	No	Yes	No	BoT	-	Energy Consumption	-	Simulation
Huang <i>et al.</i> [44] (2019)	No	No	Yes	No	BoT	-	Execution Time	-	Simulation
Lu and Sun [54] (2019)	No	No	Yes	No	BoT	-	Energy Consumption	-	Simulation
Teylo <i>et al.</i> [74] (2019)	Yes	No	Yes	Yes	BoT	Migration	Monetary Cost	Deadline	Simulation
Pham and Fahringer [66] (2019)	Yes	No	Yes	No	Workflows	Migration	Monetary Cost and Makespan	-	Simulation
Varshney and Simmhan [87] (2019)	Yes	No	Yes	No	BoT	Checkpoint and Migration	Monetary Cost	Deadline	Simulation and Prototype on EC2
Baarzi <i>et al.</i> [19] (2019)	No	Yes	Yes	No	Web Server and Memcached Application	Migration	Monetary Cost	-	Prototype on EC2
Teylo <i>et al.</i> [76] (2020)	Yes	No	Yes	Yes	BoT	Migration and Checkpoint	Monetary Cost	Deadline	Prototype on EC2
Teylo <i>et al.</i> [77] (2020)	Yes	Yes	Yes	Yes	BoT	Migration and checkpoint	Monetary Cost	Deadline	Prototype on EC2
Chakravarthi <i>et al.</i> [23] (2020)	No	No	Yes	No	BoT	-	Execution Time	Budget	Simulation
Chhabra <i>et al.</i> [23] (2020)	No	No	Yes	No	BoT	-	Energy Consumption and Execution Time	-	Simulation

Chapter 4

Proposed Models and Framework Architecture

In this Chapter, we present the system and application models considered in this thesis and the general architecture adopted by HADS and Burst-HADS. Thus, this Chapter is divided in two sections. Section 4.1 presents the System and the application models, while Section 4.2 introduces the frameworks' architecture.

4.1 System and Application Models

As stated in Chapter 2, cloud computing providers offer computational resources packaged as VMs in different markets. Thus, let $M = M^s \cup M^o \cup M^b$ be the set of VMs that a user can deploy to execute her/his BoT application, where M^s is the set of spot instances, M^o is the set of regular on-demand VMs, and M^b is the set of burstable on-demand instances. M depends on the type and market of the VMs that can be deployed during the execution and must respect the resource restrictions imposed by cloud providers. For instance, in Amazon EC2, if the user decides to use only on-demand VMs of type *c5.xlarge* and spot VMs of type *c5.2xlarge*, M must be composed exclusively by those VMs. Moreover, since, by default, Amazon does not allow more than five VM instances with similar type and market running at the same time, in our example, M^o and M^s would be composed by a maximum of five VMs *c5.xlarge* and five *c5.2xlarge*, VMs. Besides, M^b would be an empty set, since *c5* VMs are not burstable ones. We also define *max_ondemand* as the maximum number of on-demand VMs that can be allocated simultaneously. This value is determined by the cloud provider. For instance, in Amazon EC2, by default, this value is 20 VMs per region.

Each $vm_j \in M$ has a memory capacity of m_j gigabytes, and a set of cores VC_j . Thus, since we consider two markets, the spot and the on-demand, each $vm_j \in M$ is present in only one of them with cost c_j . As stated in Chapter 2, in October 2017, Amazon adopted the per-second billing in Linux VMs, in which users are billing in one-second increments [21]. Therefore c_j corresponds to the cost in seconds of vm_j . Besides, each burstable $vm_j \in M^b$ has a current CPU credit amount cc_j that is constantly updated by the cloud provider (in the case of $vm_j \notin M^b$, i.e., non-burstable VMs, $cc_j = \infty$).

Let B be the set of tasks of the BoT application. We assume that each task $t_i \in B$ executes in only one core of a VM, requiring a known amount of memory rm_i , which must be available throughout t_i 's execution. Therefore, a multi-core VM can execute two or more tasks simultaneously (one task per core) provided that there is enough main memory for all of them. We also consider that the time required to execute each task t_i in a $vm_j \in M$ is known and given by e_{ij} .

The user defines D , which is the deadline to finish the execution of all tasks of the application with regard to the time when the application started. We then define $T = \{1, \dots, D\}$ as the set that discretizes such an execution in time intervals. Note that, in this work, we consider that a deadline miss as a *temporal failure*, where the correctness of the execution is also related to the capacity to meet the deadline.

Figure 4.1 shows the execution of tasks of an application in a spot VM. As we can observe, each core starts executing a task. However, due to the lack of memory to fulfill memory requirements of both tasks 1 and 5 at the same time, there is a gap between tasks 4 and 5 which induces $core_0$ to remain idle until task 1 finishes.

In an environment where spot VMs can hibernate and resume multiple times, let *break-point* of vm_j be the time $p \in T$ when the last hibernation of vm_j started. If vm_j resumes in time to satisfy the application deadline, as shown in Figure 4.2, its tasks can go on running from the *break-point*.

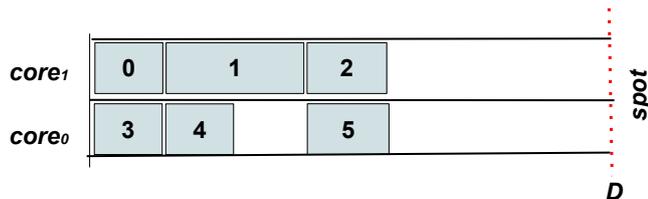


Figure 4.1: Execution without hibernation

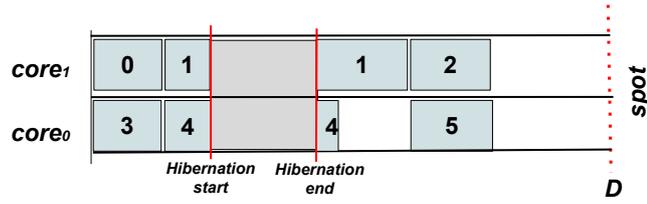


Figure 4.2: Execution with hibernation

All variables and parameters defined in this Chapter are summarized in Table 4.1.

Table 4.1: Variables and parameters of the problem.

Name	Description
B	Set of tasks
$M = M^s \cup M^o \cup M^b$	Set of VMs that can be used
M^s	Set of spot VMs that can be used
M^o	Set of on-demand VMs that can be used
M^b	Set of Burstable VMs
T	Discretized time set
D	Deadline defined by the user
vm_j	Virtual machines
m_j	Memory capacity of vm_j in gigabytes
VC_j	Set of cores of vm_j
c_j	Cost in seconds of vm_j
cc_j	Current amount of CPU credit
$max_ondemand$	Maximum number of on-demand VMs allocated simultaneously
t_i	Task t_i
rm_i	Amount of memory required by t_i
e_{ij}	Time required to execute t_i in a vm_j
$break_point$	Discrete time when vm_j hibernates
α	Time intervals a new deployed VM takes to receive the migrated tasks

4.2 Architecture of HADS and Burst-HADS Frameworks

HADS and Burst-HADS frameworks use a master-worker architecture standard. The master is responsible for defining the scheduling plan, choosing the VMs, requesting the resources to the provider, and, eventually, migrating tasks to other VMs. On the other hand, the workers operate as asynchronous daemon applications which run in the back-

ground on each deployed VM. The workers' main role is monitoring the VMs and the tasks to communicate all status changes to the master.

Workers are also an interface between the master and the deployed VMs. For example, when the master needs to configure a storage service in one specific VM, it sends a command to the corresponding worker, which executes all the configuration procedures. All the master's commands are sent throughout the network by using the HTTP protocol. Figure 4.3 represents the frameworks general architecture. Note that, in the example of Figure 4.3, the master runs on a local computer. However, it could be executed in a VM on the cloud. But, since we need the guarantee that the master will always be available, it must be executed in an on-demand VM.

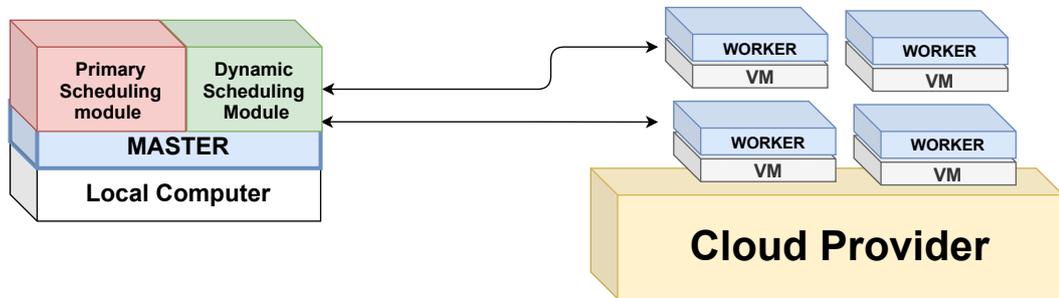


Figure 4.3: General architecture of the Frameworks

To deploy and terminate the VMs, the master uses the SDK offered by the cloud provider. In the case of AWS, the frameworks use Boto3 [13], an SDK for Python 3.0. Boto3 allows users to develop python scripts that can create, update, and delete AWS resources. Boto3 goes from the management of the services and resources of EC2 to the consult of the users' bill. All Boto3 functionalities used by the proposed frameworks are packaged in a class of the framework that standardizes function callings, input parameters, and outputs. Therefore, HADS and Burst-HADS are independent of the cloud provider's SDK, allowing them to support any provider easily.

Figure 4.4 presents the frameworks' execution steps. Firstly, to initiate a new BoT execution on the cloud, the user needs to generate two JSON files [65] that describe the job (the BoT application) and the available VMs (the execution environment). In Figure 4.4, those files are presented as *job.json* and *env.json*, respectively. Also, the user needs to define some input parameters, for example, the deadline D and the storage service. Note that both frameworks have utility tools to help the users generate the input JSON files.

Figure 4.5 present an example of the *env.json* and *job.json* files. As can be seen, the *env.json* presents the information of the VMs type *c3.large* and *t3.xlarge*, while the

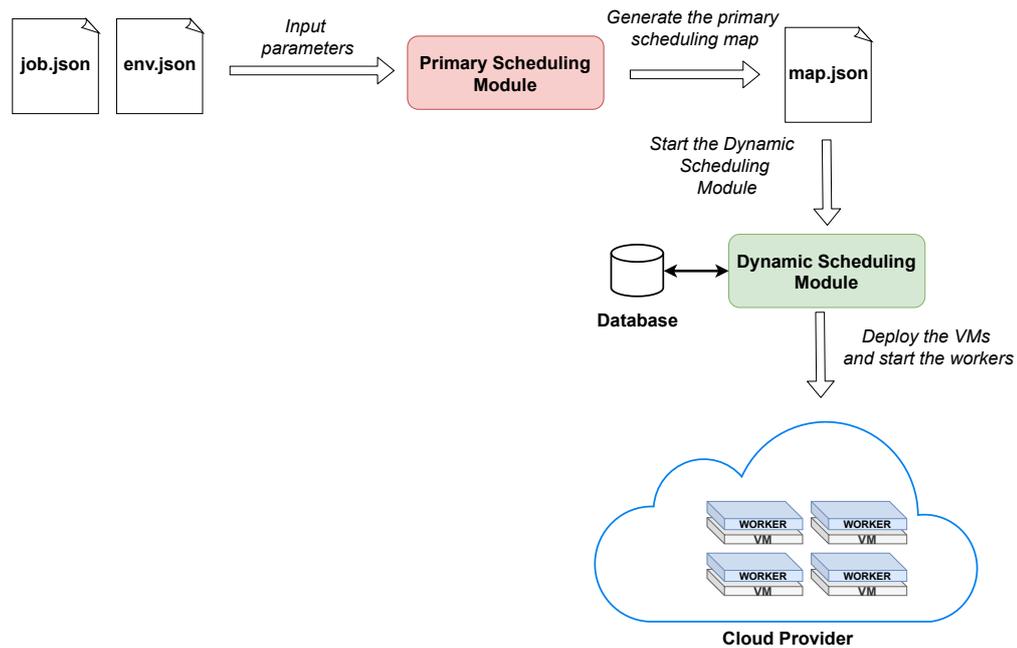


Figure 4.4: Frameworks' execution steps

job.json represents a synthetic job with three tasks. The provider gives all VMs' information presented in the *env.json*. The file also includes what market the VM can be launched in and indicates if the VM is burstable or not. In the case of the *job.json*, we also have the execution time of each task in each one of the VMs presents in the *env.json*, the task' memory requirement, and the command that need to be executed by the worker at the moment of the task execution.

```

env.json
1 {
2   "instances": {
3     "c3.large": {
4       "vcpu": 2,
5       "memory": 3.75,
6       "markets": {
7         "on-demand": "yes",
8         "spot": "yes"
9       },
10      "burstable": {
11        "burstable": "no",
12        "cpu_credit_rate": 0,
13        "baseline": 0
14      }
15    },
16    "t3.xlarge": {
17      "vcpu": 4,
18      "memory": 16.0,
19      "markets": {
20        "on-demand": "yes",
21        "spot": "no"
22      },
23      "burstable": {
24        "burstable": "yes",
25        "cpu_credit_rate": 96,
26        "baseline": 30.0
27      }
28    }
29  }
30 }
31 }

job.json
1 {
2   "job_id": 10,
3   "job_name": "Test",
4   "description": "A synthetic job",
5   "tasks": {
6     "0": {
7       "command": "sh exec.sh 400",
8       "memory": 10.060959,
9       "runtime": {
10        "c3.large": 167.0,
11        "t3.xlarge": 119.0
12      }
13     },
14     "1": {
15       "command": "sh exec.sh 400",
16       "memory": 10.060959,
17       "runtime": {
18        "c3.large": 167.0,
19        "t3.xlarge": 119.0
20      }
21     },
22     "2": {
23       "command": "sh exec.sh 400",
24       "memory": 10.060959,
25       "runtime": {
26        "c3.large": 167.0,
27        "t3.xlarge": 119.0
28      }
29     }
30   }
31 }
32 }
  
```

Figure 4.5: Examples of the *env.json* and *job.json* input files

All the user-given information is used as input to the Primary Scheduling Module (Burst Primary Scheduling Module, in the case of Burst-HADS). The Primary Schedul-

ing Module generates the initial scheduling map by executing some heuristics that use the input information to define the initial scheduling strategy. Those heuristics will be presented in the next chapters. As shown in Figure 4.4, the module outputs the map also in the JSON format (represented as *map.json* in the figure). Choosing the JSON format has the following motivations: i) that format is human-readable, allowing users to understand the map and the input parameters; ii) JSON files are easily editable, allowing users or third-party tools to change the scheduling and input parameters; and iii) JSON files can be easily extended to support new functionalities or features of the frameworks.

After generating the primary scheduling map, the framework starts the Dynamic Scheduling Module. That module requests the VMs defined by the Primary Scheduling Module and dispatches the tasks according to the initial scheduling plan described in the *map.json*. Then, the Dynamic Scheduling Module manages all the BoT execution, reacting to events, such as spot hibernations, that can cause any deadline violation.

Besides the execution’s management, the Dynamic Scheduling Module is also responsible for starting the workers at each VM. Moreover, whenever a new BoT application is submitted to the execution, the environment must be ready to receive and execute the tasks. In other words, libraries and dependence files need to be installed on the VMs before the execution. However, instead of installing the dependencies directly on the VMs, HADS and Burst-HADS use nested containers in the VMs.

Currently, the frameworks use the Docker platform [59]. When a new container is created, a VM image (Amazon Machine Image, in the case of AWS) holding the containers is also created. Thus, if a BoT application needs to be executed multiple times, the frameworks do not need to create the same containers again. Instead, they load that image at the moment of the VMs deployment.

The Dynamic Scheduling Module is also responsible for coordinating the recording of tasks’ checkpoints and for contracting and configuring the cloud storage service that will be used to store all checkpoint files. Thus, when the worker is started, the master sends a command to define the storage system. When the worker completes the storage system mounting process, tasks can then start executing. During the execution, the worker records checkpoints according to a pre-defined checkpoint interval (see Section 6.1.2). In this work, the checkpoints are recorded using the Checkpoint Restore In Userspace tool (CRIU) [29]. CRIU is a widely used checkpointing tool that can record the state of individual applications. Different from other tools, CRIU is implemented in the user space and does not need superuser privileges. Moreover, it has a straightforward installation,

and it is available in most Linux repositories.

As represented in Figure 4.4, the Dynamic Scheduling Module has access to a database, which is used to store all events that occur along with the execution and keep information of the resources and tasks, including which tasks each VM executed and task migrations events. Each event stored in the database has a timestamp, allowing users to track the events chains. Moreover, by querying the database, the framework can define the execution total monetary cost and time.

Chapter 5

Evaluating AWS Storage services for Checkpointing and Recovering

When using a checkpoint/recovery approach, it is essential to ensure that all files required for recovery of the application will always be available in case of failure. In cloud environments, different storage services can be hired and used along with the VMs. As presented in Chapter 2, AWS offers several options of storage services [15], with different features, prices and purposes. Considering the HADS and Burst-HADS framework, we are interested in using three general-purpose storage services offered by AWS (Amazon S3, EBS, and EFS) to store and recover checkpoints files of BoT tasks running in spot VMs. Thus, in this Chapter, we conduct a study evaluating the impact of such approaches on the time and monetary cost of the execution, considering the use of each one of those storage services. All experimental tests presented in this Chapter were performed using VMs of type c3.2xlarge, composed of 8 vCPUs and 16 GBs of memory. Moreover, we also used the synthetic application proposed in [4], which allows us to define the associated task's memory footprint and execution time.

This Chapter is divided into three sections. Firstly, in Section 5.1, we evaluate and measure the times required to record checkpoints on the three evaluated services. Next, in Section 5.2, we analyze the impact of checkpoints in the execution time of the application, considering a scenario where there are no spot hibernations. Moreover, we also evaluate the overheads associated with the recovery procedure. Finally, in Section 5.3, we estimate how checkpoints considering each one of three services would impact the execution's monetary cost.

5.1 Dump Time Evaluation

According to [25] "the dump time is the overhead spent writing out the checkpoint files required to restart the application after an interrupt". To characterize that overhead, concerning the evaluated storage services, we create a set of synthetic tasks with memory footprint varying from 140 MB to 7,750 MB (one task by memory size). For each task, we define an average base time, i.e. execution time without checkpoint and revocations, of ≈ 20 minutes and a fixed checkpointing interval of 5 minutes, resulting in 4 checkpoints per task. The dump time evaluation is presented in Figure 5.1. The Y-axis in Figure 5.1 is in log scale for better visibility of the small times.

As shown in Figure 5.1, the S3 service presented by far the worst results in the evaluated scenarios. Consider the task with the smallest footprint size (140 MB), when using S3, the dump time was 89.35% higher than EFS and 96.75% higher than EBS. In fact, for all used memory ranges, the dump time with S3 presented an increment of 72.57% and 89.37% on average when compared to EFS and EBS, respectively. Besides that, in S3, the larger the size of the memory footprint, the greater the amount of variation in dump time. The same amount of variation is not seen in the other evaluated services.

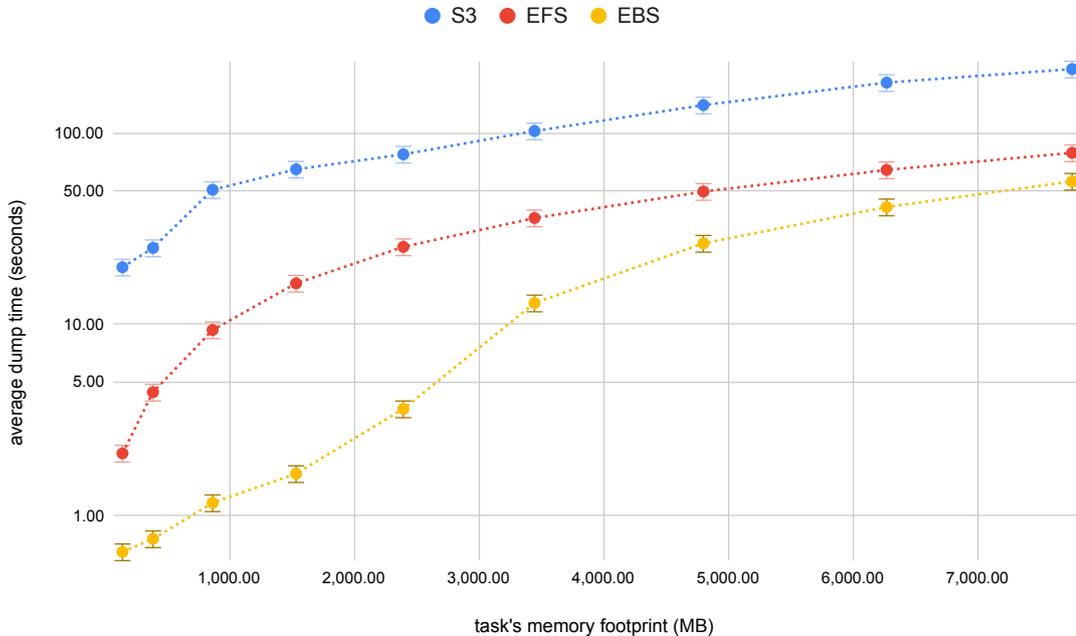


Figure 5.1: Dump time in S3, EBS and EFS for different sizes of memory footprint

It is worth mentioning that the results of S3 are intrinsically linked to the tool used to create the storage system. As said before, S3 does not offer native support to a file

system, requiring third-party tools to it. Therefore, we would probably see different results for S3 experiments with different file system tools. Moreover, any alteration in a file in S3 requires the download of the entire file to a local storage system. So, if an incremental checkpoint technique is adopted, in which a new checkpoint modifies the previous checkpoint file (according to the current state of the application), possibly the use of S3 will not compensate for the higher cost.

On the other hand, EBS presented the best results, with dump time varying from 0.65 to 55.82 seconds, followed by EFS (2.12 to 78.73 seconds). Those results were expected since EBS is a local storage service and thus had no additional transfer overhead associated with the storage operation. However, it is important to mention that in the scenario where multiple tasks are sharing the same EBS volume to store their checkpoint files, the recovery procedure cannot spread those tasks easily among different VMs. However, it is important to note that the EBS volume is independent of the VM, i.e. the volume and all the data stored in it is maintained by the provider even when the VM is revoked or terminated.

5.1.1 Dump Time evaluation of Concurrent Checkpoints

To evaluate the impact of concurrent checkpoints in the shared storage services S3 and EFS, the task with the biggest memory footprint (7,750 MB) was executed considering scenarios where one, two, four, and six VMs shared the same file system. To avoid concurrency with other resources, we considered only one task per VM. Moreover, the checkpoint recording was synchronized by HADS, and the task execution time was also around 20 minutes with an interval of 5 minutes between checkpoints. Figure 5.2 shows the results of this test.

As can be seen in Figure 5.2, while in EFS the dump time increased with the number of VMs, in S3, the dump time was almost the same, for all cases. For one VM, the average dump time with S3 was 65.92% greater than EFS. However, with two VMs, that difference drops to 46.31%. At the four VMs scenario, the time already becomes bigger in EFS than S3 (3.03% of increment). In the six VMs scenario, the dump time with concurrent checkpoint recording increased 37.89% with EFS in comparison to S3.

