

A dynamic task scheduler tolerant to multiple hibernations in cloud environments

Luan Teylo, Luciana Arantes, Pierre Sens, Lucia M A Drummond

▶ **To cite this version:**

Luan Teylo, Luciana Arantes, Pierre Sens, Lucia M A Drummond. A dynamic task scheduler tolerant to multiple hibernations in cloud environments. Cluster Computing, 2020, 10.1007/s10586-020-03175-2 . hal-03136616

HAL Id: hal-03136616
https://inria.hal.science/hal-03136616

Submitted on 9 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

A Dynamic Task Scheduler Tolerant to Multiple Hibernations in Cloud Environments

Luan Teylo · Luciana Arantes · Pierre Sens · Lucia M. A. Drummond

Received: date / Accepted: date

Abstract Cloud platforms usually offer several types of Virtual Machines (VMs) with different guarantees in terms of availability and volatility, provisioning the same resource through multiple pricing models. For instance, in the Amazon EC2 cloud, the user pays per use for on-demand VMs while spot VMs are instances available at lower prices. However, a spot VM can be terminated or hibernated by EC2 at any moment. In this work, we propose the Hibernation-Aware Dynamic Scheduler (HADS) that schedules Bag-of-Tasks (BoT) applications with deadline constraints in both hibernation prone spots VMs and on-demand VMs. HADS aims at minimizing the monetary costs of executing BoT applications on Clouds ensuring that their deadlines are respected even in the presence of multiple hibernations. Results collected from experiments on Amazon EC2 VMs using synthetic applications and a NAS benchmark application show the effectiveness of HADS in terms of monetary costs when compared to on-demand VM only solutions.

Keywords Cloud Computing · Dynamic BoT Scheduling · Temporal Failures · Spot VM Hibernation · Monetary cost minimization

1 Introduction

In the past few years, cloud computing has emerged as an attractive option to run different classes of applications due to several advantages over other platforms, such as: (i) immediate access to computational resources, (ii) no upfront capital investments, and (iii) pay-per-use model. Some cloud providers offer several classes of Virtual Machines (VMs) with different guarantees in terms of availability and volatility, provisioning the same resource through multiple pricing models. Amazon EC2, for example, offers VMs in two main markets: on-demand and spot.

On-demand VMs can be deployed at any time, offering high availability since they cannot be interrupted by Amazon provider while allocated by a user. On the other hand, spot VMs are unused EC2 resources with a huge discount (according to Amazon the discount can be up to 90% when compared to on-demand prices) but can be revoked and terminated by Amazon whenever it requires the resources back.

Since December 2017, Amazon EC2 has defined a VM allocation policy where spot prices are more stable and with little differences over the days, i.e., they do not vary according to users' resource requests demand [23]. Furthermore, Amazon EC2 has introduced the spot VM hibernation feature that hibernates a spot VM instead of terminating it definitively [4]. When a VM hibernates, its memory and context are saved. In the case of on-demand VMs, only the user can decide to hibernate/resume a VM while, in the case of spot VMs, Amazon hibernates a given VM when it needs its resources and resumes it when cloud resource demands decrease. When a VM is resumed, all tasks that were executing at the moment of the hibernation restart from that point. Note that the user is not charged for the period that the VM keeps hibernated.

We are particularly interested in Bag-of-Task (BoT) applications executing in Clouds. This type of application is composed of independent tasks which can be ex-

Luan Teylo, Lucia M. A. Drummond
Instituto de Computação - Universidade Federal Fluminense, Niterói - Brazil
E-mail: {luanteylo,lucia}@ic.uff.br

Luciana Arantes, Pierre Sens
Sorbonne Université, CNRS, INRIA, LIP6 - France
E-mail: {luciana.arantes, pierre.sens}@lip6.fr

ecuted in any order and in parallel. In addition, we consider that the BoT applications may require deadline-bounds where the correctness on 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 in a heterogeneous environment is a well known NP-hard problem which makes that an even more challenge problem [12, 8].

Therefore, we propose in this paper the Hibernated-Aware Dynamic Scheduler, HADS, a dynamic cloud scheduler for deadline constraint Bag-of-Tasks applications, that minimizes 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. HADS avoids the problem of temporal failures, by firstly creating a primary scheduling map of tasks to spot and on-demand VMs and then executing a dynamic scheduling module that applies task migration and work-stealing techniques, whenever necessary. When mapping a task to a spot VM, the primary scheduler considers a maximum time limit for the application makespan which guarantees that in case of hibernation of a spot VM there will be enough spare time to migrate the running tasks and those not yet executed to other VMs, avoiding, in this way, a temporal failure. In addition, in case of executing in spot VMs, tasks periodically save their states in a stable storage and, upon migration, the execution of the currently running task is re-started in the new selected VM from the last checkpoint.