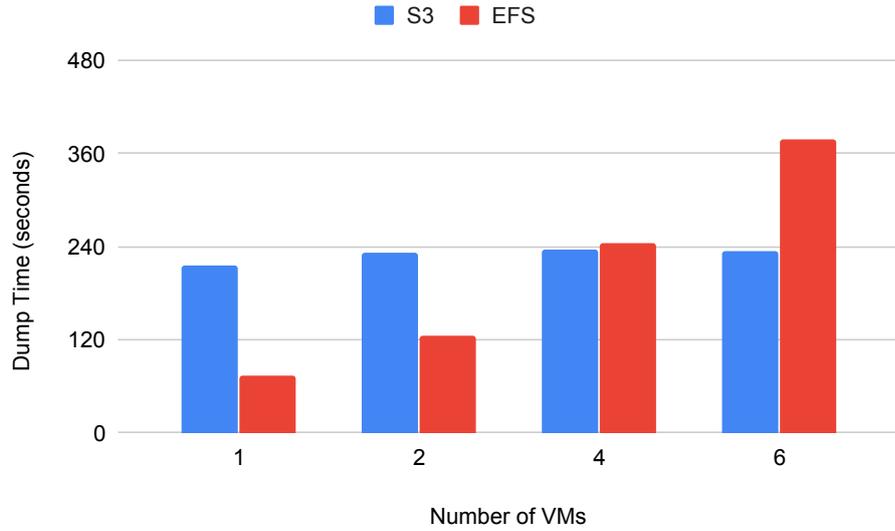


Figure 5.2: Dump time of concurrent checkpoints in S3 and EFS storage services

Those results show that to adopt EFS, the checkpoint/recovery solution has to consider the number of concurrent recordings, once the performance of this service showed a fast degradation in the evaluated scenarios.

When we consider several VMs, each one with its local storage (EBS), there is no concurrency in the storage system to be analyzed. Also, the tool used in this work to record the checkpoint (CRIU) does not allow concurrent checkpoints of tasks running in the same VM. When two or more checkpoints requests are submitted, the tool queues them and executes the recordings one at a time. Thus, the concurrency in EBS service was not analyzed in this work.

5.2 Overall Overhead Analysis

Figure 5.3 presents the average percentages of time spent by the frameworks operations in the scenario where there are no spot revocations. Once again, the task with the biggest footprint size was used in this evaluation, and the checkpoints were recorded with a 5 minutes interval. Since there is no spot revocation, in this scenario, there are three distinct operations: spot VM launching, checkpoint recording and task execution.

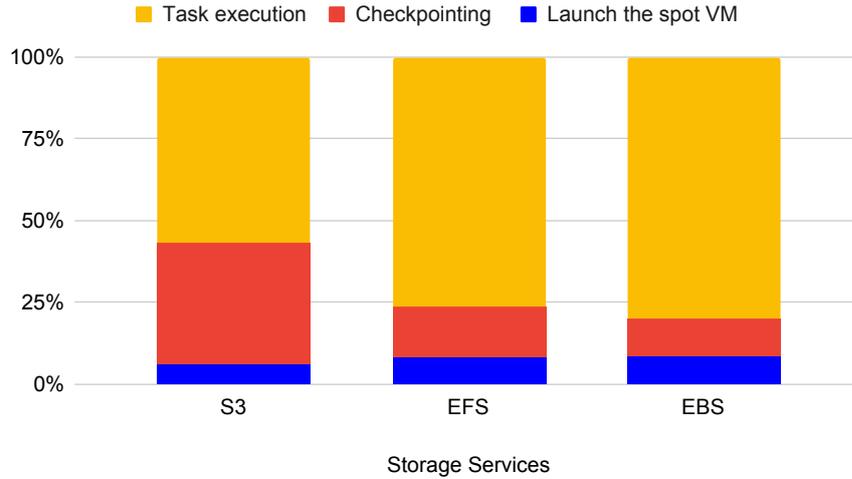


Figure 5.3: Overheads of launching a spot VM and recording checkpoints

The overhead of launching a spot VM includes the time for requesting the VM, configuring and mounting the storage system and, finally, booting the VM. As can be seen in Figure 5.3, the overhead of that operation is almost the same for all services. In the case of S3 that overhead was 6% of the total execution time, while in EFS and EBS it was 8%. As explained in Section 2.2.2, differently from the other services, EBS has to create the storage volume, what results in additional overhead, that is smaller than the time required to boot a VM.

The checkpointing overhead presented in Figure 5.3 confirms the results of Section 5.1, where the overhead percentage depends on the used storage service.

The increment in the total execution time of the application is equal to the product of the number of recorded checkpoints by the dump time. The checkpoint time represented 37.1%, 15.4% and 11.4% of the total execution time using the services S3, EFS and EBS, respectively.

Finally, considering all overheads caused by the checkpoint procedure and the VM launching, the useful work accomplished by the VM considering the scenario of Figure 5.3, i.e., the execution of the task itself, represents 56.7% of the total execution time using S3, 76.5% in the case of EFS and 79.8% using EBS. Therefore, if the user aims to maximize the useful work executed by the VM, the services EFS and EBS seems to be better than S3.

Figure 5.4 shows the overhead of the task recovery procedure concerning the evaluated storage services. To estimate that overhead, a controlled revocation was implemented,

considering the execution of the largest execution time task and a 5 minutes checkpoint interval. The base execution time of the task is around 20 minutes and the VM revocation was emulated by terminating the VM before 10 minutes of execution. Thus, in this test, only one checkpoint was recorded before the revocation (saving the first 5 minutes of execution). Since only one spot VM was used, the time presented in Figure 5.4 includes the time necessary to launch a new on-demand VM, to set up the storage system and to recover the task using CRIU [29].

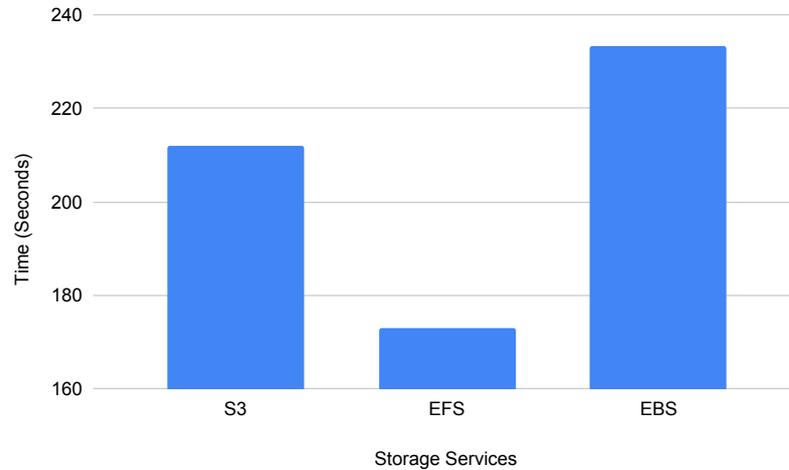


Figure 5.4: Recovery Procedure Times

As can be seen, EBS presented the largest overhead in the recovery procedure. The time of EBS is 9.14% higher than S3 and 25.86% higher than EFS. EBS requires that the VM is associated with the storage volume to access it. To mount an EBS volume and associate it to a VM, HADS uses the API Boto3. However, during a VM revocation, HADS cannot associate the volume to other VM until the complete termination of the revoked VM. In the tests presented in Figure 2, 148 seconds on average were spent to terminate the spot VM. That result shows that although EBS has the best results in terms of dump time (Section 5.1), the mechanism of association caused an additional overhead on the recovery procedure. Note that, in the case of S3, the time spent in the recovery is mostly due to the time necessary to recover the task (121 seconds on average). So, in our tests, S3 presented the worst results both for recording checkpoints and for recovering these checkpoints files as well.

Table 5.1: Monetary Costs of Services S3, EBS and EFS in a Long-running Application

Checkpoint Interval (h)	# of Checkpoints	Total Execution Time (h)			Total Monetary Cost (US\$)		
		S3	EBS	EFS	S3	EBS	EFS
1	720	763.14	731.16	735.75	\$23.13	\$24.50	\$30.64
5	144	728.63	722.23	723.15	\$22.11	\$24.24	\$30.27
10	72	724.31	721.12	721.57	\$21.99	\$24.20	\$30.22
15	48	722.88	720.74	721.05	\$21.94	\$24.19	\$30.20
20	36	722.16	720.56	720.79	\$21.92	\$24.19	\$30.20
25	28	721.68	720.43	720.61	\$21.91	\$24.18	\$30.19

5.3 Monetary Cost Estimation

In our previous tests, we have executed only short-running applications with tasks requiring less than one hour of execution. Moreover, all the files stored in the evaluated storage services were deleted with less than 24 hours after the executions. In those cases, the type of storage service used for checkpoint and recovery does not impact the monetary cost of the executions. However, based on the dump times previously obtained and the monetary cost of the storage services obtained from [21], we could estimate the monetary costs of long-running tasks for each storage service.

To compute those costs, we considered an application with only one task with the same characteristics of the largest task presented in Figure 5.1, executing for 30 days without any interruption or revocation. We also considered that throughout the execution, the dump time did not vary. In terms of storage, the users are charged at 30 days based price (Section 2.2.2), and we assumed that 30 GBs of data were kept in the storage service, including the checkpoint files, along those days. Additional charges of those storage services were not considered.

The estimated results are presented in Table 5.1, where the first column shows the checkpoint intervals, followed by the number of checkpoints, obtained by dividing the task execution time by the dump time. Thus, we can estimate the total execution time by adding the total checkpointing overhead to the task execution time. That time is then used to compute the monetary costs of the execution. As can be seen in Table 5.1, although the estimated execution time using S3 is the greatest one, it presents the lowest monetary cost estimation. The monetary cost using S3 is 9.43% less than EBS and 36.66% less than EFS. Thus, although the cost of VM hiring increases with S3 (because of the increase in the execution time, the VM has to be hired for a longer time), the saving with that type of storage service makes it attractive, considering the total cost.

Figure 5.5 shows the monetary cost that would be paid for the VM and the storage

service, considering the example where 25 hours is used between the checkpoints (that case is presented in the last row of Table 5.1).

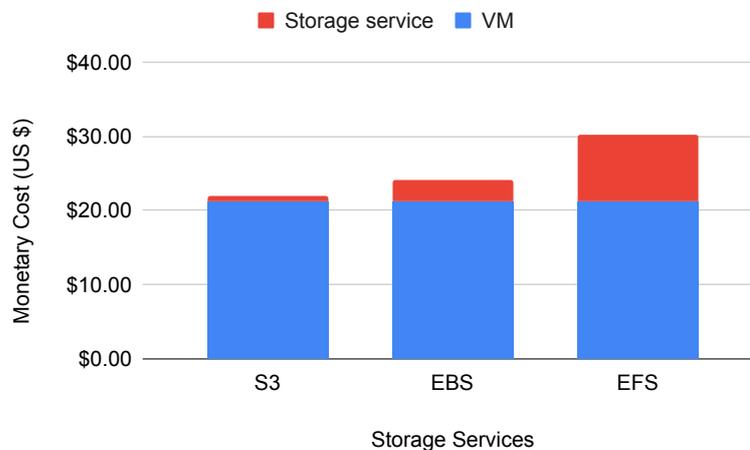


Figure 5.5: Monetary Costs of Storage and VM hiring with Services S3, EBS and EFS

In this example, while the user pays US\$0.69 for the 30 GBs stored for 30 days in S3, in EBS and EFS those costs are US\$3.0 and US\$9.01, respectively. Therefore, there is a clear connection between the cost of the service and its performance. That connection makes S3 a competitive option when the monetary cost is more important than the execution time. Moreover, we can also see in Table 5.1 that EFS is the most expensive service. That result can be even worse when the total execution time of EFS and EBS are analyzed. As can be seen, the difference in the execution times of those services is less than 1%. However, the monetary cost with EFS is 19.93% higher than EBS. Therefore, the adoption of EFS is justified only in cases where a shared storage system with performance near a local storage system are necessary.

From the results presented in this Chapter we see that the EBS service showed the best tradeoff between monetary cost and performance. However, as that service is a local storage system, it prevents tasks sharing the same EBS volume from being spread among different VMs in the recovery process. Moreover, in the recovery process, that service presents an additional overhead caused by the necessity of associating the storage volume containing the checkpoint with the VM, where the recovered task will execute. In the case of EFS, the service presented checkpointing and recovery times close to EBS but with higher monetary costs than the other services. Also, the EFS performance showed a significant degradation with concurrent checkpoint records. We also see that S3 can be a very attractive option when the monetary cost is more important than the application's total execution time and also for concurrent checkpoints. Thus, since our main concern is

the monetary cost, for the evaluations of HADS (Chapter 7) and Burst-HADS (Chapter 9) we used S3. However, note that, for the study case presented in Section 9.5, the execution time of the application was very impacted by S3. Moreover, we do not record checkpoints in this study case. Thus, in this case, we adopt EBS.

Chapter 6

Hibernation Aware Dynamic Scheduler

In this Chapter, we present the HADS framework. At first, in Section 6.1, we introduce HADS' Primary Scheduling Module, which creates an initial scheduling map based on the tasks' execution time estimation, the deadline, and sets of on-demand and spot instances. Next, in Section 6.2, we present the Dynamic Scheduling Module, including some scheduling concepts adopted by HADS and the functions that handle spot instance hibernations and task migrations.

6.1 Primary Scheduling Module

This section presents HADS' Primary Scheduling Module. Firstly, we define the D_{spot} value, a parameter that delimits the primary tasks makespan. This limit is necessary to ensure that there will be enough remaining time to execute the application in case of hibernations, respecting the deadline D . Next, we show how the checkpoint intervals are defined by the framework and present the heuristics used to create the initial scheduling map.

6.1.1 Estimation of D_{spot}

To avoid a temporal failure caused by a spot VM hibernation, the non-finished tasks, which include running tasks and tasks waiting to be executed, should be migrated to other VMs. Thus, we have to guarantee that there will always be enough spare time to migrate and execute those tasks before the deadline of D . Therefore, aiming at ensuring the application deadline no matter if and when spot VM hibernations could happen, we determine the D_{spot} value. The D_{spot} is the worst-case estimated makespan and is used to

guarantees enough spare time to migrate tasks of any hibernated spot VM to other VMs and execute them before the deadline D . D_{spot} is computed by considering the deadline D and the execution time of the longest tasks that might need to be migrated and executed to/in the slowest VMs.

Algorithm 1 renders the estimation value of D_{spot} . The algorithm receives, as input, the set of tasks B , the set M of VMs, the maximum number of on-demand VMs that can be allocated simultaneously (*max_ondemand* parameter) and the deadline D . The average number of tasks n that could be assigned to a VM can be estimated by dividing the number of tasks in B by the maximum number of VMs (line 1). Then, the n longest tasks are included in set $L \subset B$ (line 2). Therefore, the execution time of these tasks in the slowest VM of M , vm_s , obtained by calling procedure *get_slowest_vm* (line 3), characterizes the worst case makespan, denoted mkp_w . Note that the tasks are not actually scheduled to vm_s . The *assign* function just emulates the scheduling of each $t_i \in L$ to vm_s .

In line 7, mkp_w , is estimated by calling the procedure *get_makespan* which considers the slowest VM of M , vm_s , D_{spot} is, then, the difference between D and mkp_w plus α , the overhead to migrate the tasks (line 8). If D_{spot} is equal to zero, the scheduler deploys only on-demand VMs. Figure 6.1 illustrates the D_{spot} estimation.

Algorithm 1 *compute_* D_{spot}

Input: $B, M, max_ondemand$, and D

- 1: $n \leftarrow \lceil \frac{|B|}{max_ondemand} \rceil$
 - 2: $L \leftarrow get_longest_tasks(n, B)$
 - 3: $vm_s \leftarrow get_slowest_vm(M)$
 - 4: **for all** $t_i \in L$ **do**
 - 5: *assign*(t_i, vm_s)
 - 6: **end for**
 - 7: $mkp_w = get_makespan(vm_s)$
 - 8: return $max(D - (mkp_w + \alpha), 0)$
-



Figure 6.1: D_{spot} definition

The time complexity of Algorithm 1 depends on the execution of functions *get_longest_tasks* (line 2) and *get_slowest_vm* (line 3). Besides, in line 4, a for-loop is exe-

cuted for the n longest tasks. Thus, since the complexity of the functions *get_longest_tasks* is $\mathcal{O}(|B|)$, and *get_slowest_vm* is $\mathcal{O}(|M|)$, the complexity of Algorithm 1 in the worst case is $\mathcal{O}(|B| + |M| + n)$. However, as $|B| > n$ and $|B| > |M|$ the algorithm has time complexity $\mathcal{O}(|B|)$, where $|B|$ is the size of the set of tasks B and $|M|$ is the size of the set of VMs M . Note that, the algorithm receives as input the set B of tasks and the set M of VMs as lists of integers, where each element is either a task ID or a VM ID. Thus, as $|B| \gg |M|$, in terms of space complexity we have $\mathcal{O}(|B|)$ as well.

6.1.2 Checkpoint Intervals

Since in BoT application tasks execute independently to each other, an uncoordinated checkpoints approach is very suitable because it does not require any global tasks synchronization. Hence, only the last checkpoint needs to be kept for each task. We consider that, when executing a task t_i scheduled to a spot VM, it will have its checkpoint periodically saved.

We define the input parameter *ovh* as the maximum percentage of time overhead that the checkpoint mechanism is allowed to add in the original execution time of a task. The necessary time to record a checkpoint of a task t_i , called *dump*(t_i) increases with the task's memory size. Consequently, the number of checkpoints, n_ckp_i , taken along the execution of t_i scheduled to spot vm_j , is defined by Equation 6.1. Note that, we consider that the execution time of a task t_i allocated to a vm_j increases *ovh* percent ($e_{ij} = e_{ij}(1 + ovh)$).

$$n_ckp_i = (e_{ij} \times ovh) / dump(t_i) \quad (6.1)$$

In Equation 6.2, we also define *intv_ckpt_i* as the time interval between two consecutive checkpoints of task t_i scheduled to spot vm_j .

$$intv_ckpt_i = e_{ij} / n_ckp_i \quad (6.2)$$

Note that in this work checkpoints are taken in parallel. Therefore, tasks running in the same VM can have their checkpoints recorded concurrently.

6.1.3 Primary Scheduling Heuristic Algorithm

Algorithm 3 describes the primary scheduling heuristic, which is a greedy algorithm that schedules a set of application tasks $t_i \in B$ to a set of spot and on-demand VMs. The algorithm receives as input the set B of tasks, sets of VMs M , M^s and M^o , the deadline D , the parameter ovh that indicates the maximum percentage of time overhead induced by checkpoint, the migration overhead α and the *max_ondemand* parameter. The primary scheduling heuristic has three distinct phases: i) scheduling of a task to an already selected VM; ii) scheduling of a task to a new spot VM; and iii) scheduling of a task to a new on-demand VM.

Initially, in line 1, Algorithm 3 computes the estimated time limit D_{spot} (previously explained in Section 6.1.1). Then, it sorts the tasks of B in descending order by their memory size requirements (line 2) and tries to schedule them, following the three phases in the order. By calling the procedure *check_schedule* (Algorithm 2), the heuristic verifies if it is possible to schedule t_i in the selected vm_j of the current phase, i.e., if t_i is scheduled in vm_j , (1) memory requirements must be satisfied and (2) the application deadline D (resp., D_{spot}), in case of on demand (resp., spot) vm_j , should not be violated (line 4 of Algorithm 2). If these two conditions are not ensured, the algorithm goes to the next phase; otherwise t_i is scheduled and the algorithm tries to schedule the next task of B .

Note that if vm_j is a spot VM, the time to execute t_i should include the overhead ovh induced by the checkpoints (line 2 of Algorithm 2). Furthermore, for spot VMs, Algorithm 3 also computes *intv_ckp*, the interval time between two consecutive checkpoints using Equation 6.2 presented in Section 6.1.2 (lines 9 and 22). The three phases of Algorithm 3 are described next.

Phase 1: Scheduling tasks to an already allocated VM avoids VM deploying time. Thus, for each task $t_i \in B$, the algorithm tries to schedule it in a core of a virtual machine vm_j from A , the set of already selected VMs (lines 7 to 17). It firstly tries spot VMs. We point out that for the first task, *Phase 1* is bypassed since A is initially empty.

Phase 2: If t_i can not be scheduled to a vm_j of A , the algorithm tries to select a new spot VM (lines 19 to 28) using a weighted round-robin algorithm (WRR) [48]. In WRR, each spot VM has an associated weight and the algorithm selects the VMs in round-robin way, according to such weights. As shown in Equation 6.3, the weight of vm_j , $weight(vm_j)$, is equal to the quotient between $Gflops_j$ of vm_j , and c_j , the price of the VM per second. $Gflops_j$ of vm_j is estimated by using the LINPACK benchmark [28]

and express the computing power of this VM.

Our choice in using WRR and spot VMs with different configurations is in agreement with Amazon’s recommendations¹ that say that an application should use different types of spot VMs to increase the availability of spot VM instances. According to Kumar et al. [50], interruptions of spot VMs, which include hibernation, usually take place in VMs of the same type. Therefore, a choice of heterogeneous spot VMs minimizes the impact of VM hibernations.

$$weight(vm_j) = Gflops_j/c_j, \text{ where } vm_j \in M \quad (6.3)$$

Phase 3: If it is not possible to allocate a new spot VM to schedule t_i , the algorithm schedules it in the cheapest on-demand VM instance (lines 30 to 35).

Finally, when all tasks have been scheduled, the algorithm creates the primary scheduling map (line 37), which describes the initial execution strategy that the Dynamic Scheduling Module of HADS should follow.

Algorithm 2 *check_schedule*

Input: t_i , vm_j , D_k and ovh

- 1: **if** $vm_j \in M^s$ **then**
 - 2: $e_{ij} \leftarrow e_{ij} + (e_{ij} \times ovh)$
 - 3: **end if**
 - 4: **if** $start_{t_i} + e_{ij} < D_k$ **and** $enough_mem(rm_i, m_j)$ **then**
 - 5: **return True**
 - 6: **else**
 - 7: **return False**
 - 8: **end if**
-

¹<https://aws.amazon.com/pt/ec2/spot/instance-advisor/>

Algorithm 3 *Primary Scheduling Heuristic***Input:** B, M, M^s, M^o, D, ovh and $max_ondemand$

```

1:  $D_{spot} = compute\_D_{spot}(B, M, max\_ondemand, D)$ 
2:  $sort\_by\_memory(B)$  {Sort tasks by memory requirement  $rm_i$ }
3:  $A \leftarrow \emptyset$  {Set of selected VMs}
4: for all  $t_i \in B$  do
5:   {Phase 1: Try to schedule the task to an already selected VM}
6:    $sort\_by\_price(A)$  {Sort the selected VMs by price}
7:   for all  $vm_j \in A$  do
8:     if  $vm_j \in M^s$  and  $check\_schedule(t_i, vm_j, D_{spot}, ovh)$  then
9:        $intv\_ckp_i \leftarrow compute\_intv\_ckp(t_i, vm_j, ovh)$ 
10:       $schedule(t_i, vm_j, intv\_ckp_i)$ 
11:       $break$  {Schedule next task}
12:     end if
13:     if  $vm_j \in M^o$  and  $check\_schedule(t_i, vm_j, D, -)$  then
14:        $schedule(t_i, vm_j, -)$ 
15:        $break$  {Schedule next task}
16:     end if
17:   end for
18:   {Phase 2: Try to schedule the task to a new spot VM}
19:   if not  $scheduled$  then
20:      $vm_k \leftarrow get\_wrr\_VM(M^s)$  {Select a spot VM using the weighted round-robin heuristic}
21:     if  $check\_schedule(t_i, vm_k, D_{spot}, ovh)$  then
22:        $intv\_ckp_i \leftarrow compute\_intv\_ckp(t_i, vm_k, ovh)$ 
23:        $schedule(t_i, vm_k, intv\_ckp_i)$ 
24:        $A \leftarrow A \cup \{vm_k\}$  {Update the set of selected VMs}
25:        $M^s \leftarrow M^s \setminus \{vm_k\}$ 
26:        $break$  {Schedule next task}
27:     end if
28:   end if
29:   {Phase 3: Schedule task to the cheapest new on-demand VM}
30:   if not  $scheduled$  then
31:      $vm_k \leftarrow get\_cheapest\_vm(M^o)$ 
32:      $schedule(t_i, vm_k, -)$ 
33:      $A \leftarrow A \cup \{vm_k\}$  {Update the set of selected VMs}
34:      $M^o \leftarrow M^o \setminus \{vm_k\}$ 
35:   end if
36: end for
37:  $map \leftarrow create\_primary\_map(A)$ 
38: return  $map$ 

```

For each phase of Algorithm 3, we have the following time complexity. Since functions

check_schedule (see Algorithm 2) and *compute_intv_ckp* have both time complexity $\mathcal{O}(c)$, the phase 1 of Algorithm 3 (lines 7 to 17) has complexity $\mathcal{O}(|A|\log|A|)$, where $|A|$ is the size of the set of selected VMs A . The phase 2 (lines 19 to 28) has complexity $\mathcal{O}(c)$ which is the complexity of function *get_wrr_VM*, that returns the next VM according to the weight computed with Equation 3. Finally, the phase 3 (lines 30 to 35) has complexity $\mathcal{O}(|M^o|)$, that is the time spent by function *get_slowest_vm*(M^o), where $|M^o|$ is the size of set M^o of on-demand VMs. Therefore, the complexity of the algorithm is $\mathcal{O}(|B| \times (\log|B| + |A|\log|A| + |M^o|))$, where $\mathcal{O}(|B|\log(|B|))$ is the complexity of the merge sort algorithm executed on line 2. Moreover, Algorithm 3 receives four lists of integers, representing sets B , M , M^s and M^o , where each element in a list represents either the ID of a task or of a VM. Thus, as $|B| \gg |M|$, in terms of space complexity Algorithm 3 is $\mathcal{O}(|B|)$.

The functions and procedures called by HADS' Primary Scheduling Module algorithms are summarized in Table 6.1, while variables and parameters are presented in Table 6.2.

Table 6.1: Functions and Procedures called by Primary Scheduling Module Algorithms

Name	Description
<i>get_longest_tasks</i> (n, S)	Renders the n longest tasks of set S
<i>get_slowest_vm</i> (M)	Renders the slowest VM of M
<i>assign</i> (t_i, vm_s)	Emulates the assignment of task t_i to vm_s . In this case task t_i is not actually scheduled to vm_s
<i>get_makespan</i> (vm_s)	Renders the worst case makespan as if the n longest tasks had been scheduled to the slowest VM of M
<i>sort_by_memory</i> (B)	Sorts the tasks of set B in descending order by memory size demand
<i>sort_by_price</i> (A)	Sorts the VMs of set A by price
<i>enough_mem</i> (rm_i, m_j)	Returns True, if there is enough memory to schedule t_i to vm_j considering the memory requirement rm_i and the memory capacity m_j ; otherwise returns False
<i>get_cheapest_vm</i> (M^o)	Selects the cheapest on-demand VM
<i>get_wrr_VM</i> (M^s)	Selects the next spot VM using the weighted round-robin heuristic

Table 6.2: Variables and parameters used on Primary Scheduling Module Algorithms

Name	Description
B	Set of tasks
M	Set of VMs that can be used
M^s	Set of VMs spots
M^o	Set of on-demand VMs that can be allocated
D	Application deadline
D_{spot}	Estimated time limit which ensures that there will be enough spare time to migrate tasks of a hibernated VM spot to other VMs no matter when hibernations take place
mkp_w	Worst case makespan (Used only on Algorithm 1)
L	Set with n longest tasks (Used only on Algorithm 1)
$max_ondemand$	Maximum number of on-demand VMs allocated simultaneously
rm_i	Amount of memory required by t_i
e_{ij}	Execution time of a task t_i in the spot vm_j
ovh	The maximum percentage of time overhead induced by checkpoint
$intv_ckp_i$	checkpointing time interval of task t_i
$start_{ij}$	The time that a task t_i will start if it is allocated to a vm_j
α	Time overhead to migrate tasks to a new deployed VM
A	Set of VMs selected to execute the tasks (Used on Algorithm 3)
t_i	Task t_i
vm_s, vm_j, vm_k	Virtual machines

6.2 Dynamic Scheduling Module

In this section, we present the HADS' Dynamic Scheduling Module. Firstly, in Section 6.2.1, we discuss some scheduling concepts used by the module, including the VMs states, Allocation Cycles and the Migration Time Limit estimation. Next, in Section 6.2.4, we introduce the event handler procedure, while Section 6.2.5 presents the migration procedure. Finally, Section 6.2.6 presents the work-stealing procedure, a load balance strategy used to reduce the allocation time of on-demand VMs.

6.2.1 Preliminary Concepts

HADS Dynamic Scheduling Module is an event-driven algorithm that performs some actions in response to events, such as spot VM hibernation, resuming, idleness, etc., that may occur along the application. These actions aim at reducing monetary costs and meeting the application deadline. They may also change the VM state (e.g., from busy

to idle, idle to terminated, etc.). To decide if an idle VM should terminate or not, the algorithm divides its execution time into logical units, denoted Allocation Cycles (AC). Such a concept is presented in Section 6.2.2.

As previously discussed, if a hibernated spot VM does not resume or does it but too late to avoid a temporal failure, the module should execute a migrate procedure. However, in this case, it is also necessary to define a Migration Time Limit ($mtt \in T$) to start executing this procedure. Otherwise, it will be useless to execute it. Hence, in this section, we first explain the concept of allocation cycles, then how mtt is computed (Section 6.2.3).

By considering that a spot vm_j hibernates at time $p \in T$, we define $Q_j = RT_j \cup WT_j$ as the set of tasks that should be migrated if that VM does not resume in time, where $RT_j \in B$ contains the tasks that were running in vm_j at p and WT_j contains tasks that were waiting to be executed in that virtual machine. For instance, in Figure 6.2 vm_j hibernates, does not resume in time, and unfinished tasks, $\{1, 2, 4, 5\}$, should be migrated. In this case, $RT_j = \{1, 4\}$, $WT_j = \{2, 5\}$. The migration procedure starts at mtt , having an overhead of α . In the example, the tasks are migrated to two VMs: i) a new on-demand VM with two cores and ii) an already allocated spot VM with one core. We observe that contrary to task 4, which does not have any checkpoint, task 1 will start executing in the new VM from its last checkpoint.

6.2.2 VM states and Allocation Cycle concept

Let BR , IR , HR , and TR subsets of M be the sets of *busy*, *idle*, *hibernated*, and *terminated* VMs, respectively. We consider that a VM can be in one of the following states i) *busy*, if active and executing tasks ($vm_j \in BR$); ii) *idle*, if active but not executing any task ($vm_j \in IR$); iii) *hibernated*, if it has been hibernated by the cloud provider ($vm_j \in HR$); and iv) *terminated*, if the VM has terminated or it was not available at the beginning of the application execution ($vm_j \in TR$).

To decide if an idle spot VM should terminate or not HADS has introduced an allocation/termination policy. On the one hand, since VMs are charged by second (see Section 4.1), the user has an interest that a VM terminates as soon it becomes *idle*. On the other hand, since already deployed VMs can receive and execute tasks without deploying overheads, it would be interesting that it does not terminate.

Therefore, to be able to decide if a VM should terminate or not, the Dynamic Schedul-

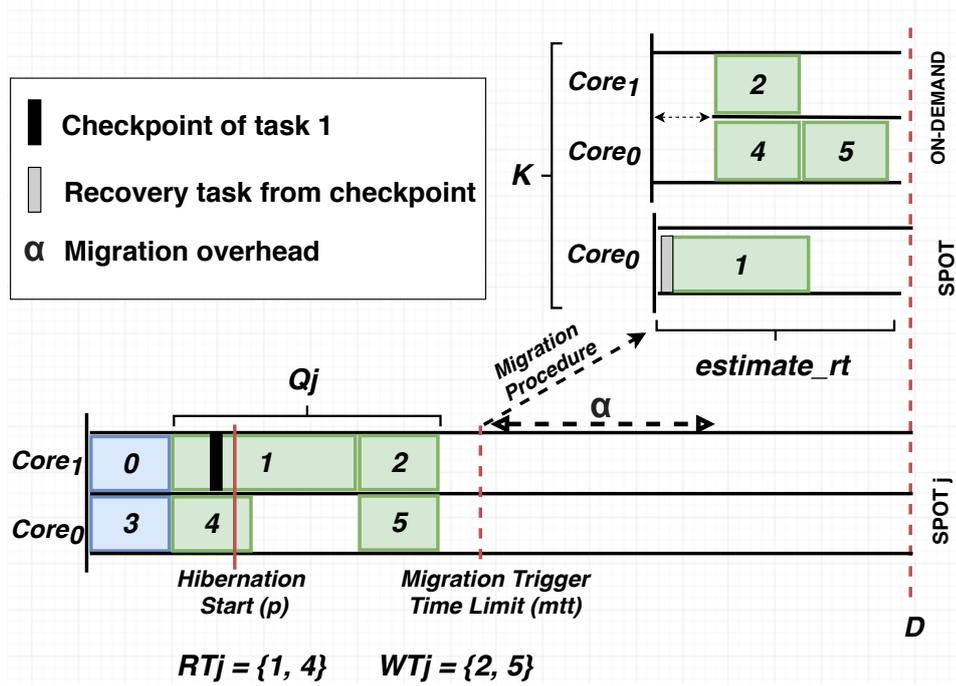


Figure 6.2: Execution where v_j hibernates at p , does not resume, and its tasks are migrated at mtt

ing Module logically divides the VM’s execution time into units denoted Allocation Cycles (ACs), and a vm_j is terminated when it is in the *idle* state and reaches the end of its current AC, denoted AC_{cur_j} . Figure 6.3 shows an example in which the execution of scheduled tasks to a VM requires two ACs (AC_1 and AC_2). In this example, if the VM becomes *idle* and does not receive any new task during $AC_{cur_j} = AC_2$, the VM will terminate at the end of AC_2 .

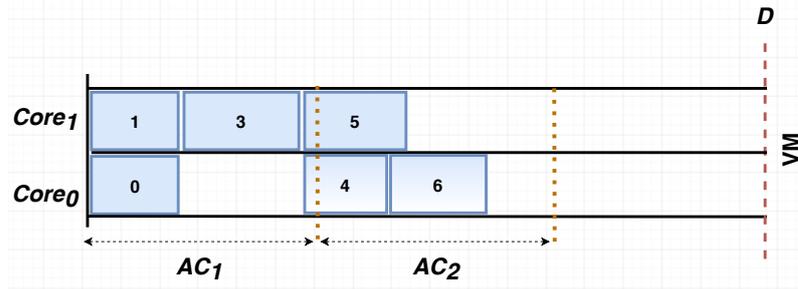


Figure 6.3: A scheduling with two logical Allocation Cycles (AC)

6.2.3 Migration Time Limit (mtt)

Let $K \subset M$ be the set of VMs that will receive the migrated tasks. For instance, in Figure 6.2, K includes a new on-demand VM with two cores and an already allocated spot VM with one core. The function $estimate_rt$ renders the time intervals required to perform

all tasks of Q_j in VMs of K , which is an estimation of the makespan of Q_j 's tasks if they were scheduled to VMs of K .

Algorithm 4 shows the *estimate_rt* algorithm. The algorithm dynamically selects the set of VMs to execute the tasks of Q_j by including them in set K . For each task t_i , by calling the function *check_migration*, it tries to assign t_i to an already allocated vm_k , according to the following order: firstly $vm_k \in K$; if not possible, an idle VM ($vm_k \in IR$); if not possible, a busy VM ($vm_k \in BR$). This order is defined by the function *sort_selected* of line 3 of the algorithm. In each of these sets, their respective VMs are sorted by price in ascending order. The *check_migration* algorithm (Algorithm 7) is described in Section 6.2.5. It basically verifies if vm_k has enough memory for executing t_i and if t_i is migrated to vm_k , the deadline is still respected. On the other hand, if an allocated VM cannot be select, the algorithm selects a new on-demand VM (line 14). Finally, the makespan, considering the assignment of tasks of Q_j to the VMs of K , is estimated (line 20).

In Algorithm 4, the time complexity of the function *check_migration* is $\mathcal{O}(|Q_k|)$ (see Algorithm 7), where $Q_k \subset B$ is composed of the tasks allocated to vm_k that were not finished yet. Moreover, let m' be equal to $|K| + |IR| + |BR|$, where $|K|$ is the size of the set of selected VMs, $|IR|$ is the size of the set of *idle* VMs and $|BR|$ is the size of the set of *busy* VMs. For each task $t_i \in Q_j$, the algorithm executes a for-loop where the function *check_migration* is executed m' times (lines 3 to 9). In the sequence, whenever a task t_i is not assigned to a vm_k in the for-loop of line 3, Algorithm 4 executes a second for-loop (line 11), where the *check_migration* function is executed $|M^o|$ times. Thus, since the complexity of function *assign* (executed in lines 5 and 13) is $\mathcal{O}(c)$, complexity of Algorithm 4 in the worst case is $\mathcal{O}(|Q_j| \times (|Q_k|(m' + |M^o|)))$. In addition, since sets Q_j , Q_k , IR , BR and M^o are represented by lists of integers and as the number of tasks is usually greater than the number of VMs, space complexity of Algorithm 4 is $\mathcal{O}(|Q_j| + |Q_k|)$.

Equation 6.4 expresses the migration time limit ($mtt \in T$) that defines when the migration procedure must be triggered; otherwise, it will not be possible to meet the application deadline D and, therefore, a temporal failure will take place. We consider that a new deployed VM takes α time intervals to receive the migrated tasks.

$$mtt = D - (estimate_rt(Q_j, IR, BR, M^o, D) + \alpha) \quad (6.4)$$

Algorithm 4 *estimate_rt***Input:** Q_j, IR, BR, M^o and D

```

1:  $K \leftarrow \emptyset$ 
2: for  $t_i \in Q_j$  do
3:   for  $vm_k \in \text{sort\_selected}(K, IR, BR)$  do
4:     if  $\text{check\_migration}(t_i, vm_k, D)$  then
5:        $\text{assign}(t_i, vm_k)$ 
6:        $K \leftarrow K \cup \{vm_k\}$ 
7:       break {Next task}
8:     end if
9:   end for
10:  if task was not assign then
11:    for  $vm_k \in M^o$  do
12:      if  $\text{check\_migration}(t_i, vm_k, D)$  then
13:         $\text{assign}(t_i, vm_k)$ 
14:         $K \leftarrow K \cup \{vm_k\}$ 
15:        break {Next task}
16:      end if
17:    end for
18:  end if
19: end for
20: return  $\text{get\_makespan}(K)$ 

```

6.2.4 Event Handler

The events considered by the Dynamic Scheduling Module are handled by Algorithm 5. Those events and the respective actions taken by the module are present in Figure 6.4 and described next.

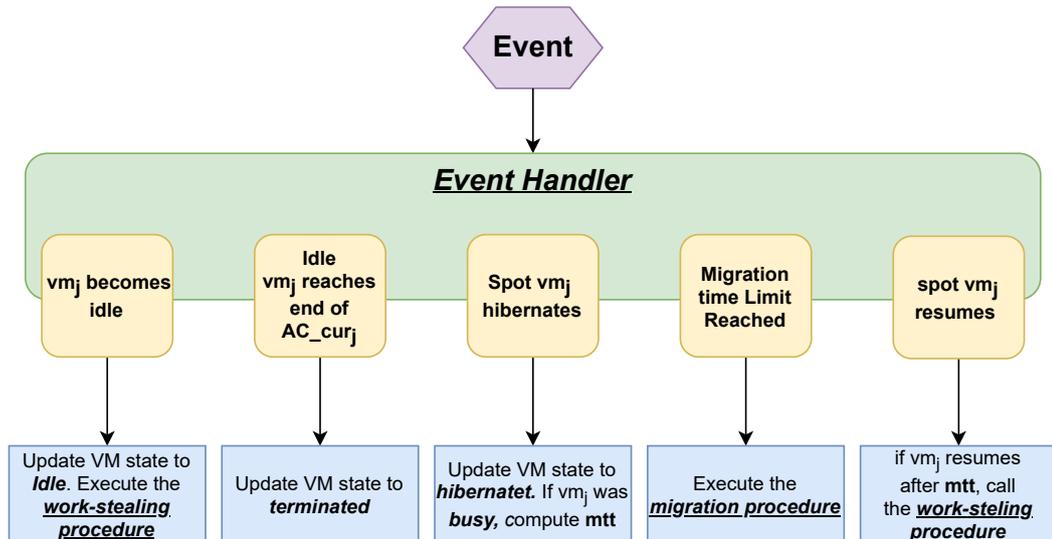


Figure 6.4: Diagram with the events and actions handle by the Dynamic Scheduling Module

Figure 6.5 shows the state diagram of spot vm_j and the transition between them according to those events. As can be seen, a *busy* or *idle* spot VM can go to hibernate

if the provider hibernates it. Moreover, when hibernated, a spot VM can go to the idle or busy state if the VM resumes before the mtt value or after it, respectively. We also see in Figure 6.5 that a busy VM goes from busy to idle when it finishes all its task ($Q_j = \emptyset$), and an idle VM can go back to the busy state when it receives tasks from the work-stealing procedure (Section 6.2.6) or due to the migration procedure (Section 6.2.5). Finally, an idle VM is terminated when it remains idle and reach the ends of its current AC .

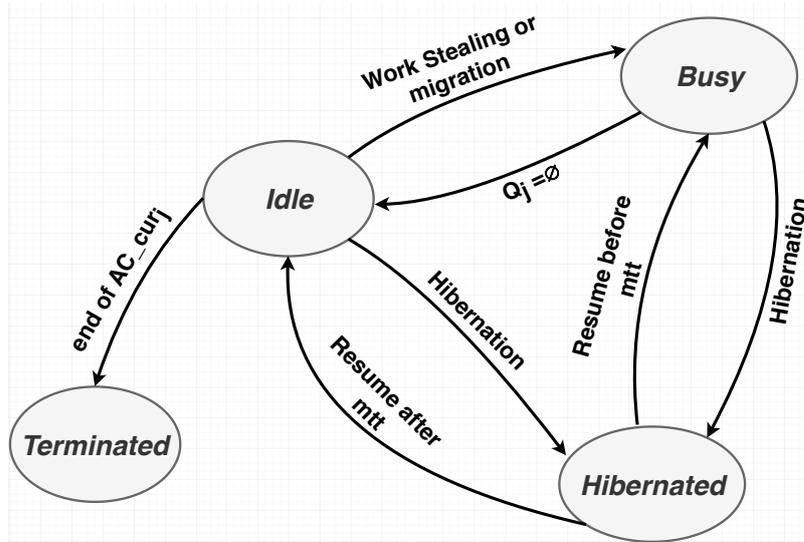


Figure 6.5: Spot vm_j state diagram

vm_j becomes idle (line 2)

Upon finishing to execute all tasks scheduled to it ($Q_j = \emptyset$), vm_j changes its state from *busy* to *idle* (lines 3 and 4) and the work stealing procedure is executed (line 5) since the current AC has not ended. Note that if, due to the execution of this procedure, vm_j receives new tasks to execute, its state will be changed to *busy* again. More details about the work stealing procedure are presented in Section 6.2.6.

Idle vm_j reaches the end of its current AC (line 6)

As previously explained, if after the end of the current AC (AC_{cur_j}), vm_j is *idle* and has no new tasks to execute, i.e., $Q_j = \emptyset$, it will be terminated. In this case, vm_j is removed from the set IR of *idle* VMs and included in the set TR of *terminated* VMs (lines 7 and 8).

Spot vm_j hibernates (line 9)

The cloud provider can hibernate either *busy* or *idle* spot VMs. In both cases, the algorithm changes the vm_j 's state to *hibernated* updating the corresponding sets (lines

12, 14, and 16).

Furthermore, if vm_j was *busy*, it also computes the migration time limit mtt (Equation 6.4), considering the unfinished tasks Q_j of vm_j (line 11). To this end, the algorithm compute the time required to execute the tasks by calling $estimate_rt(Q_j)$, as explaining in Section 6.2.3. If the vm_j was *idle*, the algorithm just changes its state to *hibernated* (lines 12 and 16).

Migration time limit reached (line 17)

Whenever vm_j is *busy* and the migration time limit mtt is reached, in order to satisfy the application deadline D , all unfinished tasks (Q_j) of vm_j should be migrated to other VMs. For this purpose, the algorithm calls the migration procedure (line 18), which is described in details in Section 6.2.5.

Spot vm_j resumes (line 19)

The spot vm_j may resume before the migration time limit mtt or not. In both cases, it is excluded from the set HR of hibernated VMs (line 20). As previously discussed, if a VM resumes before the time limit, the VM turns its state to *busy* (line 22), and the tasks scheduled to it continue their execution from their respective break-point, and no additional action is necessary. However, when a VM resumes after mtt , the event *Migration time limit reached* already happened, and, consequently, the tasks of this VM were already migrated to other VMs. Hence, in this case, the algorithm executes a work-stealing procedure (line 24) that tries to move tasks from *busy* VMs to vm_j . This procedure is described in Section 6.2.6.

Algorithm 5 *Event Handler*

Input: $event, vm_j, Q_j, IR, BR, TR, HR, M^o, \alpha,$ and D

- 1: **switch** ($event$)
- 2: **case** $Q_j = \emptyset$:
- 3: $BR \leftarrow BR \setminus \{vm_j\}$
- 4: $IR \leftarrow IR \cup \{vm_j\}$
- 5: $work_stealing_procedure(vm_j, BR)$ {/*Algorithm 8*/}
- 6: **case** $vm_j \in IR$ **and** AC_cur_j **ended** :
- 7: $IR \leftarrow IR \setminus \{vm_j\}$
- 8: $TR \leftarrow TR \cup \{vm_j\}$
- 9: **case** vm_j *hibernates*:
- 10: **if** $vm_j \in BR$ **then**
- 11: $mtt \leftarrow D - (estimate_rt(Q_j, IR, BR, M^o, D) + \alpha)$
- 12: $BR \leftarrow BR \setminus \{vm_j\}$
- 13: **else**
- 14: $IR \leftarrow IR \setminus \{vm_j\}$
- 15: **end if**
- 16: $HR \leftarrow HR \cup \{vm_j\}$
- 17: **case** mtt *reached*:
- 18: $migration_procedure(Q_j, D, IR, BR, M^o)$ {/*Algorithm 6*/}
- 19: **case** vm_j *resumes* :
- 20: $HR \leftarrow HR \setminus \{vm_j\}$
- 21: **if** *resumes before mtt* **then**
- 22: $BR \leftarrow BR \cup \{vm_j\}$
- 23: **else**
- 24: $work_stealing_procedure(BR, IR, vm_j, D)$ {/*Algorithm 8*/}
- 25: **end if**
- 26: **end switch**

6.2.5 Migration Procedure

The migration procedure is presented in Algorithm 6. It receives as input the set of tasks to be migrated (Q_l), the deadline D , the set of *idle* and *busy* VMs (IR and BR , respectively) and the set of on-demand VMs that can be allocated (M^o).

Firstly, for each task t_i in Q_l , the algorithm tries to schedule t_i to one of the *idle* VMs of set IR (lines 4 to 12). If it is successful, the algorithm updates the state of the selected VM by removing it from set IR and included it in the set BR (lines 8 and 9). Otherwise, the algorithm tries to schedule t_i to one of the *busy* VMs of set BR (lines 16 to 22). If not possible, the task is scheduled to a new on-demand VM of set M^o (lines 27 to 35). In this case, it is only necessary to consider the start period of t_i in vm_j ($start_{ij}$),

the corresponding execution time (e_{ij}), and the overhead α in order to avoid deadline violation (line 28). The new allocated on-demand VM is then removed from set M^o and included in set BR (lines 31 and 32).

It is worth pointing out that if $t_i \in RT_i$, i.e., t_i was running when the hibernation happened, it will start its execution in the selected target VM from the last recorded checkpoint, and only the remaining execution time of the task will be considered in the migration procedure. In addition, if the target VM is a spot one, by $intv_chk_i$, provided by the Primary Scheduled module for t_i , it knows the time interval with which the new checkpoints of t_i must be taken.

This approach tries to minimize monetary costs, since VMs are allocated by AC units and the available time of current allocated ACs of *idle* and *busy* VMs are requested by the migration procedure, whenever possible. A final remark is that, in the case of both *idle* (line 3) and *busy* VMs (line 15), the algorithm gives priority to spot VMs rather than on demand ones, which also reduces monetary costs.