A preliminary version of HADS was presented in [28]. However, in that paper, we considered that a spot VM could hibernate just once and, upon migration, the task execution re-started from the beginning even if it had already executed part of its code before hibernation. In the current version, HADS tolerates that spot VMs hibernate multiple times and the implementation of an uncoordinated checkpointing approach allows tasks to save their states in a stable storage and, therefore, to recover from the last checkpoint, in case of task migration. This new feature reduces the impact in the execution time of the application when task migration takes place. In the previous work, the experiments were only simulated, while in the current one they were conducted in Amazon EC2, using S3 for storage. Furthermore, different scenarios were considered where spot VMs hibernate and resume several times and, periodically, save their states by taking checkpoints.

Therefore, the main contributions of this work are threefold:

- a scheduler of BoT applications tolerant to temporal failure that supports multiple spot VM hibernation events;
- an uncoordinated checkpointing mechanism that reduces the impact in the application execution time, in case of task migration;
- analysis of the proposed strategy in a real cloud provider environment, Amazon EC2 with S3 for storage of checkpoints, considering different scenarios with spot VM hibernation and resuming events.

Results from the experiments using synthetic applications [2] and a NAS benchmark application [5], running in both hibernated-prone spot and on-demand VMs, confirm the effectiveness of HADS in terms of monetary costs when compared to an approach based only on Amazon EC2 on-demand VMs. They also show that HADS avoids temporal failures, even in the presence of multiple spot VM hibernation events, and that the checkpoint/recover strategy is able to reduce the impact in the application execution time, in case of task migration.

The remaining of this paper is organized as follows. Section 2 discusses some related work. Section 3 defines the system and application models, while Section 4 presents HADS and its algorithms. Evaluation results from experiments on EC2 cloud are presented in Section 5. Finally, Section 6 concludes the paper and introduces some future directions.

2 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. Thus, the heterogeneity and elasticity of clouds were extensively studied and explored in the context of BoT scheduling. 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 [21].

Table 1 summarizes the main characteristics of some works that consider the scheduling of BoT applications in clouds and HADS. The following features are highlighted in the table: Spot and On-demand, which indicates if the solution considers the spot and on-demand

markets; hibernate/resume, which shows if hibernation-prone VMs are used; the applied fault tolerance technique; the objectives and constraints of the scheduling algorithm; and how the proposed scheduler is evaluated.

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.* [30], for example, propose a scheduler that satisfies a deadline, and that minimizes the monetary cost, by using only on-demand and reserved VMs. In [14], an agent-based strategy that uses different heuristics to schedule concurrent BoT applications to on-demand VMs is presented. Farahadaby *et al.* [11] propose FPRAS, a scheduler algorithm for BoT applications in multi-cloud environments whose objective is to minimize the monetary cost of the execution. In [17], 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.* [15], the authors aimed to minimize the total execution time of BoT applications executed in on-demand VMs, by developing a PSO-based scheduler. In [22], the authors present BaTs, a budget-constrained scheduler that uses on-demand VMs to execute BoT applications. In [7], 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 scheduling problems. In [26], 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 [8], 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 [20], 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 [3], 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.

Likewise to HADS, several works in the related literature [25, 21, 24, 19, 29, 27, 28] propose, for monetary cost sake, the use, whenever possible, of spot VMs for scheduling tasks. 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. Consequently, contrary to HADS, they do not tackle the problem of temporal failures for applications with deadline constraints. The common objective of them is rather a tradeoff between monetary cost, reliability, and execution time.

In [25], Subramanya *et al.* present SpotOn, a batch computing service platform that uses checkpointing, migration, and replication to mitigate the impact of spot VMs revocations. SpotOn uses the price history of the spot market to select the fault-tolerance mechanism that minimizes the expected monetary cost of job execution. Loo *et al.* [19] propose a hybrid approach that considers on-demand VMs for high priority tasks and spot VMs for non-priority ones. In order to tolerate spot VMs interruptions, a certain number of on-demand VMs are reserved as spare resources to execute backup tasks. Whenever spot instances are terminated, the workload is immediately migrated to on-demand VMs.

SpotCheck [24] uses nested VMs within spot VMs to provide the illusion of a platform that offers always-available VMs. The nested VMs are transparently migrated to an on-demand VM when a spot revocation is detected. In [21], an online learning algorithm, which selects spot and on-demand VMs to execute batch jobs that arrive over time, is proposed. The algorithm dynamically adapts the resource allocation by learning from its performance on prior job executions and from the history of spot prices. The solution is able to switch to on-demand resources whenever there are not enough available spot VMs to ensure the desired performance.

The majority of the above works exploit the historical of spot VM price variation to predict spot VMs' revocations. However, with the new prices model announced in December 2017 by Amazon, prices of VMs in the spot market are quite stable. According to Ama-

Table 1: Main characteristics of both HADS and some works related to the scheduling problem of BoT applications in clouds.