In order to migrate a task t_i to a spot vm_j , Algorithm 6 calls the procedure *check_migration* of Algorithm 7. Task t_i can be migrated to vm_j provided that the latter, after scheduling t_i , has enough spare time for executing a migration procedure of its longest scheduled task, since vm_j itself can hibernate just before finishing executing this longest task. Among both the running and waiting scheduled tasks of vm_j (Q_j) and t_i , the call *get_longest_Tasks*(1, $Q_j \cup \{t_i\}$) renders a set with the longest task t_m (line 1). The algorithm then computes eft_j , the expected finishing time of vm_j considering the migration of t_i to vm_j (line 2). Finally, the function returns True if the two conditions of line 3 hold: (1) the difference between D and eft_j is greater than e_{mj} , the execution time of the longest task t_m , plus α ; (2) there exists enough memory in vm_j to execute t_i ; otherwise, it returns False. Figure 6.6 shows a scenario where condition (1) is satisfied. Note that task 6 is the longest one.

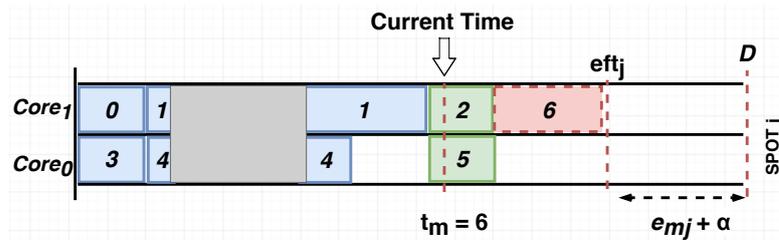


Figure 6.6: Evaluating the migration of task 6 to a spot VM

We should emphasize that it is crucial to leave a spare time in vm_j between the end of

the execution of vm_j tasks and the application deadline D , which is equal to the execution time of the longest task because vm_j is also subject to hibernation. If it occurs along the execution of tasks of Q_j or t_i , the scheduling will wait for a resuming event until mtt , whose value has been computed by Equation 6.4. However, if vm_j hibernates and does not resume before mtt , there always exists on-demand VMs with the same characteristics of vm_j that can be allocated. Thus, the spare time reservation guarantee that all tasks will have enough time to migrate and execute, respecting the application's deadline D , no matter when hibernation events take place. Figure 6.7 illustrates this last case, where a hibernation occurs just before the end of task 6, which is, therefore, migrated to a new on-demand VM.

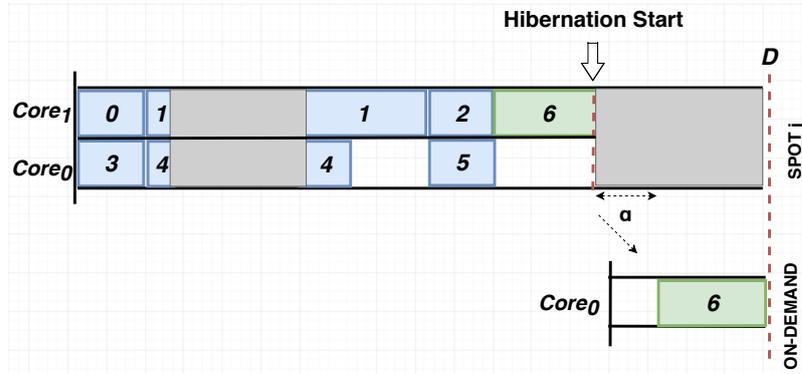


Figure 6.7: Migration of task 6 to a new on-demand VM

The migration procedure executes merge sort procedures whose complexities are $\mathcal{O}(|IR|\log(|IR|))$, $\mathcal{O}(|BR|\log(|BR|))$ and $\mathcal{O}(|M^o|\log(|M^o|))$ (lines 3, 15 and 26). Moreover, in attempts 1 and 2, the algorithm executes the function *check_migration* (Algorithm 7) that, as presented before, has complexity $\mathcal{O}(Q_j)$. Thus, in attempt 1 the complexity is $\mathcal{O}(|IR|(\log|IR| + Q_j))$, since the algorithm *check_migration* is executed $|IR|$ times, where $|IR|$ is the size of the set of *idle* VMs. In attempt 2 the complexity is $\mathcal{O}(|BR|(\log|BR| + Q_j))$, since the algorithm *check_migration* is executed $|BR|$ times, where $|BR|$ is the size of the set of *busy* VMs. Finally, since in attempt 3 only the merge sort is executed, the complexity is $\mathcal{O}(|M^o|\log|M^o|)$, which is the time spent to sort the set of on-demand VMs M^o .

In the worst case, each attempt is executed $|Q_l|$ times, where $|Q_l|$ is the number of tasks that will be migrated. Thus, the complexity of Algorithm 6 is:

$$\begin{aligned} \mathcal{O}(|Q_l| \times [(&|IR| \log(|IR|) + |Q_j|) + \\ &(|BR| \log(|BR|) + |Q_j|) + \\ &(|M^o| \log(|M^o|))]) \end{aligned}$$

However, since $|BR| + |IR| + |M^o| < |M|$ the complexity of Algorithm 6 is $\mathcal{O}(|Q_l| \times (|M| \log |M| + |Q_j|))$. In terms of space complexity, Algorithm 6 is $\mathcal{O}(|Q_l|)$.

Algorithm 6 *Migration Procedure*

Input: Q_l , D , IR , BR , and M^o

```

1: for each  $t_i \in Q_l$  do
2:   { /*Attempt 1*/ }
3:   sort_by_market( $IR$ ) { /* Prioritizes spot VMs */ }
4:   for each  $vm_j \in IR$  do
5:     { /* Call Algorithm 7 */ }
6:     if check_migration( $t_i, vm_j, D$ ) then
7:       migrate( $t_i, vm_j$ )
8:        $IR \leftarrow IR \setminus \{vm_j\}$ 
9:        $BR \leftarrow BR \cup \{vm_j\}$ 
10:      break {Migrate next task}
11:    end if
12:  end for
13:  if not migrated then
14:    { /*Attempt 2*/ }
15:    sort_by_market( $BR$ ) { /* Prioritizes spot VMs */ }
16:    for each  $vm_j \in BR$  do
17:      { /* Call Algorithm 7 */ }
18:      if check_migration( $t_i, vm_j, D$ ) then
19:        migrate( $t_i, vm_j$ )
20:        break {Migrate next task}
21:      end if
22:    end for
23:  end if
24:  if not migrated then
25:    { /*Attempt 3*/ }
26:    sort_by_price( $M^o$ )
27:    for each  $vm_j \in M^o$  do
28:      if  $start_{ij} + e_{ij} + \alpha < D$  then
29:        start_vm( $vm_j$ )
30:        migrate( $t_i, vm_j$ )
31:         $M^o \leftarrow M^o \setminus \{vm_j\}$ 
32:         $BR \leftarrow BR \cup \{vm_j\}$ 
33:        break {Migrate next task}
34:      end if
35:    end for
36:  end if
37: end for

```

Algorithm 7 *check_migration*

Input: t_i , vm_j , and D

- 1: $tm \leftarrow (tk \in get_longest_Tasks(1, Q_j \cup \{t_i\}))$
- 2: $eft_j \leftarrow compute_eft(vm_j, t_i)$
- 3: **if** $D - eft_j > e_{mj} + \alpha$ **and** $enough_mem(rm_i, m_j)$ **then**
- 4: return **True**
- 5: **else**
- 6: return **False**
- 7: **end if**

6.2.6 Work-Stealing Procedure

For monetary costs sake, the work-stealing procedure, presented in Algorithm 8, aims at reducing the allocation time of on-demand VMs as well as balancing the load of spot VMs. It is triggered whenever the spot vm_j : i) resumes after the migration time limit mtt has been reached (line 24 of Algorithm 5) or ii) VM becomes *idle* after the execution of its scheduled tasks ($Q_j = \emptyset$) and the current AC of the VM has not ended (line 5 of Algorithm 5). Basically, the algorithm tries to migrate tasks from both on-demand and busy spot VMs to the idle VM in question.

Algorithm 8 *Work-Stealing Procedure*

Input: BR , IR , vm_k , D

- 1: $sort_by_market(BR)$ {/* Prioritizes on-demand VMs */}
- 2: **for each** $vm_j \in BR$ **do**
- 3: $S \leftarrow selectStolenTasks(WT_j)$ {/* get tasks that can be stolen */}
- 4: **for each** $t_i \in S$ **do**
- 5: { /* Call Algorithm 7 */ }
- 6: **if** $check_migration(t_i, vm_k, D)$ **then**
- 7: $migrate(t_i, vm_k)$
- 8: **end if**
- 9: **end for**
- 10: **end for**
- 11: **if at least one task was stolen then**
- 12: $BR \leftarrow BR \cup \{vm_k\}$
- 13: $IR \leftarrow IR \setminus \{vm_k\}$
- 14: **end if**

For each *busy* $vm_j \in BR$ the procedure selects the tasks that can be stolen from it (line 3) and tries to migrate them to the *idle* vm_k (lines 4-9). Since on-demand VMs are more expensive than spot VMs, the procedure considers firstly the tasks of the former (line 1).

Note that the working stealing procedure is applied only to the waiting tasks WT_j of vm_j and not to those that were executing. Figure 6.8 shows an example where tasks are spread over three ACs (AC_1 , AC_2 , and AC_3) of a VM. Since the current Allocation

Cycle $AC_{cur} = AC_1$, tasks 5, 6, 7, and 8 are candidates to be stolen and start in the next cycles.

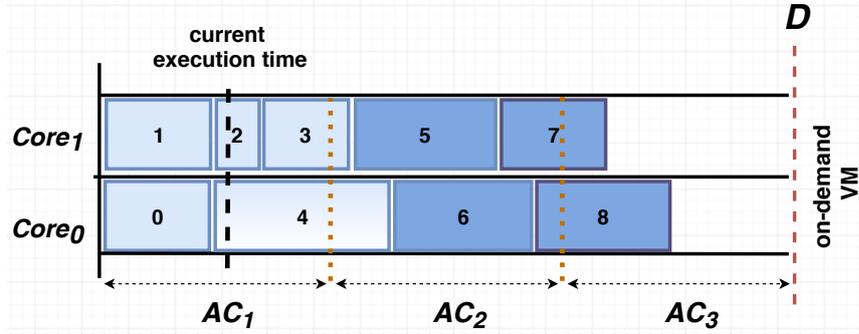


Figure 6.8: Example of tasks in an on-demand VM that can be stolen by the work-stealing procedure

Similarly to the migration procedure, by calling the function *check_migration* of Algorithm 7 for each selected task, the work-stealing procedure also verifies if the task migration would result in deadline violation (line 6). If it is not the case, the task is migrated (line 7). Finally, if at least one task has been migrated to the *idle* spot vm_k , its state changes to *busy* and, therefore, it is included in the set of *busy* VMs and removed from the set of *idle* ones (lines 12 and 13).

Note that, whenever the work-stealing procedure is executed, it sorts the set BR of *busy* VMs with a sort algorithm of complexity $\mathcal{O}(|BR|\log(|BR|))$ (line 1). In the sequence, for each $vm_j \in B$, the function *selectStolenTaks*, whose complexity is $\mathcal{O}(|WT_j|)$, is called. Moreover, in lines 4 to 9, the *check_migration* procedure is called $|S|$ times, where $|S|$ is the number of tasks selected to be stolen. Thus, the complexity of Algorithm 8 is $\mathcal{O}(|BR| \times (\log(|BR|) + |WT_j| + |S||Q_k|))$, where $\mathcal{O}(|Q_k|)$ is the complexity of the *check_migration* procedure. Since the number of tasks is greater than the number of VMs, we have that $WT_j > \log(|BR|)$ and $|S||Q_k| > \log(|BR|)$. Thus, the complexity of Algorithm 8 is $\mathcal{O}(|BR| \times (|WT_j| + |S||Q_k|))$. Besides, as the sets WT_j , S , Q_k , BR , and IR are represented by lists of integers and $|WT_j| \geq |S|$, the space complexity of Algorithm 8 is $\mathcal{O}(|WT_j| + |Q_k|)$.

All variables and parameters used by HADS' Dynamic Scheduling Module are presented in Table 6.3. The functions and procedures called by the algorithms are summarized in Table 6.4.

Table 6.3: Variables and parameters used by the Dynamic Scheduling Module algorithms

Name	Description
B	Set of tasks
M	Set of VMs that can be used
M^s	Set of VMs spots
M^o	Set of on-demand VMs that can be allocated
IR	Set of <i>idle</i> VMs
BR	Set of <i>busy</i> VMs
TR	Set of <i>terminated</i> VMs
HR	Set of <i>hibernated</i> VMs
K	Set of selected VMs
$R = IR \cup BR \cup K$	Set of VMs that are available during the migration
Q_j	Set of unfinished tasks of a vm_j
WT_j	Set of tasks that are waiting to be executed in vm_j
S	The set of tasks that can be stolen from vm_j
$event$	An event that trigger an action in the Algorithm
D	Application deadline
e_{ik}	Execution time of a task t_i in the spot vm_k
mtt	Migration time limit
$intv_ckp_i$	Checkpointing time interval of task t_i
α	Time intervals to migrate tasks to a new deployed VM

Table 6.4: Functions and Procedures called by the Dynamic Scheduling Module algorithms

Name	Description
$get_longest_tasks(n, S)$	Renders the n longest tasks of set S
$get_slowest_vm(M)$	Renders the slowest VM of M
$enough_mem(rm_i, m_j)$	Returns True, if there is enough memory to schedule t_i to vm_j considering the memory requirement rm_i and the memory capacity m_j ; otherwise returns False
$get_cheapest_vm(M^o)$	Selects the cheapest on-demand VM
$schedule(t_i, vm_j, intv_ckp_i)$	Schedules task t_i to a core of vm_j . $intv_ckp_i$ informs, in case of a spot VM, the time interval between two consecutive checkpoints
$sort_selected(K, IR, BR)$	sorts the VMs of K , IR , BR , according to the order: (1) VMs of K , (2)VMs of IR , and (3)VMs of BR . For each of these sets, the VMs are sorted by price
$get_makespan(K)$	Renders the makespan as if the tasks of Q_j had been scheduled to the VMs of K
$selectStolenTasks(WT_j)$	Returns all tasks that can be stolen from a <i>busy</i> vm_j . Receive as input the set of waiting tasks WT_j that are waiting to be executed in vm_j
$migrate(t_i, vm_k)$	Migrates task t_i to one of the virtual cores of the spot vm_k

Chapter 7

Experimental Results of HADS

This Chapter presents the experimental evaluation conducted with HADS. Firstly, In Section 7.1 we present the experimental environment, including the used BoT applications, VMs, the generated hibernated and resume scenarios, and all input parameters used in the tests. Next, in Section 7.2, we define two execution baseline cases: i) *spot VMs without hibernation*, where HADS execute all tasks without any spot hibernation; and ii) *on-demand only*, where only on-demand VMs are used. Then, in Section 7.3, we discuss the experiments conducted on Amazon EC2.

7.1 Experimental Environment

All results presented in this chapter were obtained from real executions using VMs from EC2. According with EC2 only the VMs of families c3, c4, c5, m4, m5, r3, and r4 with less than 100 GB of memory, running in the spot market, are hibernation-prone. Therefore, in the experiments, we have selected spot VMs of the families c3 and c4 which provide good computation power and have high availability in the spot market ¹. Table 7.1 shows the computational characteristics of VMs that we used in our experiments as well as its respective prices in on-demand and spot markets. Note that all results presented in this chapter were published in [75]. Therefore, all presented VMs' prices were obtained in December 2019.

¹<https://aws.amazon.com/ec2/spot/instance-advisor/>

Table 7.1: VMs attributes

Type	#VCPUs	Memory	Gflops	On-demand price	Spot price
				per hour	per hour
c3.large	2	3.75 GB	22.09	0.105\$	0.0294\$
c4.large	2	3.75 GB	40.73	0.100\$	0.0308\$
c3.xlarge	4	7.50 GB	44.46	0.210\$	0.0596\$
c4.xlarge	4	7.50 GB	83.33	0.199\$	0.0673\$

Besides the VMs, to evaluate HADS we use the following BoT applications:

Synthetic: In this case, the jobs are composed by tasks generated with the application template proposed by Alves et al. [3] which is based on vector operations and whose execution time depends on the size of the vectors. We thus created several synthetic tasks, each one with memory footprint between 2.81MB and 13.19 MB, resulting in execution times which vary from 1:42 to 5:30 minutes. Then, we conceived three BoT applications, J60, J80, and J100, by randomly selecting those tasks.

NAS benchmark: It concerns the ED application, a real embarrassingly distributed application, offered on the GRIDNBP 3.1 suite of NAS benchmark [20]. By executing ED, we created the job ED200 which is composed of 200 tasks running the largest problem size (class B).

Table 7.2 shows the characteristics of the four BoT jobs, including their respective number of tasks, memory footprint, and runtime.

Table 7.2: Jobs characteristics

job	# tasks	runtime (minutes)			memory footprint		
		min	avg	max	min	avg	max
J60	60	01:42	03:18	05:23	2.85MB	4.69MB	12.20MB
J80	80	01:43	03:19	05:22	2.91MB	4.71MB	13.19MB
J100	100	01:47	03:10	05:30	2.81MB	4.49MB	10.86MB
ED200	200	02:41	03:31	05:54	153.74MB	168.68MB	177.77MB

To store checkpoint files in those tests, we use the S3 service. That service was chosen due to its economic advantage and because the memory footprint of the executed applications does not cause a huge overhead in terms of dump time, i.e., the time it takes to record a checkpoint. Moreover, EBS and EFS were not used for the following reasons. EBS service is a local storage system that prevents tasks sharing the same EBS

volume from being spread among different VMs in the recovery process. Furthermore, in the recovery process, EBS presents an additional overhead caused by the necessity of associating the storage volume containing the checkpoint with the VM, where the recovered task will execute. In the case of EFS, the service presented checkpointing and recovery times close to EBS but with higher monetary costs than the other services. Also, the EFS performance showed a significant degradation with concurrent checkpoint records. All results related to the storage services are presented in detail in Chapter 5.

7.1.1 Emulation of the Hibernation and Resume Events

Cloud users have no control over spot VMs hibernations since the cloud provider decides when to hibernate and resume a given spot VM according to resource demands variation. Thus, to evaluate different patterns of spot VMs hibernation and resuming, we have emulated the hibernation feature by using Poisson distribution [1] to generated the hibernation and resuming events for each type of spot VM.

Since in Amazon EC2, when a spot VM of a given family type hibernates, other VMs of the same type will probably hibernate too, we emulate events for groups of VMs of identical types. In other words, when an event happens for a VM, it has an impact not only on this VM but also on all VMs of that type. Thus, we use distinct Poisson functions for modeling the events, which allows the creation of scenarios where *hibernating* and *resuming* events have different probability mass functions defined by the parameters λ_h and λ_r , respectively.

Whenever an emulated hibernation event occurs, the spot VM state is saved by using the checkpoint tool CRIU[29], and all tasks allocated to it are paused. Hence, if the VM resumes later, those tasks can be recovered and continue their execution. Note that, although the hibernation event is emulated, the feature of the hibernation event was preserved, i.e., all tasks are recovered from the break-point when a hibernated spot VM resumes. The λ values used to evaluate HADS are presented next.

7.1.2 Parameters Setting and Generated Scenarios

In order to evaluate HADS, we firstly generated different scenarios. To this end, we considered allocation cycles of 15 minutes ($AC = 900s$) and, based on empirical experiments, task migration overhead equals to 3 minutes ($\alpha = 180s$). In addition, the sets M^s and M^o were built considering the allocation constraints adopted by Amazon EC2 in December

2019, which means that up to five VMs of each type in each market could be allocated. The checkpoint overhead ovh , was set to 10%. For all jobs, the execution deadline was 35 minutes ($D = 2100s$).

Let the λ parameter of Poisson distribution be the number of expected events divided by a time interval. Since the application execution is discretized by time interval and D is the application deadline, if we respectively define k_h and k_r , as the expected number (rate) of hibernating and resuming events during the application execution, λ_h and λ_r parameters are given by $\lambda_h = k_h/D$ and $\lambda_r = k_r/D$. We should point that, since we are considering scenarios where multiple events can happen, the actual number of hibernation (resp., resuming) events that occur in an experiment might be greater than k_h (resp., k_r), as shown in Table 7.5, discussed later. Table 7.3 presents seven different scenarios by varying k_h and k_r .

Table 7.3: Different execution scenarios generated by varying parameters λ_h and λ_r .

ID	<i>hibernating</i>	<i>resuming</i>	λ_h	λ_r
sc_1	$k_h = 1$	$k_r = 0$	$1/2100$	$0/2100$
sc_2	$k_h = 5$	$k_r = 0$	$5/2100$	$0/2100$
sc_3	$k_h = 1$	$k_r = 5$	$1/2100$	$5/2100$
sc_4	$k_h = 5$	$k_r = 5$	$5/2100$	$5/2100$
sc_5	$k_h = 3$	$k_r = 2.5$	$3/2100$	$2.5/2100$
sc_6	$k_h = 2$	$k_r = 1$	$2/2100$	$1/2100$
sc_7	$k_h = 2$	$k_r = 2$	$2/2100$	$2/2100$

Figure 7.1 shows the average duration of a hibernation in each scenario. As expected, scenarios sc_1 and sc_2 present the longest hibernation time duration (25:20 and 31:18 minutes, respectively) since, in these scenarios, the resuming event does not occur ($k_r = 0$). The smallest times are observed in sc_3 and sc_4 (14:07 and 15:37 minutes, respectively) because they have the highest rate of resuming events ($k_h = 5$), which reduces the average duration of hibernation. On the other hand, in scenario sc_5 where $k_r = 2.5$, the hibernation time (20:12 minutes) is longer than in sc_6 and sc_7 (17:30 and 16:41 minutes, respectively) since, in those cases, the expect number of hibernation events is $k_h = 2$ while in scenario sc_5 , it is $k_h = 3$.

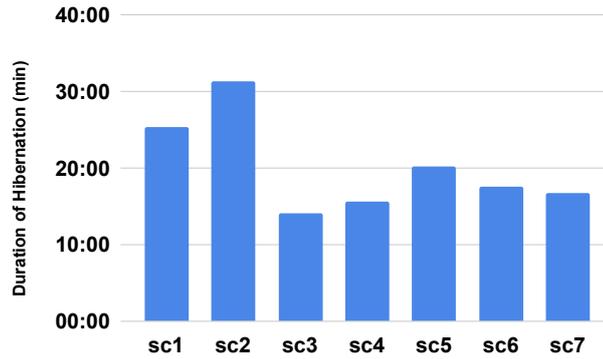


Figure 7.1: Average duration of hibernation in different scenarios

7.2 Baseline Executions

We have considered two baseline cases: i) *spot VMs without hibernation*, which is the case where the initial scheduling defined by the Algorithm 3 is followed without the need of migration; and ii) *on-demand only*, which uses the same scheduling, but only with on-demand VMs. Table 7.4 presents the average costs of executing the four evaluated jobs (J60, J80, J100 and ED200) on both baseline cases. It also contains the type and number of used VMs, the average makespan in minutes, and the percentage difference between their execution costs (diff).

Note that, because the scheduling is the same in both cases, except for the market, the cost difference is around 66.33% to 76.2%, which is close to the difference in the price between the used spots and on-demand VMs (see Table 7.1). Note also that, in Table 7.4 the jobs makespan are below the 35 minutes of the jobs deadline. This occurs because Algorithm 3 respects the D_{spot} limit presented in Section 6.1.1.

Table 7.4: Baseline executions

job	#VMs	makespan	spot without hibernation	on-demand only	diff
J60 (6 VMs)	2-c3.large 2-c4.large 2-c4.xlarge	20:08	\$0.08	\$0.32	76.25%
J80 (8 VMs)	2-c3.large 1-c3.xlarge 3-c4.large 2-c4.xlarge	19:49	\$0.10	\$0.37	72.97%
J100 (10 VMs)	2-c3.large 1-c3.xlarge 4-c4.large 3-c4.xlarge	18:43	\$0.13	\$0.43	70.78%
ED200 (16 VMs)	5-c3.large 2-c3.xlarge 5-c4.large 4-c4.xlarge	31:27	\$0.33	\$0.98	66.33%

7.3 Performance Results

Table 7.5 presents the performance results related to the execution of jobs J60, J80, J100, and ED200 in each of the seven scenarios. It presents the average number of hibernations, the number of used on-demand VMs in executions where hibernation took place, followed by the corresponding average values of both the makespan and monetary costs. The percentage difference between the latter and the on-demand VMs baseline is presented in the last column (diff). Note that the makespan of the four jobs is less than the 35 minutes of the deadline, which confirms the effectiveness of Algorithm 3 in respecting applications deadline.

Figures 7.2 and 7.3 show the percentage of time that a procedure is executed in respect to the total number of hibernations occurred along with the job execution. For instance, if four hibernations occurred and in two of them the on-demand migration procedure took place, there will be a bar with 50%. In the Figures *on-demand*, *idle*, and *busy* migrations mean the type of state of the VMs to which a task is migrated according to Algorithm 6.

We can observe in Table 7.5 that, when compared to the on-demand baseline, HADS presents cost reductions in all cases which vary from 19.79% to 72.92%.

For all jobs, the worst results in terms of monetary cost are those for scenario sc_2 . Such a result is expected since sc_2 has no VMs resuming ($k_r = 0$) and has the highest hibernation rate ($k_h = 5$). Therefore, in this case, it is always necessary to allocate many on-demand VMs throughout the execution to avoid temporal failures. Since there is no resuming of spot VMs in this scenario, hibernated VMs will always reach the migration time limit (mtt). Figures 7.2 and 7.3 show that in sc_2 the migration to on-demand VMs occurs in more than 50% of the hibernation cases. The impact of the hibernation is only mitigated by migrations of tasks to busy and idle VMs.

Nevertheless, it is worth pointing out that, although there is no possibility of resuming in scenario sc_1 either, the cost is reduced in more than 50% for almost all jobs, excepted for J80 where the reduction was 46.87%. Such a reduction happens because, in this scenario, the number of hibernations is low ($k_h = 1$), i.e., in general, less than half of the spot VMs hibernate. Thus, the tasks are migrated to busy or idle VMs instead of new allocated on-demand VMs. That can be seen in Figures 7.2 and 7.3, where migrations to busy VMs happen in more than 50% of the hibernation cases.

Scenario sc_3 presents the best results in terms of cost reduction (more than 60% for all jobs) because it has the lowest hibernation rate ($k_h = 1$) and the highest resuming

Table 7.5: Execution of HADS in scenarios sc_1 to sc_7 . The table shows the probabilistic mass function of the hibernation (λ_h) and the resume events (λ_r) for each scenario, the average number of hibernations, the number of used on-demand VMs, the average makespan, and the average monetary cost

Jobs	scenario	λ_h	λ_r	# hibernation	# on-demand	makespan (min)	cost	diff
J60 (6 VMs spots)	sc_1	1/2100	0/2100	1.33	1.33	25:13	\$0.146	54.52%
	sc_2	5/2100	0/2100	4.33	3.33	34:24	\$0.256	19.79%
	sc_3	1/2100	5/2100	1.67	1.0	24:39	\$0.087	72.92%
	sc_4	5/2100	5/2100	2.02	1.33	30:40	\$0.145	54.69%
	sc_5	3/2100	2.5/2100	2.00	0.67	24:31	\$0.090	71.77%
	sc_6	2/2100	1/2100	2.00	1.00	30:45	\$0.097	69.79%
	sc_7	2/2100	2/2100	2.00	1.67	32:02	\$0.093	70.94%
J80 (8 VMs spots)	sc_1	1/2100	0/2100	2.57	1.0	27:11	\$0.197	46.82%
	sc_2	5/2100	0/2100	6.33	4.67	34:56	\$0.284	23.12%
	sc_3	1/2100	5/2100	2.67	1.30	31:53	\$0.117	68.47%
	sc_4	5/2100	5/2100	4.00	2.33	32:41	\$0.140	62.09%
	sc_5	3/2100	2.5/2100	4.33	3.33	33:26	\$0.213	42.34%
	sc_6	2/2100	1/2100	2.67	1.33	26:58	\$0.153	58.56%
	sc_7	2/2100	2/2100	2.67	1.33	29:13	\$0.123	66.67%
J100 (10 VMs spots)	sc_1	1/2100	0/2100	2.33	0.67	26:42	\$0.167	61.77%
	sc_2	5/2100	0/2100	7.67	3.67	30:14	\$0.302	30.66%
	sc_3	1/2100	5/2100	1.33	1.00	26:08	\$0.150	65.61%
	sc_4	5/2100	5/2100	3.40	1.89	34:12	\$0.189	56.64%
	sc_5	3/2100	2.5/2100	3.00	2.70	32:59	\$0.223	48.78%
	sc_6	2/2100	1/2100	4.67	2.00	28:54	\$0.177	59.48%
	sc_7	2/2100	2/2100	3.67	1.33	32:31	\$0.160	63.30%
ED200 (16 VMs spots)	sc_1	1/2100	0/2100	3.00	3.33	32:19	\$0.430	56.12%
	sc_2	5/2100	0/2100	8.33	7.67	33:03	\$0.657	32.99%
	sc_3	1/2100	5/2100	2.33	4.00	34:43	\$0.353	63.95%
	sc_4	5/2100	5/2100	5.33	4.93	34:22	\$0.413	57.82%
	sc_5	3/2100	2.5/2100	4.67	5.00	33:00	\$0.442	54.84%
	sc_6	2/2100	1/2100	4.00	4.67	34:01	\$0.523	46.60%
	sc_7	2/2100	2/2100	4.33	3.00	33:15	\$0.410	58.16%

rate ($k_r = 5$). Figures 7.2 and 7.3 show that spot VMs resumed in all executions. For example, in J60, J80, and J100 more than 25% of the hibernated VMs resumed before the hibernation time limit, while in ED200 this number drops to 18%.

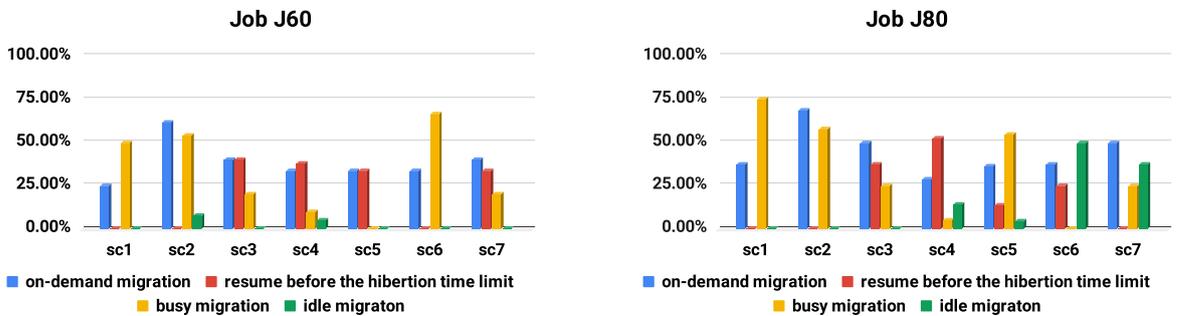


Figure 7.2: Percentage distribution of the procedures used by HADS during the execution of jobs J60 and J80

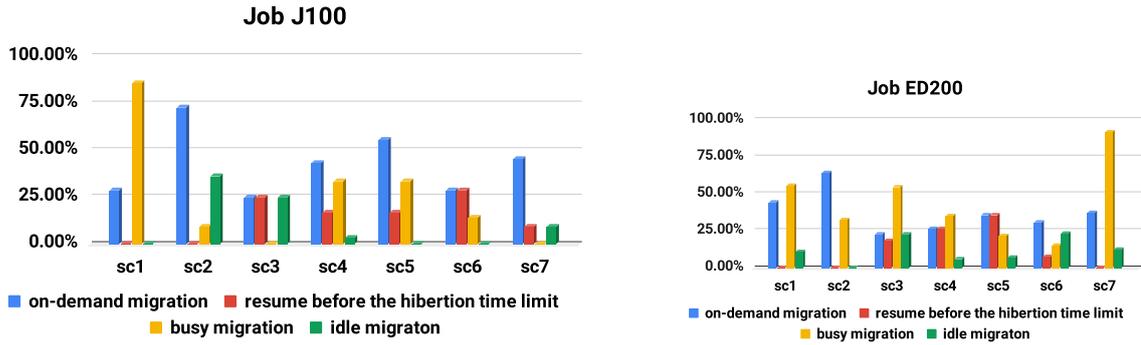


Figure 7.3: Percentage distribution of the procedures used by HADS during the execution of jobs J100 and ED200

By comparing scenarios sc_2 and sc_4 , we can evaluate the impact that resuming spot VMs has on HADS behavior and on the final execution costs. In both scenarios the hibernation rate is $k_h = 5$, while the number of resuming is $k_r = 0$ and $k_r = 5$ for sc_2 and sc_4 , respectively. Due to such a difference, the cost gain in sc_4 is more than 50% for all evaluated jobs, against a maximum cost gain of 32.99% in sc_2 (job ED200). Thus, although the hibernation rate is the same for both scenarios, we observe in Figures 7.2 and 7.3 that migrations to on-demand VMs occurred in less than 50% of the hibernations for sc_4 , but it is more than 50% in sc_2 . Furthermore, as the resuming rate is high in sc_4 , the bars idle migration, busy migration, and resuming before the hibernation time limits are non zero in all executions, as shown in the figures. This behavior is also observed in sc_6 and sc_7 . Both scenarios have a similar hibernation rate ($k_h = 2$). However, as the resume rate of scenario sc_7 ($k_r = 2$) is higher than the one of sc_6 ($k_r = 1$), the former outperforms the latter in terms of cost (more than 50% in all cases) which means that even small increments in the resume rate have a significant impact on cost reduction.

In scenario sc_5 with $k_h = 3$ and $k_r = 2.5$, even if HADS has migrated tasks to on-demand VMs in all evaluated jobs, their cost reduction is between 40% and 70%. We thus suspect that the algorithm's efficiency depends on a trade-off between the number of hibernation and resume events that take place during the execution of an application. Hence, to better understand the impact of these rates, in the next section, we present results from several experiments by varying the rate with which spot VMs hibernate and resume.

7.3.1 Impact of hibernation and resuming

Aiming at evaluating the cost of hibernation rates, we have submitted job ED200, which contains the largest number of tasks compared to the synthetic jobs, to three different scenarios where, at each new execution, the hibernation rate increases by 1. Besides the two baseline cases, we have considered three scenarios: i) any spot VM resumes ($k_r = 0$); ii) executions with medium chances of resuming ($k_r = 3$); and iii) executions with high chances of resuming ($k_r = 7$).

By analyzing Figure 7.4, we could say that in scenarios where spots hibernate, monetary costs variation has an intrinsic relation to the resuming rate of spot VMs. In the scenario where any spot VM resumes ($k_r = 0$), by increasing the number of hibernations, the monetary cost gets closer to the on-demand baseline cost. However, it tends to decrease when the rate of resuming events increases becoming, therefore, cheaper than the former.

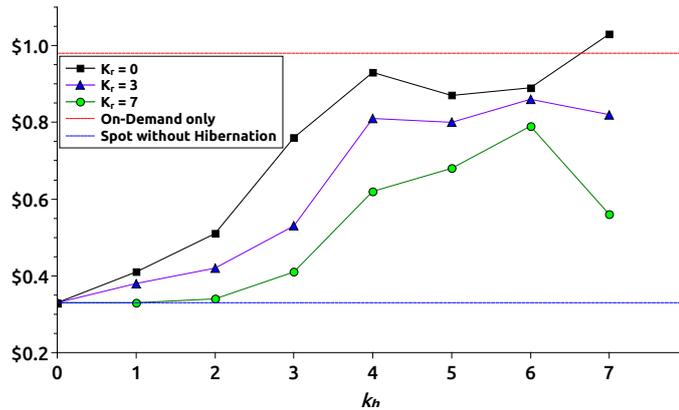


Figure 7.4: Impact of variation of k_h in the execution costs of job ED200

When $k_h = 6$, all three scenarios have almost the same cost. We have observed in this case that, in the three scenarios, all spot VMs hibernated but in different times of tasks execution. Furthermore, some of them hibernated just before the end of the execution of a task, requiring, therefore, on-demand VMs to meet the application deadline. On the other hand, with a hibernation rate of $k_h = 7$, spot VMs behavior changed and they always hibernated at the first few seconds of the tasks execution, consequently leaving more time for the resuming event to take place. Hence, in scenarios where $k_r > 0$, execution costs decrease since the probability that a resuming event happens before mtt increases. On the other hand, when $k_r = 0$, the execution cost is higher than the on-demand baseline cost because the user pays for both the time of the first allocation cycle of the spot VMs

and for the on-demand VMs used in the migration.

7.3.2 Built-in Functions Evaluation

Considering the execution of job ED200, this section studies the behavior of the proposed procedures. Figure 7.5 shows the percentage of times the work-stealing procedure was successfully performed, i.e., it stole at least one task during its execution, in relation to the total number of times it was called. For instance, if during the job execution the procedure was called 10 times and it succeeded in 2 of them, the figure shows the 20% bar.

We observe in Figure 7.5 that scenarios with higher hibernation rates benefit most from work-stealing. In scenario sc_2 ($k_h = 5$), for example, 34.8% of work-stealing calls were successful, followed by sc_5 ($k_h = 3$), where the success rate was 26.30%. On the other hand, in scenario sc_1 , which corresponds to the lowest hibernation rate ($k_h = 1$), only 18.18% of calls succeeded. Such behavior is expected since the work-stealing procedure only steals tasks that are waiting to be executed and which would start executing in the next VM's AC (see Section 6.2.6). In addition, the migration procedure tends to schedule tasks to the next AC too, and the higher the hibernation rate is, the higher the migration rate is. Therefore, scenarios that face more migration events, such as sc_2 and sc_5 , present a higher work-stealing success rate in comparison to the other scenarios.

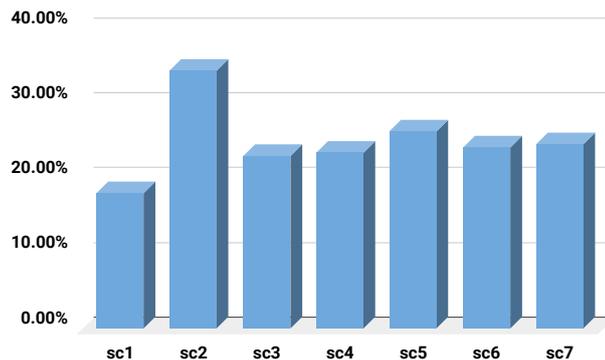


Figure 7.5: Work-stealing distribution of Job ED200

Figures 7.6 and 7.7 illustrate the variation in the number of scheduled tasks (left axis) of VMs c3.large and c4.large, used in one of the executions of job ED200 in scenario sc_2 . Defined by the primary scheduling heuristic, 8 tasks (Figure 7.6) and 6 tasks (Figure 7.7) were initially scheduled to c4.large and c3.large VMs respectively. The figures show that the initial tasks were executed and then, as both VMs became *idle*, the work-stealing

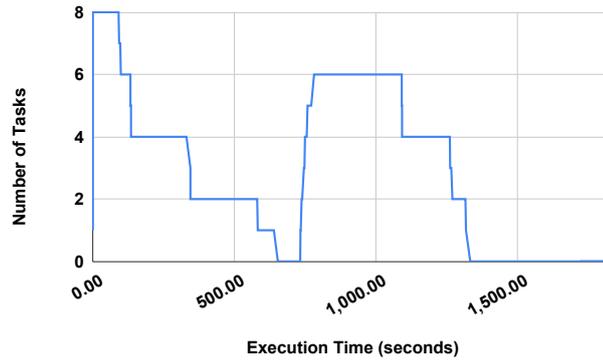


Figure 7.6: Tasks progress of spot VM c4.large during the execution of Job ED200 in scenario sc_2

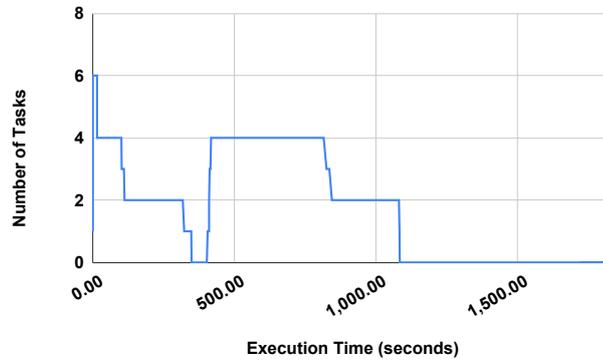


Figure 7.7: Tasks progress of spot VM c3.large during the execution of Job ED200 in scenario sc_2

procedure was performed, stealing 4 (resp., 6) tasks in the case of VM c4.large (resp., c3.large).

Note that, throughout the execution, in both VMs, the work-stealing procedure was executed only once and before the first 15 minutes of execution, i.e., during the first AC of the VMs. Another point is that the VMs pattern of Figures 7.6 and 7.7 was also observed for the other VMs used to execute job ED200. In general, if the tasks of the initial list were completed during the first VM's AC, the work-stealing is effectively performed. Such behavior confirms that the size of the AC, as well as the initial distribution of tasks, has an impact on the work-stealing successful rate.

In order to study the impact of checkpointing in the execution of the ED200 job, we evaluated the dump time to perform checkpoints, varying the size of the memory footprint of the tasks. Based on the results, shown in Figure 7.8, we have analyzed how the former grows in relation to the latter, and then, using a linear regression technique, we have defined an equation that expresses such a relation. We observe in the figure that the

dump time presents almost a linear growth in relation to the memory footprints. Such a relation can be defined by the equation $y = 12.99 + 0.022 \times x$, where x is the memory footprint of a task, and y is the estimated dump time to perform a checkpoint.

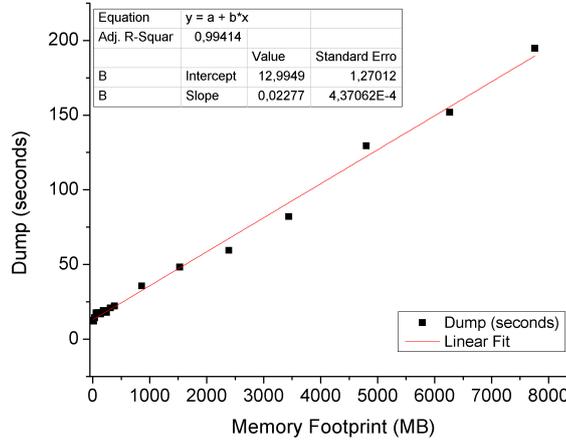


Figure 7.8: Dump time variation

Considering the execution of the ED200 job in each of the seven scenarios described in Table 7.3, Table 7.6 summarizes the average number of checkpoints, task migrations, and tasks recoveries. The last column concerns the saved CPU time due to the use of the checkpoints. The values of the table show that the average number of checkpoints are almost the same for the seven scenarios, slightly varying from 93.00 to 108.67. On the other hand, we observe the strong impact of hibernation rate in the number of task migrations. For example, there is a ratio of nine between the number of migrations in scenario sc_2 ($k_h = 5$ and $k_r = 0$) and scenario sc_1 ($k_h = 1$ and $k_r = 0$). Moreover, in scenario sc_2 , more than 12 minutes of CPU time were saved, while in sc_1 , this number drops to 3:40.

Table 7.6: Average number of checkpoints, migrations, recoveries, and CPU save time from checkpoint in of job ED200 in the seven scenarios

	scenario	# checkpoints	# task migrations	# recoveries from checkpoint	CPU save time
ED200	sc_1	93.00	8.67	3.33	3:40
	sc_2	99.67	80.00	17.33	12:44
	sc_3	101.00	7.67	3.00	4:59
	sc_4	108.67	9.00	5.33	7:04
	sc_5	103.33	8.33	2.67	3:42
	sc_6	96.67	11.00	7.67	9:01
	sc_7	99.33	8.67	8.00	6:51

Chapter 8

Burst Hibernation Aware Dynamic Scheduler

The approach used by the HADS framework increases the total execution time of the application when spot VMs hibernates. To tackle this problem, we propose the use of burstable VMs aiming at reducing the makespan in case of hibernations. Thus, this Chapter presents Burst-HADS, an extension of HADS framework that includes burstable VMs on the scheduling and migration decisions. Unlike HADS, Burst-HADS adopts a multi-objective approach that aims to minimize both the monetary cost and the total execution time while meeting the deadline of the application. In Section 8.1, we present the Burst Primary Scheduling Module, and in Section 8.2 we introduce the Burst Dynamic Scheduling Module.

8.1 Burst Primary Scheduling Module

In this section, we present the extended version of the Primary scheduling module, called Burst Primary Scheduling Module. We first introduce the mathematical formulation of the multi-objective primary task scheduling problem considered by Burst-HADS. After that, we present the proposed Iterated Local Search (ILS) based heuristic [52] to solve that problem in a realistic time.

8.1.1 Mathematical Formulation

We model the primary task scheduling problem of Burst-HADS as a multi-objective integer programming problem whose objectives are to minimize both the monetary cost and the total execution time of the application. In other words, in our case, it is defined as the

problem of creating an initial scheduling strategy, respecting the limit of available vCPUs and memory capacity of the used VMs while minimizing the makespan and the monetary costs of the execution.

Since one of the objectives is to minimize the execution's monetary cost, in the mathematical formulation, we considered only spot VMs of set $M^s \subset M$. Therefore, instead of the deadline D the scheduling solution must respect the D_{spot} value, i.e., the worst-case estimated makespan, which guarantees there will always have enough spare time to migrate tasks of any hibernated spot VM to other VMs and execute them before the deadline D . The D_{spot} value was introduced previously in Section 6.1.1. Let the binary variable X_{ij}^v indicate whether a task $t_i \in B$ allocated to a $vm_j \in M^s$ will start executing ($X_{ij}^v = 1$), or not ($X_{ij}^v = 0$), at time period $v \in T$. Let also Z_j and ZT be continuous variables which respectively keep the last period of execution of a $vm_j \in M^s$ and the total execution time of the application (makespan).

The proposed objective function (Equation 8.1) is a weighted function that minimizes the monetary cost and the makespan, where ω is the weight given by the user for the objectives.

$$\min(\omega \times (\sum_{vm_j \in M^s} Z_j \times c_j) + (1 - \omega) \times ZT) \quad (8.1)$$

Both the monetary cost and the makespan have to be first normalized. The normalization procedure updates the target values to share the same minimum and maximum values, 0 and 1, respectively. Thus, the solution's total monetary cost was divided by the product of the monetary cost of hiring the most expensive spot $vm_j \in M^s$, during D_{spot} periods, times the maximum number of VMs that can be deployed. Similarly, the makespan is divided by D_{spot} . The objective function is subject to the following constraints.

Constraint 8.2 guarantees that every task $t_i \in B$ must be executed, starting at a time $v \in T$ in a $vm_j \in M^s$. Constraint 8.3 assures that tasks' memory demand does not outpace the memory capacity of the VM, while constraint 8.4 guarantees that the number of parallel tasks allocated to a $vm_j \in M^s$ does not exceed the number of virtual cores of the VM.

$$\sum_{vm_j \in M^s} \sum_{v \in T} X_{ij}^v = 1, \forall i \in B \quad (8.2)$$

$$\sum_{t_i \in B} \sum_{q=p}^v r m_i \times X_{ij}^q \leq m_j, \quad (8.3)$$

$$\forall v m_j \in M^s, \forall v \in T, \text{ and } p = \max(v - e_{ij}, 1)$$

$$\sum_{t_i \in B} \sum_{q=p}^v X_{ij}^q \leq |VC_j|, \quad (8.4)$$

$$\forall v m_j \in M^s, \forall v \in T, \text{ and } p = \max(v - e_{ij}, 1)$$

Inequalities 8.5 and 8.6 relate the last period of execution of each $v m_j \in M^s$ with the application total execution time (makespan). Finally, constraint 8.7 assures that the application makespan does not exceed the D_{spot} value.

$$X_{ij}^v \times (v + e_{ij}) \leq Z_j \quad (8.5)$$

$$\forall t_i \in B, \forall v m_j \in M^s \text{ and } \forall v \in T$$

$$X_{ij}^v \times Z_j \leq ZT \quad (8.6)$$

$$\forall t_i \in B, \forall v m_j \in M^s \text{ and } \forall v \in T$$

$$Z_j \leq D_{spot}, \forall v m_j \in M^s \quad (8.7)$$

All variables and parameters used in the mathematical formulation are summarized in Table 8.1.

Table 8.1: Notation and Variables used in the Mathematical Formulation.

Name	Description
B	Set of tasks
M^s	Set of spots VMs
T	Discretized time set
D_{spot}	Estimated time limit which ensures that there will be enough spare time to migrate tasks of a hibernated spot VM to other VMs no matter when hibernations take place
vm_j	Virtual machine
m_j	Memory capacity of vm_j in gigabytes
c_j	Cost per period of time of vm_j
VC_j	Set of cores of vm_j
t_i	a task of the BoT application
rm_i	Amount of memory required by a task t_i
e_{ij}	Time required to execute task t_i in a vm_j
X_{ij}^v	Binary variable which indicates whether task $t_i \in B$ begins its execution in a $vm_j \in M^s$ at time period $v \in T$ or not
Z_j	Continuous variable which keep the last period of execution of a vm_j
ZT	Continuous variable which indicates the total time to execute the BoT application (makespan)
ω	Weight of the objectives in the fitness function

8.1.2 Iterated Local Search Heuristic

To create the initial scheduling map according to the mathematical formulation, we propose an Iterated Local Search (ILS) able to solve the problem in an acceptable time. The ILS [52] is a metaheuristic that aims at improving a final solution by sampling in a broader and distant neighborhood of candidate solutions and then applying a local search technique to refine solutions to their local optima. It explores a sequence of solutions created by perturbations of the current best solution to reach these distant neighborhoods.

After finding a scheduling map with the ILS, a second heuristic is applied to include burstable instances of set M^b into the solution. In case of spot VM hibernations, these instances will be used in burst mode by the Dynamic Scheduler Module, as an attempt to minimize the impact of these hibernations in the monetary cost and/or the execution time. Therefore, the Primary Task Scheduling Algorithm (Algorithm 9) has two parts: i) the Iterated Local Search, that solves the scheduling problem and ii) the burstable instances allocation, that includes burstable instances to the final solution.

Firstly, in line 2 of Algorithm 9, a initial solution is generated by calling the Primary Scheduling Heuristic (Algorithm 3, presented in Chapter 6, Section 6.2). After obtaining that solution, the ILS tries to improve it by applying local search and perturbation procedures (Algorithm 9, lines 3 to 20). Thus, let S be a solution that defines a scheduling map of all tasks $t_i \in B$ to a subset of VMs of $M^s \cup M^o$. Let $fitness(S)$ be a weighted function that assigns a value to the quality of S . Since this function is equivalent to the objective function presented in Equation 8.1, we define in Equation 8.8 the $fitness(S, D_{spot})$ function, where $cost$ is the total monetary cost of S and mkp is the total execution time of the application.

$$fitness(S, D_{spot}) = \begin{cases} \infty, & \text{if violates } D_{spot} \\ \omega \cdot cost + (1 - \omega) \cdot mkp, & \text{otherwise} \end{cases} \quad (8.8)$$

Algorithm 9 executes a local search by calling, in line 3, the *local_search* procedure (Algorithm 10), which executes a series of attempts to improve the current solution by swapping tasks between the selected VMs. Algorithm 10 receives as input the current solution S , the *max_attempt* parameter that determines the number of times the local search will be executed, the set of tasks B , the *swap_rate* and the D_{spot} value. The *swap_rate* $\in [0, 1]$ parameter is tuned before the execution and determines the number of tasks that will be swapped at each iteration. All the parameters used in our tests, including the *swap_rate*, *max_attempt*, *max_iteration* and other parameters will be presented next in Chapter 7.

As can be observed in lines 4 and 8 of Algorithm 10, a solution S is composed of two structures: (i) a vector, which controls task allocation, where indexes correspond to tasks, and each element keeps the identity of the VM that will execute the corresponding task, and (ii) a list composed by selected VMs. Firstly, the algorithm computes the number of tasks that will be swapped at each iteration (line 2) and randomly selects a destination VM (vm_{dest} , line 4). After that, the algorithm starts the tasks swapping procedure (lines 5 to 14), where n tasks, also randomly selected (line 7), are moved to the vm_{dest} . After each swap *movement*, the *local_search* procedure checks if the quality of the new generated solution has improved (line 9) and it updates the S_{best} solution, if necessary. In the end, the procedure returns the best solution (line 15).

Algorithm 9 *ILS Primary Task Scheduling***Input:** $B, M^o, M^s, M^b, max_iteration, max_attempt, max_failed, relaxed_rate, D_{spot}$ **and** D

```

1: {/*PART 01 - Iterated Local Search*/}
2:  $S \leftarrow initial\_solution(B, M^s, D_{spot})$  {Algorithm 3}
3:  $S \leftarrow local\_search(S, max\_attempt, B, swap\_rate, D_{spot})$  {Algorithm 10}
4:  $S_{best} \leftarrow S$ 
5:  $RD_{spot} \leftarrow D_{spot}$ 
6:  $it, it_{best} \leftarrow 0$ 
7: while  $it < max\_iteration$  do
8:    $vm_j \leftarrow random\_choice(M^s)$ 
9:    $S.selected\_vms \leftarrow S.selected\_vms \cup vm_j$ 
10:   $M^s \leftarrow M^s \setminus \{vm_j\}$ 
11:  if  $(it - it_{best}) > max\_failed$  then
12:     $RD_{spot} \leftarrow RD_{spot} + (relaxed\_rate \times RD_{spot})$ 
13:  end if
14:   $S \leftarrow local\_search(S, max\_attempt, B, swap\_rate, RD_{spot})$ 
15:  if  $fitness(S, RD_{spot}) < fitness(S_{best}, RD_{spot})$  then
16:     $S_{best} \leftarrow S$ 
17:     $it_{best} \leftarrow it$ 
18:  end if
19:   $it \leftarrow it + 1$ 
20: end while
21:
22: {/*PART 02: Burstable instance allocation*/}
23:  $n' \leftarrow \lceil burst\_rate \times |S_{best}.selected\_vms| \rceil$ 
24:  $S_{final} \leftarrow burst\_allocation(S_{best}, burst\_rate, M^b, D_{spot}, D)$ 
25:  $map \leftarrow create\_primary\_map(S_{final})$ 
26: return  $map$ 

```

After the first execution of the *local_search* procedure (Algorithm 9, line 3), Algorithm 9 has a loop that firstly performs a perturbation (lines 8 to 13) and then a new local search (line 14). The perturbation is responsible for diverting the metaheuristic from local optimal solutions. In the current work, we use two perturbation strategies. The first one includes a not selected spot $vm_j \in M^s$ into the current solution S (lines 8 to 10). The second one, called relaxing perturbation, increases the D_{spot} limit (lines 11 to 13). Note that the latter is executed only when the number of iterations without finding a better solution is higher than the *max_failed* parameter (line 11). In this case, the metaheuristic increases the D_{spot} limit in *relaxed_rate* percent, where $relaxed_rate \in]0, \dots, 1]$ is also a parameter defined by the user.

Upon finishing the ILS (Part 1), Algorithm 9 executes the *burst_allocation* procedure

Algorithm 10 *Local Search*

Input: S , $max_attempt$, B , $swap_rate$ and D_{spot}

- 1: $S_{best} \leftarrow S$
- 2: $n \leftarrow swap_rate \times |B|$
- 3: $attempt \leftarrow 0$
- 4: $vm_{dest} \leftarrow random_choice(S.selected_vms)$
- 5: **while** $attempt < max_attempt$ **do**
- 6: **for** $k \in \{1, \dots, n\}$ **do**
- 7: $t_i \leftarrow random_choice(B)$
- 8: $S.allocation_array[t_i] \leftarrow vm_{dest}$
- 9: **if** $fitness(S, D_{spot}) < fitness(S_{best}, D_{spot})$ **then**
- 10: $S_{best} \leftarrow S$
- 11: **end if**
- 12: **end for**
- 13: $attempt \leftarrow attempt + 1$
- 14: **end while**
- 15: **return** S_{best}

(Part 2, lines 22 to 24). In this procedure, n' burstable VMs are included in the final solution. The number of burstable VMs is defined as a percentage, given by the parameter $burst_rate$, of the selected spot VMs of the best solution found by the ILS. For example, if 20 spot VMs were selected by the ILS, with $burst_rate = 0.1$, only 2 burstable VMs will be included.

Since the relaxed perturbation leads some tasks to violate the D_{spot} limit, the $burst_allocation$ procedure also moves these tasks to the burstable instances. Each burstable VM can receive at most one task to be executed in baseline mode. However, if there still exist tasks violating D_{spot} and no available burstable VM, the procedure allocates them to the cheapest regular on-demand VMs. On the other hand, if a burstable VM remains idle, the task with the latest finishing time in the scheduling map is moved to it. Remark that our strategy of having a single task per burstable instance at a time, executing in baseline mode, induces CPU credits accumulation. Consequently, these burstable instances become the best candidates to receive tasks in case of hibernations.

The first part of Algorithm 9 have complexity $\mathcal{O}(|B| \times (\log |B| + |A| \log |A| + |M^o|) + (max_iteration \times max_attempt \times n))$ which is the time complexity of Algorithm 3, used to create the initial solution, plus the complexity of ILS main loop (lines 7 to 20). At each iteration of the main loop the algorithm calls the $local_search$ procedure (Algorithm 10), whose complexity is $max_attempt \times n$, where n is the number of swapped tasks and $max_attempt$ is the number of times the procedure try to find a better solution. In the second part of Algorithm 9 (lines 22 to 24), the complexity is $\mathcal{O}(n' \times k)$, where n' is the number of burstable VMs that will be included in the final solution and k represents the number of tasks violating the D_{spot} value. Since $|B| \times (\log |B| + |A| \log |A| +$

$|M^o|) + (max_iteration \times max_attempt \times n) \gg n' \times k$ the complexity of Algorithm 9 is $\mathcal{O}(|B| \times (\log |B| + |A| \log |A| + |M^o|) + (max_iteration \times max_attempt \times n))$. Moreover, Algorithm 9 receives four lists of integers, representing sets B , M^o , M^s and M^b , where each element in a list represents either the ID of a task or of a VM. Thus, as $|B| \gg |M|$, in terms of space complexity Algorithm 9 is $\mathcal{O}(|B|)$.

8.2 Burst Dynamic Scheduling Module

As in the case of the Primary Scheduling Module, HADS' Dynamic Scheduling Module was also extended to include the burstable instances. Moreover, instead of computing the migration time limit value (mtt) and wait (as present previously in Chapter 6), the Burst Dynamic Scheduling Module immediately migrates tasks when a spot instance hibernates. That approach is adopted since the module has burstable VMs available to receive the tasks. Note that, in this case, although Burst-HADS does not wait for a hibernated VM to resume and continues executing its tasks, the framework keeps all hibernated VM as a resource that, if resumes, can be used again to reduce the monetary cost and the execution time of the application. The migration procedure is presented next.

8.2.1 Migration Procedure

As previously explained, as soon as a spot vm_l hibernates, Burst-HADS executes the migration procedure, searching for a set of VMs to assign and execute non-finished tasks, denoted affected tasks, that were previously scheduled to vm_l . Algorithm 11 presents the migration procedure, which always respects the deadline D when selecting tasks to migrate. It receives as input the set $Q_l \subset B$ of affected tasks, the sets of idle, busy, and non-launched regular on-demand VMs (IR , BR and M^o , respectively), and the deadline D . Note that, in the case of a burstable $vm_j \in M^b$, e_{ij} is the execution time of task t_i in vm_j in burst mode (100% of the vm_j processing power) and each $vm_j \in M^b$ have a current CPU credit amount cc_j that is constantly updated by the cloud provider (in the case of $vm_j \notin M^b$, i.e., non-burstable VMs, $cc_j = \infty$).

Initially, Q_l is ordered, giving priority to those tasks that were executing at the moment of the hibernation and had been previously checkpointed (line 1). In order to avoid the overhead of launching new VMs, the migration procedure gives priority to the use of already launched VMs. For each task $t_i \in Q_l$ the algorithm first tries to migrate the task to an *idle* burstable VM (lines 4 to 13). Otherwise, it tries to schedule t_i to one of the

non-burstable *idle* or *busy* VM of set $K = IR \cup BR$ (lines 18 to 27). Note that, the set K is created by sorting the idle and busy VMs according to its market, putting the spot VMs ahead of the set (Algorithm 11 line 17).

Algorithm 11 *Migration Procedure***Input:** Q_l, IR, BR, M^o , and D

```

1:  $Q_l \leftarrow \text{sort\_tasks}(Q_l)$  {/* Prioritizes tasks with checkpoints */}
2: for each  $t_i \in Q_l$  do
3:   {Attempt 1 - Try to migrate task to a Burstable IDLE VM}
4:   for each burstable  $vm_j \in IR$  do
5:      $rcc_{ij} \leftarrow \lceil e_{ij}/\text{burst\_period} \rceil$ 
6:     if  $cc_j > rcc_{ij}$  and  $\text{check\_migration}(t_i, vm_j, D)$  then
7:       {Migrate  $t_i$  to burstable  $vm_j$  on burst mode}
8:        $\text{migrate}(t_i, vm_j)$ 
9:        $IR \leftarrow IR \setminus \{vm_j\}$ 
10:       $BR \leftarrow BR \cup \{vm_j\}$ 
11:       $\text{break}$  {/*Migrate next task*/}
12:     end if
13:   end for
14:   {Attempt 2 - Try to migrate task to a NON-burstable Idle or Busy VM}
15:   if not  $\text{migrated}$  then
16:     {/* Prioritizes idle spot VMs */}
17:      $K \leftarrow \text{sort\_by\_market}(IR \cup BR)$ 
18:     for each NON-burstable  $vm_j \in K$  do
19:       if  $\text{check\_migration}(t_i, vm_j, D)$  then
20:          $\text{migrate}(t_i, vm_j)$ 
21:         if  $vm_j \in IR$  then
22:            $IR \leftarrow IR \setminus \{vm_j\}$ 
23:            $BR \leftarrow BR \cup \{vm_j\}$ 
24:         end if
25:          $\text{break}$  {/*Migrate next task*/}
26:       end if
27:     end for
28:   end if
29:   {Attempt 3 - Migrate task to a new NON-burstable on-demand VM}
30:   if not  $\text{migrated}$  then
31:      $\text{sort\_by\_price}(M^o)$ 
32:     for each  $vm_j \in M^o$  do
33:       if  $\text{start}_{ij} + e_{ij} + \alpha < D$  then
34:          $\text{start\_vm}(vm_j)$ 
35:          $\text{migrate}(t_i, vm_j)$ 
36:          $M^o \leftarrow M^o \setminus \{vm_j\}$ 
37:          $BR \leftarrow BR \cup \{vm_j\}$ 
38:          $\text{break}$  {/*Migrate next task*/}
39:       end if
40:     end for
41:   end if
42: end for

```

Tasks migrated to burstable VMs are executed in burst mode, i.e., using 100% of the VM's CPU processing power. Therefore, it is necessary to guarantee that a selected burstable VM will have enough CPU credits to execute all tasks assigned to it. Let *burst_period* be the number of periods of T corresponding to one credit consumption in burst mode. Since we consider that a task t_i is executed in only one core (see Section 4.1), it is possible to estimate (line 5) the number of CPU credits consumed by task t_i , where e_{ij} is the execution time of t_i in vm_j at burst mode, and rcc_{ij} is the estimated number of required CPU credits. Then, the algorithm checks if vm_j has enough CPU credits and call the *check_migration* function to guarantee that vm_j has enough memory and will execute t_i before the deadline D (line 6). If both conditions are satisfied, t_i is migrated to vm_j at the burst mode (line 8). In this case the burstable vm_j is removed from set IR and included in set BR (lines 9 and 10). Note that the cloud provider continuously updates the number of credits, cc_j , of every burstable VM.

Moreover, when Algorithm 11 migrates a task t_i to a non-burstable *idle* or *busy* vm_j , it also calls the function *check_migration* (Algorithm 7) to verify if the VM has enough memory and will be able to finish the task before the deadline D (line 19). Furthermore, if vm_j is a spot VM, the function *check_migration* should also verify if there will be enough spare time in vm_j between the end of the execution of vm_j tasks (including task t_i) and the deadline D since, in this case, a *busy* or *idle* spot vm_j is also subject to hibernation. The spare time has to be greater than the execution time of the longest task scheduled to vm_j , ensuring, therefore, that if a hibernation occurs, there will be enough time to migrate and execute all affected tasks before the deadline D .

Finally, if there does not exist any available already deployed VM able to execute task t_i , the algorithm migrates the task to a new on-demand VM of set M^o (lines 32 to 40). In this case, it is necessary to verify that, considering the start period of t_i in vm_j ($start_{ij}$), plus its execution time (e_{ij}) will not violate the deadline (line 33). The new allocated on-demand VM is then removed from set M^o and included in set BR (lines 36 and 37). Note that, in terms of time and memory complexity, the analysis of Algorithm 11 is similar to the analyses presented in Section 6.2.5.

8.2.2 Work-Stealing Procedure

As presented in Section 6.2.6, the work-stealing procedure aims to reduce the allocation time of regular on-demand VMs and balance the load of spot VMs. It is triggered when a hibernated spot VM resumes or when a VM (spot or on-demand) becomes *idle*. In the

case of Burst-HADS, Algorithm 12 tries to move tasks from non-burstable *busy* VMs to the *idle* VM.

Algorithm 12 *Burst Work-Stealing Procedure*

Input: BR, IR, vm_k, D

```

1: sort_by_market( $BR$ ) {/*Prioritizes non-burstable on-demand VMs*/}
2: for each NON-burstable  $vm_j \in BR$  do
3:    $ST_j \leftarrow selectTasks(vm_j)$  {Select the tasks that can be stolen/}
4:   for each  $t_i \in ST_j$  do
5:     if check_migration( $t_i, vm_k, D$ ) then
6:       if  $vm_k$  is burstable then
7:         set_baseline_mode( $t_i, vm_k$ )
8:         migrate( $t_i, vm_k$ )
9:         stops the loop
10:      else if  $vm_k$  not burstable then
11:        migrate( $t_i, vm_k$ )
12:      end if
13:    end if
14:  end for
15: end for
16: if at least one task was stolen then
17:    $BR \leftarrow BR \cup \{vm_k\}$ 
18:    $IR \leftarrow IR \setminus \{vm_k\}$ 
19: end if

```

For each non-burstable *busy* $vm_j \in BR$ the procedure selects the tasks that can be stolen from it (line 3) and tries to migrate them to the *idle* vm_k (lines 4 to 14). Since regular on-demand VMs are more expensive than spot ones, the procedure considers firstly the tasks from the former (line 1).

Similarly to the migration procedure, for each selected task of a vm_j , the work-stealing procedure also verifies, by calling the function *check_migration* (Algorithm 7), if the task migration would result in the deadline violation (line 5). Since tasks migrated to burstable VMs by the work-stealing procedure are executed in the baseline mode, after verifying if the *idle* vm_k is burstable (line 6), the algorithm sets up the execution to the baseline mode (line 7). In this work, to set up a burstable VM to the baseline mode means that the task cannot run using 100% of the CPU processing power, but uses only the baseline performance defined by the provider¹. To limit the CPU utilization, Burst-HADS uses the *cpulimit* tool [55], an open-source program that limits the CPU usage of a process in

¹<https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/burstable-credits-baseline-concepts.html>

GNU/Linux. Moreover, if the *idle* vm_k is burstable, only one task is moved to it, to avoid a queue of tasks to be executed at the baseline mode, which could increase the application total execution time. Thus, after moving one task to the burstable vm_k , the algorithm stops the loop (line 9). Finally, if at least one task is migrated to vm_k , its state changes to *busy* and, consequently, it is included into the set of *busy* VMs and removed from the *idle* set (lines 17 and 18). Note that, in terms of time and memory complexity, Algorithm 12 has a complexity similar to the analyses of Algorithm 8, presented in Section 6.2.6.

Chapter 9

Burst-HADS Experimental Results

In this Chapter, we present the evaluation results related to the Burst-HADS framework. At first, in Section 9.1, we present the experimental environment, including the input parameters and the used VMs. Next, in Section 9.2, we evaluate the first part of the Algorithm 9 (presented in Chapter 8) in terms of quality of solution and execution time. In Section 9.3, we present the baseline job execution cases, while in Section 9.4 we show the practical evaluation in EC2 considering different scenarios of hibernation and resuming. Finally, Section 9.5 presents a study case where Burst-HADS was used to execute SARS-CoV-2 sequence comparison with the MASA-OpenMP tool.

9.1 Experimental Environment

As in HADS evaluation presented in Chapter 7, to evaluate Burst-HADS, we also use the BoT applications presented in Section 7.1, and the hibernation and resume events was also emulated (Section 7.1.1). Moreover, we also adopted VMs from c3 and c4 family, however, as a burstable resource, we include the newest generation of the burstable instance t3.large. To store checkpoint files, the tests present in Sections 7.2 and 9.4 use the S3 service. Table 9.1 shows the computational characteristics and the corresponding prices in on-demand and spot markets. Note that the results present in this chapter were originally published in [77]. Thus, all presented prices are from November 2020.

For the execution of Algorithms 9 and 10, the following input parameters were used: $\alpha = 0.5$, $max_iteration = 200$; $max_attempt = 50$; $swap_rate = 0.10$; $max_failed = 20$; $relaxed_rate = 0.25$; $burst_rate = 0.2$. Moreover, we defined $AC = 900$ seconds and the deadline was setup to 45 minutes ($D = 2700s$). Except for $\alpha = 0.5$, which was used to give the same weights to both objectives, the parameters' values were defined by executing

Table 9.1: VMs attributes

Type	#VCPUs	Memory	Price per Hour (USD)		Baseline performance
			on-demand	spot	
c3.large	2	3.75 GB	0.105\$	0.0299\$	-
c4.large	2	3.75 GB	0.100\$	0.0366\$	-
c3.xlarge	4	7.50 GB	0.199\$	0.0634\$	-
t3.large	2	8 GB	0.0832\$	-	20%

a set of empirical tests.

We create five different scenarios for Burst-HADS' practical tests by varying parameters k_h and k_r of the events emulation. Table 9.2 presents those scenarios. Note that, since we adopt a different deadline, the scenarios are not the same as the ones presented in Table 7.3.

Table 9.2: Different execution scenarios generated by varying parameters λ_h and λ_r

ID	<i>hibernating</i>	<i>resuming</i>	λ_h	λ_r
s_1	$k_h = 1$	$k_r = 0$	1/2700	0/2700
s_2	$k_h = 5$	$k_r = 0$	5/2700	0/2700
s_3	$k_h = 1$	$k_r = 5$	1/2700	5/2700
s_4	$k_h = 5$	$k_r = 5$	5/2700	5/2700
s_5	$k_h = 3$	$k_r = 2.5$	3/2700	2.5/2700

9.2 Evaluation of the ILS Primary Task Scheduling

To evaluate the quality of the solution given by the ILS executed in the first part of the primary scheduling heuristic (Algorithm 9), we compared it with the optimal solution given by the exact model and also with the solutions given by three widely used benchmark scheduling heuristics [40]:

- **MinMin**, where the scheduling gives priority to VMs with the minimum earliest completion time;
- **MaxMin**, where the scheduling gives priority to VMs with the maximum earliest completion time; and
- **Greedy**, where the shortest execution time task is scheduled to the cheapest available VM.

For the comparison with the exact model we create six small jobs using also the synthetic application proposed in [3], composed of tasks with execution times varying from 5 to 15 seconds (from 5 to 30 tasks). We opted for that set with up to 30 tasks because the exact model took more than ten hours to solve the problem for jobs with more than 30 tasks. In these experiments, the deadline was fixed in 15 minutes ($D = 900$) for all jobs, and we used as input the spot VMs presented in Table 9.1, except for the burstable VM, since we are evaluating just the ILS heuristic. In these experiments, a computer with a processor Intel Core i7-3770 CPU 3.40 GHz with 12GB of memory and Ubuntu 18.04 was used. The mathematical formulation was solved by using the Gurobi Solver 9.0 [39].

Table 9.3 summarizes the results of both the exact approach and the ILS-based heuristic. The first column identifies the number of tasks. The three columns that follow correspond to the results achieved by ILS: makespan, cost and the execution time to obtain the solution. The next three columns present the same results for the exact approach. The values shown for the ILS are averages of three executions. The ILS standard deviations were zero for all jobs.

Table 9.3: Results of the ILS-based Primary Scheduling Heuristic and the Exact Approach

# Tasks	ILS			Exact Approach		
	Makespan	Cost	Time	Makespan	Cost	Time
5	21	\$0.0005	0.0071	21	\$0.0005	49.04
10	34	\$0.0011	0.0306	34	\$0.0011	70.10
15	34	\$0.0020	0.0297	34	\$0.0020	106.03
20	34	\$0.0021	0.0558	34	\$0.0021	233.10
25	34	\$0.0027	0.0736	34	\$0.0027	751.72
30	34	\$0.0035	0.0922	34	\$0.0035	9346.21

As we can observe in Table 9.3, ILS obtained the same solutions of the exact approach for these small jobs, taking significantly less time than the mathematical formulation. On average, the ILS takes 0.048s against 2101.43s of the exact approach. Although these results are very encouraging, the experiments were limited to jobs with few tasks (up to 30 tasks) due to the huge required time to obtain the exact solution for jobs with more tasks.

For the evaluation with the other heuristics, six synthetic jobs were created with sizes varying from 50 to 200 tasks. The tasks execution times varied from 15 to 35 seconds, and the same deadline of 15 minutes ($D = 900$) was applied. Table 9.4 presents the makespan and monetary cost obtained by the ILS, MinMin, MaxMin and Greedy heuristics. As

shown in Table 9.4, the ILS heuristic outperforms MinMin in terms of monetary cost but increases the makespan. On average the ILS makespan presents an increment of 18.11%, while the average monetary cost reduces by more than 38.00%.

Table 9.4: Results of the ILS-based Primary Scheduler, MinMin, MaxMin and Greedy Heuristics.

# Tasks	ILS		MinMin		MaxMin		Greedy	
	mkp	cost	mkp	cost	mkp	cost	mkp	cost
50	57	\$0.0060	47	\$0.0073	610	\$0.0055	597	\$0.0051
100	86	\$0.0112	71	\$0.0220	897	\$0.0107	843	\$0.0101
150	109	\$0.0169	99	\$0.0338	885	\$0.0157	823	\$0.0148
200	138	\$0.0235	123	\$0.0376	898	\$0.0220	819	\$0.0205
250	163	\$0.0282	133	\$0.0461	888	\$0.0261	793	\$0.0246
300	204	\$0.0340	168	\$0.0551	899	\$0.0323	775	\$0.0317

Compared to MaxMin and Greedy, the ILS presents an average reduction in the makespan by more than 80%, with an average increment in the monetary cost by only 6.78% when compared to MaxMin, and 13.14% when compared to the Greedy heuristic. Contrary to the MinMin heuristic, both MaxMin and Greedy heuristics reduce the solution’s monetary cost since they give priority to the cheapest VMs, which increases the makespan.

It is worth noticing that both objectives were evenly considered in the ILS solutions and that the average loss in one of the objectives was always smaller than the gain in the other. Those results confirm the effectiveness of the proposed ILS to the BoT primary scheduling problem.

9.3 Baseline executions

We have firstly evaluated Burst-HADS in a scenario without hibernation, comparing it with (1) the schedule given by the proposed ILS using only on-demand VMs and (2) the schedule given by HADS in a scenario without hibernation. The aim of these experiments is to measure the impact in the monetary cost and execution time of including burstable on-demand VMs into the scheduling procedure. Table 9.5 presents the average of three executions of the synthetic jobs J60, J80, and J100, and the real application ED200, for each case.

In comparison with HADS, Table 9.5 shows that Burst-HADS reduces the makespan in 44.37%, 42.09%, 28.82%, and 11.82%, for jobs J60, J80, J100, and ED200, respectively. However, the average monetary cost increases by 66.34%, 44.54%, 57.55%, and 33.71%,

for the same comparison. The latter increases because Burst-HADS already starts by using some burstable on-demand VMs. Moreover, the ILS based primary scheduling uses more VMs to reduce the execution time, while in HADS, the initial scheduling aims at minimizing only the monetary cost.

Table 9.5: Cost and Makespan of Burst-HADS and HADS, without hibernation; and ILS On-demand only.

JOB	Burst-HADS		HADS		ILS On-demand	
	Without Hibernation		Without Hibernation			
	cost	makespan	cost	makespan	cost	makespan
J60	\$0.112	1274	\$0.067	2290	\$0.271	1112
J80	\$0.151	1329	\$0.104	2295	\$0.312	1190
J100	\$0.176	1660	\$0.112	2332	\$0.371	1462
ED200	\$0.357	2275	\$0.267	2580	\$0.698	1887

On the other hand, compared to the ILS on-demand strategy, on average, Burst-HADS reduces the monetary cost by more than 52.00%, with an average increase of 15% in the makespan. The ILS on-demand strategy uses the scheduling plan given by the ILS proposed in Algorithm 9, which does not include spot neither burstable VMs, but only regular on-demand VMs. The longer makespan is due to the execution of tasks in the baseline mode of burstable VMs, which does not occur in the ILS on-demand strategy.

9.4 Performance Results

Table 9.6 presents the averages of three executions of jobs J60, J80, J100, and ED200 using Burst-HADS and HADS in each of the five execution scenarios. For each job and scenario, the table shows the average number of hibernations followed by resume events. It also includes the number of non-burstable on-demand VMs launched by the Dynamic Module of both Burst-HADS and HADS that handle the hibernations as well as the average of monetary cost and makespan. Finally, the last two columns represent the percentage difference between Burst-HADS and HADS (diff) related to the monetary cost and the makespan.

As we observe in Table 9.6, Burst-HADS minimizes the makespan in all execution scenarios, presenting an average reduction of 25.87%. As explained in Section 8.2.1, whenever a spot VM hibernates, Burst-HADS immediately migrates the interrupted tasks to other VMs. Thus, the increase in the makespan is due to the overhead of this procedure, which might include the launch of new VMs. However, small jobs, i.e., jobs with fewer tasks, are less affected in those scenarios than the biggest ones. For example, while

Table 9.6: Comparison between Burst-HADS and HADS in terms of monetary cost and makespan in scenarios s_1 to s_5

Job	scenario	# hibernations	# resume	# used regular on-demand VMs		Burst-HADS		HADS		Diff (%)	
				Burst-HADS	HADS	cost	makespan	cost	makespan	cost	makespan
J60	s_1	0.66	0.00	0.00	0.00	\$0.119	1274	\$0.091	2620	-30.77%	51.37%
	s_2	3.33	0.00	1.33	2.33	\$0.204	1277	\$0.257	2549	20.54%	49.90%
	s_3	2.33	2.33	1.33	0.00	\$0.127	1752	\$0.101	2539	-26.07%	31.00%
	s_4	5.33	4.00	1.67	0.00	\$0.142	1857	\$0.119	2634	-19.90%	29.50%
	s_5	2.66	1.00	1.33	2.00	\$0.150	1445	\$0.169	2359	11.44%	38.75%
J80	s_1	1.00	0.00	1.33	0.33	\$0.167	1419	\$0.150	2581	-11.33%	45.03%
	s_2	5.00	0.00	1.00	3.00	\$0.210	2267	\$0.298	2591	29.48%	12.50%
	s_3	3.00	1.00	1.67	1.00	\$0.164	1367	\$0.147	2602	-11.34%	47.46%
	s_4	9.66	7.66	1.00	2.00	\$0.244	2488	\$0.212	2607	-15.25%	4.56%
	s_5	3.00	1.00	1.33	3.00	\$0.195	1589	\$0.246	2529	20.47%	37.17%
J100	s_1	2.00	0.00	0.00	0.00	\$0.191	1798	\$0.157	2332	-21.76%	22.90%
	s_2	7.00	0.00	1.33	3.00	\$0.212	1900	\$0.353	2518	39.94%	24.54%
	s_3	6.00	3.00	1.67	1.00	\$0.201	1925	\$0.166	2636	-21.08%	26.97%
	s_4	11.00	9.00	1.00	0.00	\$0.286	2453	\$0.278	2591	-2.88%	5.33%
	s_5	3.66	2.00	1.00	2.50	\$0.166	1547	\$0.189	2543	12.49%	39.15%
ED200	s_1	3.00	0.00	1.00	0.33	\$0.388	2327	\$0.314	2680	-23.57%	13.17%
	s_2	8.00	0.00	2.00	5.00	\$0.482	2448	\$0.512	2676	5.86%	8.52%
	s_3	6.66	4.00	2.33	1.00	\$0.427	2345	\$0.387	2672	-10.34%	12.24%
	s_4	9.00	6.00	2.00	1.00	\$0.411	2560	\$0.389	2690	-5.66%	4.83%
	s_5	4.33	2.33	1.67	3.00	\$0.367	2342	\$0.467	2674	21.41%	12.42%

for job J60, the average makespan reduction, considering all scenarios, is 40.10%, for job ED200, that reduction is only 10.24%. Such a behaviour can be explained because, in our experiments, we have fixed the same deadline for all jobs and, therefore, small jobs have more spare time between its expected makespan defined by the ILS and the deadline. Consequently, in this case, Burst-HADS benefits more from the burst mode of the burstable VMs since it has more idle time to earn CPU credits. Moreover, it also executes the work-stealing more frequently, which also reduces the makespan. On the other hand, independently of the scenario or job, HADS's makespan get closer to the deadline whenever a hibernation occurs. That happens because the HADS framework postpones as much as possible the execution of the migration procedure. Since HADS gives priority to the monetary cost save, its central idea is to wait for the resume of hibernated VMs, avoiding then the launch of new VMs.

In scenarios s_2 and s_5 , Burst-HADS improved the monetary cost for all jobs. The s_2 is the worst execution case scenario, since it has the highest rate of hibernation ($k_h = 5$) and no resume rate ($k_r = 0$), while s_5 is the average case scenario where the rate of hibernation is $k_h = 3.0$, and the rate of resume is $k_r = 2.5$ (see Table 9.2). In these scenarios, Burst-HADS uses fewer regular on-demand VMs than HADS. Moreover, in both cases, the number of hibernations is higher than the number of resumes. Hence, in those cases, by migrating tasks to busy and idle VMs as well as to burstable VMs, exploring the burst mode, Burst-HADS is more effective in minimizing the impact of spot hibernations than HADS. It is worth also pointing out that, considering all executions, the average increase of Burst-HADS makespan is 1.92%.

Compared to the ILS On-demand, both Burst-HADS and HADS minimized the monetary cost for all execution cases, presenting an average reduction of 41.80% and 39.65%, respectively. For job ED200, for example, the worst execution scenario, s_2 , Burst-HADS reduced the monetary cost by 30.96%, while HADS presented an average reduction of 26.66%.

9.5 Case study: A Sequence Alignment Problem

An important application that is well suited to cloud environments is biological sequence comparison, where sequences are pairwise compared in search of similarities. The Covid-19 pandemic study is of particular interest nowadays, and the comparison of SARS-CoV-2 sequences is crucial to understanding this lethal disease. In public genomic databases there are more than 22,000 SARS-CoV-2 complete genome sequences obtained in different countries from December 2019 to November 2020. To compare these sequences in a reasonable time with accurate results, highly specialized tools are needed. MASA-OpenMP [69] is a multithreaded freely available tool that compares two DNA sequences with the Smith-Waterman algorithm, providing the optimal result.

To execute 22,600 sequences comparisons in the cloud, we propose an execution modeling for MASA-OpenMP[69] to be managed by the Burst-HADS framework. Performance results of this work was reported in [79] and reveal that our strategy reduces the monetary cost of SARS-CoV-2 sequence comparisons with MASA-OpenMP compared to a corresponding approach that uses only on-demand instances, meeting also the deadline even in scenarios with several spot interruptions. Finally, we show that clouds can play a fundamental role in ensuring the efficiency of the execution of such applications with reduced costs. In this section we present part of those results. Specifically, we present the monetary cost and execution time of the practical execution of EC2.

In this study, each SARS-CoV-2 pair of sequences comparison with MASA-OpenMP is a task which takes milliseconds to be executed and therefore, it is considered a *short task*. If Burst-HADS worked with those original tasks, it would be necessary to create 22,600 very short tasks, which clearly would jeopardize the execution efficiency. Thus, to use the Burst-HADS scheduling efficiently, many of these short tasks are clustered into a *supertask*. The tasks in the supertasks are executed sequentially, where each one saves its results before finishing its execution. Due to the short duration of each task, there is no need to save its checkpoint periodically. Instead, if a supertask is scheduled to a spot

VM that hibernates, the framework recovers the last recorded result to determine the last executed task just before the hibernation event. Another important point is that a task in MASA-OpenMP uses all vCPUs of its respective VM, so it is not possible to accumulate credits by using burstable VMs. Consequently, no burstable VM was used in the primary and Dynamic Scheduling Module of Burst-HADS in this evaluation, and, also, in the case of hibernations, only regular on-demand VMs and spot VMs can be used.

We restrict the family options of available instances considering those with the greatest potential for efficiency and compatibility in the execution of our application, namely: c3, c4, and c5 instances of the C family (optimized for intensive computing); m3, m4, and m5 instances of the M family (optimized for general use) and r3, r4 and r5 instances of the R family (optimized for massive data processing in memory). Among these families, we only consider instances with 4 or more vCPUs, totalizing 32 instance types. Table 9.7 shows the characteristics of those VMs. Since those tests were originally reported in [79], all monetary costs presented in this section are from November 2020.

In this section, since checkpoints are not recorded, our main concern is the service's performance and cost that will be used to store both the SARS-CoV-2 sequences and the MASA-OpenMP results. Thus, in this case, we evaluate S3 and EBS to define which service would be the best choice for those tests. To evaluate the impact of execution time, we consider the execution of one supertask with only four sequences in a c5.9xlarge instance. In this case, using EBS, the execution took only 14 seconds, while with S3, the time was 2188 seconds, representing an increment of 15528.57%. That happens because Amazon AWS does not offer native support for using S3 as a file-system. Thus, we use the S3FS-FUSE tool [67], a user-level file system that provides an interface to S3 as a POSIX compatible system [67]. However, the S3FS-FUSE has a poor performance when there are random writes or appends operations because, in this case, it is necessary to require rewriting the entire object. Thus, since it is the case of Masa-OpenMP, the use of S3 became prohibitive with that application.

We have firstly evaluated Burst-HADS in a scenario without hibernation. We compare this strategy with the situation in which SARS-CoV-2 MASA-OpenMP under the supertask model is executed following the schedule also provided by the ILS (with the same parameters presented in Section 9.1), but considering only on-demand VMs – we denoted here as On-demand strategy. These experiments aim to measure the impact spot instances have on the monetary cost of the execution. Figure 9.1 presents the average monetary cost of three executions of MASA-OpenMP with 60 supertasks, each one con-

Table 9.7: Attributes of Burst-HADS' input set VMs

Type	# vCPUs	Memory (GiB)	Price per Hour (USD)	
			On-Demand	Spot
c3.xlarge	4	7.5	0.210	0.072
c4.xlarge	4	7.5	0.199	0.076
c5.xlarge	4	8	0.170	0.073
c3.2xlarge	8	15	0.420	0.161
c4.2xlarge	8	15	0.398	0.209
c5.2xlarge	8	16	0.340	0.161
c3.4xlarge	16	30	0.840	0.350
c4.4xlarge	16	30	0.796	0.246
c5.4xlarge	16	32	0.680	0.281
c3.8xlarge	32	60	1.680	0.529
c4.8xlarge	36	60	1.591	0.609
c5.9xlarge	36	72	1.530	0.684
c5.12xlarge	48	96	2.040	0.790
c5.18xlarge	72	144	3.060	1.196
m3.xlarge	4	15	0.266	0.061
m4.xlarge	4	16	0.200	0.080
m5.xlarge	4	16	0.192	0.076
m3.2xlarge	8	30	0.532	0.129
m4.2xlarge	8	32	0.400	0.206
m5.2xlarge	8	32	0.384	0.182
m4.4xlarge	16	64	0.800	0.343
m5.4xlarge	16	64	0.768	0.335
m5.8xlarge	32	128	1.536	0.637
r3.xlarge	4	30.5	0.333	0.071
r4.xlarge	4	30.5	0.266	0.095
r5.xlarge	4	32	0.252	0.098
r3.2xlarge	8	61	0.665	0.144
r4.2xlarge	8	61	0.532	0.184
r5.2xlarge	8	64	0.504	0.221
r3.4xlarge	16	122	1.330	0.309
r4.4xlarge	16	122	1.064	0.330
r5.4xlarge	16	128	1.008	0.344

taining 376 sequences, and a deadline equal to one hour. As can be seen, in comparison to the On-demand strategy, on average, Burst-HADS reduces the monetary cost by 50.4%. Since both strategies use the same instance types and scheduling plan, the execution time was basically the same in both cases – 1277 seconds in the case of the On-demand strategy, while 1274 seconds for the Burst-HADS case.

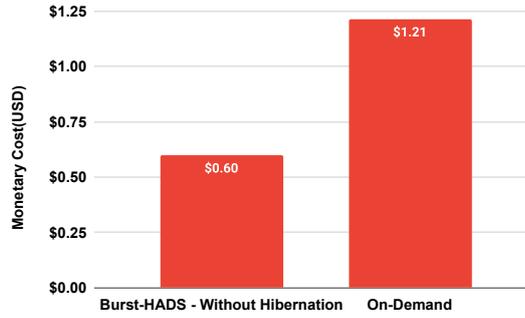


Figure 9.1: Monetary cost of a practical execution of Burst-HADS without hibernations and on-demand only strategy

Just as in the other evaluation tests, in this study, we also use Poisson distribution [1] to emulate different spot VM patterns of hibernations and resumptions. Thus, in this case, we consider three scenarios by varying k_h and k_r parameters. Table 9.8 presents those scenarios.

Table 9.8: Different execution scenarios generated by varying the parameters λ_h and λ_r

ID	<i>hibernating</i>	<i>resuming</i>	λ_h	λ_r
c_1	$k_h = 5$	$k_r = 0$	5/3600	0/3600
c_2	$k_h = 5$	$k_r = 2.5$	5/3600	2.5/3600
c_3	$k_h = 5$	$k_r = 5$	5/3600	5/3600

Table 9.9 presents the averages of three executions using Burst-HADS in each of the three execution scenarios. For each scenario, the table shows the average number of hibernations, followed by resume events. It also includes the average number of on-demand VMs launched by the Dynamic Module and the average monetary cost and makespan of the executions. Finally, the two last columns represent the percentage difference between Burst-HADS and the On-demand strategy (diff) related to the monetary cost and the makespan. As shown in the table, compared to the On-demand strategy, Burst-HADS minimized the monetary cost for the scenarios c_1 , c_2 , and c_3 , with an average reduction of 21.00% regarding the three scenarios. However, the makespan increases in all cases because whenever a spot hibernation occurs, Burst-HADS migrates tasks to other VMs,

causing additional overheads, including the time to launch new VMs. Nonetheless, in all cases, the deadline was respected.

Since c_1 has the highest hibernation level ($k_h = 5$) and no resuming event ($k_r = 0$), it is considered the worst-case scenario in our tests. As shown in Table 9.9, the monetary cost in c_1 is only 2.06% less than the On-demand strategy, since, on average, 8.33 additional on-demand VMs were launched to finish the execution before the deadline. On the other hand, c_2 shows how resuming events impact the execution’s final monetary cost, since, although the average number of hibernations was greater than c_1 (9.67 average hibernation on c_2 against 6.67 on c_1), the cost reduction was 29.75%. Finally, in scenario c_3 , we see that in the case where both the hibernation and resuming events have the same rate ($k_h = 5$ and $k_r = 5$, respectively), the framework was able to reduce the monetary cost by more than 30.00%.

Table 9.9: Comparison between Burst-HADS and On-demand strategy in terms of monetary cost and makespan in scenarios c_1 , c_2 and c_3

scenario	Burst-HADS				Diff (%)		
	# hibernation	# resuming	# on-demand	cost(\$)	makespan	cost(\$)	makespan
c_1	6.67	0.00	8.33	1.185	2978.66	2.06%	-133.25%
c_2	9.67	6.33	3.33	0.85	2785.66	29.75%	-118.141%
c_3	8.33	4.66	4.00	0.82	2350.33	32.41%	-84.0512%

Thereby, this section shows that, with MASA-OpenMP on top of Burst-HADS, we could align 22,600 SARS-CoV-2 using 24 EC2 spot instances (136 vCPUs) in 21 minutes and 23 seconds, with a monetary cost of 0.60\$ and no interruptions. If the same execution was carried out with only on-demand instances, we would have paid 1.21\$, which means Burst-HADS reduced the monetary cost by 50.4%. Moreover, we showed that although the makespan has an average increment of 111.81% in the scenarios with hibernations, the deadline was always met by Burst-HADS, with monetary cost reduced up to 32% on those cases.

Chapter 10

Conclusions and Future Work

This Chapter describes the main results and contributions of this work. In Section 10.1, we highlight the main contributions of this thesis, while we point out in Section 10.2 some interesting directions for future research in this topic area.

10.1 Concluding Remarks

Unlike the advertising of cloud providers who advocate the ease of use of cloud environments as one of the main advantages, when considering all the necessary variables to be defined to execute a given application efficiently, a cloud can become a complex environment where any decision impacts directly in the final execution and respective monetary costs. In this sense, many works have proposed tools to help the management decisions. HADS and Burst-HADS are some of these tools, and they offer additional contributions regarding its architecture and up-to-date VM instance classes. The frameworks have the advantage of being modular, lightweight, and can be easily upgraded to meet new requirements. Moreover, they include a series of built-in functions such as load balance, native checkpoint, and recovery procedures. Unlike other frameworks that cope with the termination of spot VMs, HADS and Burst-HADS explores the hibernation feature of spot VMs to minimize execution monetary costs and, in the case of Burst-HADS, the burst capacity of burstable instances to minimize the execution time of the application.

The proposed strategies were evaluated using the VMs of AWS with real executions of synthetic applications and a real embarrassingly distributed application from the NAS benchmark, considering several emulated scenarios with different spot VM hibernation and resuming rates. Our results confirm the effectiveness of HADS and Burst-HADS in terms of monetary costs and that both of them avoid temporal failures even in the presence

of multiple hibernations. Furthermore, they show that the resuming rate of hibernated spot VMs has an impact on monetary costs.

Compared to the exact approach, the results also show that Burst-HADS ILS-based heuristic also reaches the optimal solutions but in a much shorter time (less than one second on average) than the latter. Furthermore, when compared to baseline heuristics, the ILS presents more balanced solutions considering both objectives. It reduces the average monetary cost by more than 38% with an increment of the makespan of 18% when compared to MinMin. Finally, ILS reduces the makespan by more than 80%, with a small increment of the monetary cost; 6.78% and 13.14% when respectively compared to MaxMin and the Greedy approaches. In all cases, the percentage loss in one objective was smaller than the corresponding gain in the other one.

Compared to the ILS On-demand approach that uses only regular on-demand VMs, Burst-HADS reduces the monetary cost for all execution scenarios at the expense of slightly longer makespans due to the migration overhead. Moreover, compared to HADS, Burst-HADS reduces the makespan by more than 25%, with an average increase of only 1.92% in the monetary cost. Finally, our case study also demonstrates the benefits of Burst-HADS, which reduces the monetary cost of running SARS-CoV-2 MASA-OpenMP compared with the On-demand strategy, by 50.4% if no spot hibernation occurs, and up to 32% when hibernation happens.

Moreover, in this work, we also evaluate the storage services EBS, S3, and EFS offered by AWS in the context of checkpoint and recovery operations. Thus, To characterize the overheads associated with the checkpoint recording, a series of practical tests were conducted in EC2. Those tests showed that the local storage service EBS presented the best performance in relation to dump time, followed closely by EFS, while S3 had the largest dump times. Those results are confirmed by the impact on the total execution time caused by the checkpoint recording in each of the evaluated storage services. However, the estimation of the monetary costs considering the checkpointing overhead in a long-running task showed that S3 could be a very attractive approach when the monetary cost is more important than the application's total execution time.

10.2 Future Work

In this section, we present some promising future directions derived from the contributions of this thesis.

- **Evaluation of the impact of the size of the ACs:** Our results show that the AC's size impacts the frameworks' performance, especially in the work-stealing procedure. However, in our experiments, we do not conduct a deep study to evaluate how the Allocation Cycles size would influence the framework's decisions and in the final monetary cost and makespan. Thus, conducting new experiments with different ACs values is an important step to improving the frameworks' performance.
- **Developing of a strategy considering no previous knowledge about the applications' characteristics:** Currently, both frameworks require input information about each task of the BoT application. Specifically, the frameworks depend on the task's execution time and memory requirements for all scheduling and migration decisions. Although that approach simplifies the development of the scheduling heuristic, there are some drawbacks to using it. For example, it can be tough for the user to have all the task information, and the user can give bad-quality information, which would jeopardize the quality of the solutions. Moreover, the environment can suffer performance variation during the execution. Thus, it is important to develop new techniques to define tasks' characteristics and update those characteristics on the run.
- **Extending of the frameworks to other classes of deadline constraint applications:** HADS and Burst-HADS could be used in other classes of applications. Some applications, such as scientific workflows, have dependence between their tasks. In this case, the frameworks' scheduling and migration decisions need to be adapted. Besides, other applications model also bring the potential to create and study new approaches and heuristics.
- **Inclusion of prediction approaches:** All decisions of migration are currently reactive. However, the migration procedure could be improved with some predictions model. For example, some predictions of the spot market's future state, considering VMs' hibernation history, could be used to avoid the migration of tasks to spot VMs with a high chance of hibernating in the next minutes.

- **Smart selection of VM types:** The current VM type selection is based on two factors: the execution time and the monetary cost. But, other factors can influence that decision in the cloud, for example, the trade-off between price and VM performance in different execution scenarios. Moreover, the degradation of VMs performance and the hibernation rates can be used as selection criteria. In particular, a deeper analysis needs to be performed about burstable VMs. As those VMs also have a CPU credit value associate, the choice for the most suitable burstable VMs for a specific application should consider, among other things, how many CPU credits a specific Burstable VM type earns per second.
- **Evaluation of other checkpoint approaches:** In this work, we have used and evaluated only uncoordinated checkpoints that record the checkpoint files directly on the storage services' file system. However, there are several checkpoint recording approaches in the literature. Among those approaches, in particular, the two-step asynchronous recording, where initially the checkpoint is kept in the VM memory and then it is written in the storage system, have an interesting potential to be combined with the S3 services in the case of applications with a huge memory footprint which causes a high dump time. Another point is the use of a fixed time interval between checkpoints. Currently, that interval is defined based on an input parameter defined by the user. But, some formulations could be used to define better intervals without the need for any parameters defined by the user.
- **Inclusion of support to other cloud providers:** Several development decisions of HADS and Burst-HADS were made considering future support to other cloud providers. Thus, as stated in Chapters 1 and 4, both frameworks can be easily extended to support other cloud providers. Moreover, resources from different providers could also be combined by the framework in a multi-cloud environment.

Bibliography

- [1] AHRENS, J. H.; DIETER, U. Computer methods for sampling from gamma, beta, poisson and binomial distributions. *Computing* 12, 3 (1974), 223–246.
- [2] ALI, A.; PINCIROLI, R.; YAN, F.; SMIRNI, E. Cedule: A scheduling framework for burstable performance in cloud computing. In *IEEE International Conference on Autonomic Computing (ICAC)* (2018), pp. 141–150.
- [3] ALVES, M. M.; DE ASSUMPTIÃO DRUMMOND, L. M. A multivariate and quantitative model for predicting cross-application interference in virtual environments. *Journal of Systems and Software* 128 (2017), 150 – 163.
- [4] ALVES, M. M.; DRUMMOND, L. M. A multivariate and quantitative model for predicting cross-application interference in virtual environments. *Journal of Systems and Software* 128 (2017), 150 – 163.
- [5] ARMBRUST, M.; FOX, A.; GRIFFITH, R.; JOSEPH, A. D.; KATZ, R.; KONWINSKI, A.; LEE, G.; PATTERSON, D.; RABKIN, A.; STOICA, I.; ZAHARIA, M. A view of cloud computing. *Commun. ACM* 53, 4 (Apr. 2010), 50–58.
- [6] AUPY, G.; BENOIT, A.; MELHEM, R. G.; RENAUD-GOUD, P.; ROBERT, Y. Energy-aware checkpointing of divisible tasks with soft or hard deadlines. In *International Green Computing Conference, IGCC 2013, Arlington, VA, USA, June 27-29, 2013, Proceedings* (2013), pp. 1–8.
- [7] AWS. Amazon EC2 Spot Lets you Pause and Resume Your Workloads. <https://aws.amazon.com/about-aws/whats-new/2017/11/amazon-ec2-spot-lets-you-pause-and-resume-your-workloads/>, 2017. Accessed in February 2021.
- [8] AWS. Amazon Elastic Block Store. <https://aws.amazon.com/ebs/>, 2020. Accessed in February 2021.
- [9] AWS. Amazon Elastic Compute Cloud. <https://aws.amazon.com/ec2/features/>, 2020. Accessed in February 2021.

- [10] AWS. Amazon Elastic File System. <https://aws.amazon.com/efs/>, 2020. Accessed in February 2021.
- [11] AWS. Amazon S3. <https://aws.amazon.com/s3/>, 2020. Accessed in February 2021.
- [12] AWS. Amazon Web Services. <https://aws.amazon.com/>, 2020. Accessed in February 2021.
- [13] AWS. Boto 3 Documentation. <https://boto3.amazonaws.com/v1/documentation/api/latest/index.html>, 2020. Accessed in February 2021.
- [14] AWS. Burstable performance instance requirements. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html>, 2020. Accessed in February 2021.
- [15] AWS. Cloud Storage with AWS. <https://aws.amazon.com/products/storage>, 2020. Accessed in February 2021.
- [16] AWS. Instance Types. <https://aws.amazon.com/ec2/instance-types/>, 2020. Accessed in February 2021.
- [17] AWS. Working with Amazon S3 Objects. <https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingObjects.html>, 2020. Accessed in February 2021.
- [18] AZURE, M. Microsoft Azure. <https://azure.microsoft.com/>, 2020. Accessed in February 2021.
- [19] BAARZI, A. F.; ZHU, T.; URGONKAR, B. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *ACM Symposium on Cloud Computing* (2019), pp. 126–138.
- [20] BAILEY, D.; HARRIS, T.; SAPHIR, W.; VAN DER WIJNGAART, R.; WOO, A.; YARROW, M. The nas parallel benchmarks 2.0. Tech. rep., Technical Report NAS-95-020, NASA Ames Research Center, 1995.
- [21] BARR, J. New – Per-Second Billing for EC2 Instances and EBS Volumes. <https://aws.amazon.com/pt/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/>, 2017. Accessed in February 2021.

- [22] CHAKRAVARTHI, K. K.; SHYAMALA, L.; VAIDEHI, V. Budget aware scheduling algorithm for workflow applications in iaas clouds. *Cluster Computing* (2020), 1–15.
- [23] CHHABRA, A.; SINGH, G.; KAHN, K. S. Multi-criteria hpc task scheduling on iaas cloud infrastructures using meta-heuristics. *Cluster Computing* (2020), 1–34.
- [24] CLOUD, G. Google Cloud. <https://cloud.google.com/>, 2020. Accessed in February 2021.
- [25] DALY, J. T. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future generation computer systems* 22, 3 (2006), 303–312.
- [26] DELDARI, A.; SALEHAN, A. A survey on preemptible iaas cloud instances: challenges, issues, opportunities, and advantages. *Iran Journal of Computer Science* (2020), 1–24.
- [27] DIKAIAKOS, M. D.; KATSAROS, D.; MEHRA, P.; PALLIS, G.; VAKALI, A. Cloud computing: Distributed internet computing for it and scientific research. *IEEE Internet Computing* 13, 5 (Sept 2009), 10–13.
- [28] DONGARRA, J. J.; LUSZCZEK, P.; PETITET, A. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.
- [29] EMELYANOV, P. Criu: Checkpoint/restore in userspace, july 2011. URL: <https://criu.org> (2011).
- [30] FABRA, J.; EZPELETA, J.; ÁLVAREZ, P. Reducing the price of resource provisioning using ec2 spot instances with prediction models. *Future Generation Computer Systems* 96 (2019), 348–367.
- [31] FAKHFAKH, F.; KACEM, H. H.; KACEM, A. H. Workflow scheduling in cloud computing: A survey. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations* (Sept 2014), pp. 372–378.
- [32] FARAHABADY, M. H.; LEE, Y. C.; ZOMAYA, A. Y. Non-clairvoyant assignment of bag-of-tasks applications across multiple clouds. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies* (2012), IEEE, pp. 423–428.
- [33] GCLOUD. Google APP Engine. <https://cloud.google.com/appengine/>, 2021. Accessed in February 2021.

- [34] GHOBAEI-ARANI, M.; SOURI, A.; SAFARA, F.; NOROUZI, M. An efficient task scheduling approach using moth-flame optimization algorithm for cyber-physical system applications in fog computing. *Transactions on Emerging Telecommunications Technologies* 31, 2 (2020), e3770.
- [35] GODER, A.; SPIRIDONOV, A.; WANG, Y. Bistro: Scheduling data-parallel jobs against live production systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 459–471.
- [36] GOIRI, I.; JULIÀ, F.; GUITART, J.; TORRES, J. Checkpoint-based fault-tolerant infrastructure for virtualized service providers. In *IEEE/IFIP Network Operations and Management Symposium, NOMS 2010, 19-23 April 2010, Osaka, Japan* (2010), pp. 455–462.
- [37] GOOGLE. Gmail. <https://mail.google.com/>, 2021. Accessed in February 2021.
- [38] GOOGLE. Google Docs. <https://docs.google.com/>, 2021. Accessed in February 2021.
- [39] GUROBI OPTIMIZATION, L. Gurobi optimizer reference manual, 2021.
- [40] GUTIERREZ-GARCIA, J. O.; SIM, K. M. A family of heuristics for agent-based elastic cloud bag-of-tasks concurrent scheduling. *Future Generation Computer Systems* 29, 7 (2013), 1682–1699.
- [41] HASHEM, I. A. T.; YAQOUB, I.; ANUAR, N. B.; MOKHTAR, S.; GANI, A.; KHAN, S. U. The rise of “big data” on cloud computing: Review and open research issues. *Information Systems* 47 (2015), 98 – 115.
- [42] HEROKUS. Heroku. <https://www.heroku.com/>, 2021. Accessed in February 2021.
- [43] HOFFA, C.; MEHTA, G.; FREEMAN, T.; DEELMAN, E.; KEAHEY, K.; BERRIMAN, B.; GOOD, J. On the use of cloud computing for scientific workflows. In *2008 IEEE Fourth International Conference on eScience* (Dec 2008), pp. 640–645.
- [44] HUANG, X.; LI, C.; CHEN, H.; AN, D. Task scheduling in cloud computing using particle swarm optimization with time varying inertia weight strategies. *Cluster Computing* (2019), 1–11.

- [45] JIANG, Y.; SHAHRAD, M.; WENTZLAFF, D.; TSANG, D. H.; JOE-WONG, C. Burstable instances for clouds: Performance modeling, equilibrium analysis, and revenue maximization. In *IEEE INFOCOM Conference on Computer Communications* (2019), pp. 1576–1584.
- [46] JUVE, G.; DEELMAN, E.; VAHI, K.; MEHTA, G.; BERRIMAN, B.; BERMAN, B. P.; MAECHLING, P. Scientific workflow applications on amazon ec2. In *2009 5th IEEE International Conference on E-Science Workshops* (Dec 2009), pp. 59–66.
- [47] KALRA, M.; SINGH, S. A review of metaheuristic scheduling techniques in cloud computing. *Egyptian Informatics Journal* 16, 3 (2015), 275 – 295.
- [48] KATEVENIS, M.; SIDIROPOULOS, S.; COURCOUBETIS, C. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on selected Areas in Communications* 9, 8 (1991), 1265–1279.
- [49] KESHANCHI, B.; SOURI, A.; NAVIMIPOUR, N. J. An improved genetic algorithm for task scheduling in the cloud environments using the priority queues: formal verification, simulation, and statistical testing. *Journal of Systems and Software* 124 (2017), 1–21.
- [50] KUMAR, D.; BARANWAL, G.; RAZA, Z.; VIDYARTHI, D. P. A survey on spot pricing in cloud computing. *Journal of Network and Systems Management* 26, 4 (2018), 809–856.
- [51] LEITNER, P.; SCHEUNER, J. Bursting with possibilities—an empirical study of credit-based bursting cloud instance types. In *IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)* (2015), pp. 227–236.
- [52] LOURENÇO, H. R.; MARTIN, O. C.; STÜTZLE, T. Iterated local search. In *Handbook of metaheuristics*. Springer, 2003, pp. 320–353.
- [53] LU, S.; LI, X.; WANG, L.; KASIM, H.; PALIT, H. N.; HUNG, T.; LEGARA, E. F. T.; LEE, G. K. K. A dynamic hybrid resource provisioning approach for running large-scale computational applications on cloud spot and on-demand instances. In *19th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2013, Seoul, Korea, December 15-18, 2013* (2013), pp. 657–662.
- [54] LU, Y.; SUN, N. An effective task scheduling algorithm based on dynamic energy management and efficient resource utilization in green cloud computing environment. *Cluster Computing* 22, 1 (2019), 513–520.

- [55] MARLETTA, A. cpu limit tool. Available: <http://cpulimit.sourceforge.net/>.
- [56] MASDARI, M.; VALIKARDAN, S.; SHAHI, Z.; AZAR, S. I. Towards workflow scheduling in cloud computing: A comprehensive analysis. *Journal of Network and Computer Applications* 66 (2016), 64 – 82.
- [57] MELL, P.; GRANCE, T., ET AL. The nist definition of cloud computing.
- [58] MENACHE, I.; SHAMIR, O.; JAIN, N. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014*. (2014), pp. 177–187.
- [59] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.
- [60] NOVET, J. Amazon’s cloud division reports 28% revenue growth; AWS head Andy Jassy to succeed Bezos as Amazon CEO. <https://www.cnbc.com/2021/02/02/aws-earnings-q4-2020.html>, 2021. Accessed in February 2021.
- [61] OPRESCU, A.-M.; KIELMANN, T. Bag-of-tasks scheduling under budget constraints. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science* (2010), IEEE, pp. 351–359.
- [62] OVERLEAF. Overleaf. <https://overleaf.com>, 2021. Accessed in February 2021.
- [63] PALANKAR, M. R.; IAMNITCHI, A.; RIPEANU, M.; GARFINKEL, S. Amazon s3 for science grids: A viable solution? In *Proceedings of the 2008 International Workshop on Data-Aware Distributed Computing* (New York, NY, USA, 2008), DADC '08, Association for Computing Machinery, p. 55–64.
- [64] PARY, R. New Amazon EC2 Spot pricing model: Simplified purchasing without bidding and fewer interruptions. <https://aws.amazon.com/pt/blogs/compute/new-amazon-ec2-spot-pricing/>, 2017. Accessed in February 2021.
- [65] PEZOA, F.; REUTTER, J. L.; SUAREZ, F.; UGARTE, M.; VRGOČ, D. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web* (2016), pp. 263–273.
- [66] PHAM, T.-P.; FAHRINGER, T. Evolutionary multi-objective workflow scheduling for volatile resources in the cloud. *IEEE Transactions on Cloud Computing* (2020).

- [67] RIZUN, R. S3fs: Fuse-based file system backed by amazon s3. <https://github.com/s3fs-fuse/s3fs-fuse/>, 2010.
- [68] RUIZ-ALVAREZ, A.; HUMPHREY, M. An automated approach to cloud storage service selection. In *Proceedings of the 2nd International Workshop on Scientific Cloud Computing* (New York, NY, USA, 2011), ScienceCloud '11, Association for Computing Machinery, p. 39–48.
- [69] SANDES, E. F. O.; MIRANDA, G.; MARTORELL, X.; AYGAUDE, E.; TEODORO, G.; MELO, A. C. M. A. Masa: A multiplatform architecture for sequence aligners with block pruning. *ACM Transactions on Parallel Computing* 2, 4 (2016).
- [70] SERVICES, A. W. AWS Elastic Beanstalk. <https://aws.amazon.com/elasticbeanstalk/>, 2021. Accessed in February 2021.
- [71] SHARMA, P.; LEE, S.; GUO, T.; IRWIN, D. E.; SHENOY, P. J. Spotcheck: designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015* (2015), pp. 16:1–16:15.
- [72] SUBRAMANYA, S.; GUO, T.; SHARMA, P.; IRWIN, D. E.; SHENOY, P. J. Spoton: a batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015* (2015), pp. 329–341.
- [73] TANG, X.; LIAO, X.; ZHENG, J.; YANG, X. Energy efficient job scheduling with workload prediction on cloud data center. *Cluster Computing* 21, 3 (2018), 1581–1593.
- [74] TEYLO, L.; ARANTES, L.; SENS, P.; DRUMMOND, L. M. A bag-of-tasks scheduler tolerant to temporal failures in clouds. In *31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (2019), pp. 144–151.
- [75] TEYLO, L.; ARANTES, L.; SENS, P.; DRUMMOND, L. M. A hibernation aware dynamic scheduler for cloud environments. In *48th International Conference on Parallel Processing: Workshops* (2019), ACM, p. 24.
- [76] TEYLO, L.; ARANTES, L.; SENS, P.; DRUMMOND, L. M. A dynamic task scheduler tolerant to multiple hibernations in cloud environments. *Cluster Computing* (2020), 1–23.

- [77] TEYLO, L.; ARANTES, L.; SENS, P.; DRUMMOND, L. M. D. A. Scheduling bag-of-tasks in clouds using spot and burstable virtual machines. *arXiv preprint arXiv:2011.05042* (2020).
- [78] TEYLO, L.; BRUM, R. C.; ARANTES, L.; SENS, P.; DRUMMOND, L. M. D. A. Developing checkpointing and recovery procedures with the storage services of amazon web services. In *49th International Conference on Parallel Processing-ICPP: Workshops* (2020), pp. 1–8.
- [79] TEYLO, L.; NUNES, A. L.; MELO C. M. A., A.; BOERES, C.; DE A. DRUMMOND, L. M.; F., M. N. Comparing sars-cov-2 sequences using a commercial cloud with a spot instance based dynamic scheduler. In *CCGRID* (2021). Accepted Paper.
- [80] THAI, L.; VARGHESE, B.; BARKER. Task scheduling on the cloud with hard constraints. In *2015 IEEE World Congress on Services, SERVICES 2015, New York City, NY, USA, June 27 - July 2, 2015* (2015), pp. 95–102.
- [81] THAI, L.; VARGHESE, B.; BARKER, A. Executing bag of distributed tasks on the cloud: Investigating the trade-offs between performance and cost. In *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014, Singapore, December 15-18, 2014* (2014), pp. 400–407.
- [82] THAI, L.; VARGHESE, B.; BARKER, A. A survey and taxonomy of resource optimisation for executing bag-of-task applications on public clouds. *Future Generation Comp. Syst.* 82 (2018), 1–11.
- [83] TOPCUOGLU, H.; HARIRI, S.; WU, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (Mar 2002), 260–274.
- [84] TSAI, C. W.; RODRIGUES, J. J. P. C. Metaheuristic scheduling for cloud: A survey. *IEEE Systems Journal* 8, 1 (March 2014), 279–291.
- [85] ULLMAN, J. D. Np-complete scheduling problems. *Journal of Computer and System sciences* 10, 3 (1975), 384–393.
- [86] VAQUERO, L. M.; RODERO-MERINO, L.; CACERES, J.; LINDNER, M. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.* 39, 1 (Dec. 2008), 50–55.

-
- [87] VARSHNEY, P.; SIMMHAN, Y. Autobot: Resilient and cost-effective scheduling of a bag of tasks on spot vms. *IEEE Trans. Parallel Distrib. Syst.* 30, 7 (2019), 1512–1527.
- [88] WANG, C.; URGONKAR, B.; GUPTA, A.; KESIDIS, G.; LIANG, Q. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Twelfth European Conference on Computer Systems* (2017), pp. 620–634.
- [89] YAO, M.; ZHANG, P.; LI, Y.; HU, J.; LI, C.; LI, X. Cutting your cloud computing cost for deadline-constrained batch jobs. In *2014 IEEE International Conference on Web Services, ICWS, 2014, Anchorage, AK, USA, June 27 - July 2, 2014* (2014), pp. 337–344.
- [90] YI, S.; ANDRZEJAK, A.; KONDO, D. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *IEEE Transactions on Services Computing* 5, 4 (2011), 512–524.
- [91] YU, J.; BUYYA, R.; RAMAMOHANARAO, K. *Workflow Scheduling Algorithms for Grid Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 173–214.
- [92] ZHOU, A. C.; HE, B.; LIU, C. Monetary cost optimizations for hosting workflow-as-a-service in iaas clouds. *IEEE transactions on cloud computing* 4, 1 (2015), 34–48.

APPENDIX A - Published Papers

- Costa, M.; **Teylo, L.**; Drummond, L. Avaliação da Migração Vertical na Amazon Web Services. *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho*, 2017 (Best Paper Award - WSCAD-IC).
- Cieza, E.; **Teylo, L.**; Frota, Y.; Drummond, L.; Bentes, C. A GPU-based Meta-heuristic for Workflow Scheduling on Clouds. *13th International Meeting on High Performance Computing for Computational Science*, VECPAR. 2018.
- Melo, M. A.; **Teylo, L.**; Frota, Y.; Drummond, L. An Interference-Aware Strategy for Co-locating High Performance Computing Applications in Clouds. *High Performance Computing Systems 19th Symposium, WSCAD 2018*, Revised Selected Papers. 19ed.: Springer International Publishing, 2018.
- **Teylo, L.**; Arantes, L.; Sens, P.; Drummond, L. A Bag-of-Task Scheduler Tolerant to Temporal Failures in Clouds. *IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2019.
- **Teylo, L.**; Arantes, L.; Sens, P.; Drummond, L. A Hibernation Aware Dynamic Scheduler for Cloud Environments. *15th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, ICPP, 2019.
- **Teylo, L.**; Arantes, L.; Sens, P.; Drummond, L. A dynamic task scheduler tolerant to multiple hibernations in cloud environments. *Cluster Computing*, v. 4, p. 1-23, 2020 (Available Online).
- **Teylo, L.**; Brum, R.; Arantes, L.; Sens, P.; Drummond, L. Developing Checkpointing and Recovery Procedures with the Storage Services of Amazon Web Services. *16th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, ICPP, 2020.

- **Teylo, L.;** Nunes, A. L.; Melo C. M. A., A.; Boeres, C.; Drummond, L.; Martins, N. Comparing sars-cov-2 sequences using a commercial cloud with a spot instance based dynamic scheduler. *21th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2021 (Accepted Paper).