Article	Spot	On-demand	hibernate/resume	Fault Tolerance	Objective (minimize)	Constraint	Evaluation Approach
HADS	Yes	Yes	Yes	Checkpoint and Migration	Monetary Cost	Deadline	Prototype on EC2
Oprescu and Kielmann [22] (2010)	No	Yes	No	-	Execution Time	Budget	Simulation
Yi *et al.* [31] (2011)	Yes	No	No	Checkpoint and Migration	Monetary Cost	-	Simulation
Farahadaby *et al.* [11] (2012)	No	Yes	No	-	Monetary Cost	-	Simulation
Lu *et al.* [19] (2013)	Yes	Yes	No	Migration	Monetary Cost	-	Simulation
Gutierrez-Garcia *et al.* [14] (2013)	No	Yes	No	-	Monetary Cost	-	Simulation
Aupy *et al.* [3] (2013)	No	Yes	No	Checkpoint	Energy Consumption and Execution Time	Deadline	Simulation
Yao *et al.* [30] (2014)	No	Yes	No	-	Monetary Cost	Deadline	Simulation
Menache *et al.* [21] (2014)	Yes	Yes	No	Migration	Monetary Cost	Deadline	Simulation
Subramanya *et al.* [25] (2015)	Yes	Yes	No	Checkpoint, Migration and Replication	Monetary Cost	-	Simulation and Prototype on EC2
Sharma *et al.* [24] (2015)	Yes	Yes	No	Migration	Monetary Cost	-	Prototype on EC2
Keshanchi *et al.* [17] (2017)	No	Yes	No	-	Execution Time	-	Simulation
Tang *et al.* [26] (2018)	No	Yes	No	-	Energy Consumption	-	Simulation
Teylo *et al.* [27] (2019)	Yes	Yes	Yes	Migration	Monetary Cost	Deadline	Simulation
Varshney and Simmhan [29] (2019)	Yes	Yes	No	Checkpoint and Migration	Monetary Cost	Deadline	Simulation and Prototype on EC2
Huang *et al.* [15] (2019)	No	Yes	No	-	Execution Time	-	Simulation
Lu and Sun [20] (2019)	No	Yes	No	-	Energy Consumption	-	Simulation
Chakravarthi *et al.* [8] (2020)	No	Yes	No	-	Execution Time	Budget	Simulation
Chhabra *et al.* [8] (2020)	No	Yes	No	-	Energy Consumption and Execution Time	-	Simulation

zon EC2, the new price model is determined exclusively by the supply and demand for Amazon EC2 spare capacity and no more by bid prices as previously [23]. Therefore, it is no longer possible to predict the revocation/termination of spot VMs based only on the history of price variations.

Aiming at coping with spot VM revocations, in [27], Teylo *et al.* proposed a static heuristic that creates pre-defined backup maps, i.e., before the execution of the job tasks themselves. Such a heuristic was the first attempt to handle the hibernation of spot VMs, and results obtained by simulation showed that the hibernation problem is better handled with a dynamic approach. Thus, in [28], the authors present the first version of HADS. Contrarily to the current work, in that work, HADS tolerates only one spot VM hibernation, and, in case of migration, all tasks should restart from the beginning since no checkpointing technique is available. Furthermore, compared to the latter, the present work improves execution resilience by adopting a VM selection methodology that uses a weighted round-robin algorithm (presented in Section 4.2).

Several works from the related literature deal with the checkpointing problem of BoT applications in clouds [25, 13, 3, 29]. Yi *et al.* [31] 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 [25] 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 [29], 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 previously discussed, strategies based on predictions of spot interruptions cannot be adopted by HADS, since the new price model of Amazon does not present such a variation. Another remark is that the just described grace period checkpoint cannot be applied by HADS either, 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 [24] 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.

AutoBot [29] 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.

To the best of our knowledge, the hibernation mechanism of the spot VMs is only discussed in our previous works [27, 28] and in [10]. However, in [10], although the authors consider a scenario where hibernation-prone spot VMs can be used, they recognize that deadline constraints add complexity to the problem of resource provisioning, which should be evaluated in detail. But, no practice solution is presented or analyzed. Moreover, that works does not deal with a task scheduling problem, but a resource provisioning one.

3 System and Application Models

Let $M = M^s \cup M^o$ 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 and M^o is the set of on-demand ones. M depends on the type and market of the VMs that can be deployed during the execution and must respect the resources 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 should be composed by a maximum of five VMs $c5.xlarge$ and $c5.2xlarge$, respectively.

We also define $max_ondemand$ as the maximum number of on-demand VMs that can be allocated simultaneously. This value is defined 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. We consider two VM markets, spot and on-demand, and each $vm_j \in M$ is present in only one of them with cost c_j. In October 2017, Amazon adopted the per-second billing in Linux VMs, in which users are billing in one-second increments [6]. Similar billing models are also used by other providers. For example, in Google Compute Engine[1], when a new VM is initiated, users pay for its first minute (even if it was used by only 30 seconds) and after this minute, instances are charged in one second increments. Therefore c_j corresponds to the cost in seconds of vm_j.

We define B as the set of tasks of the BoT application. We also 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 also known and given by e_{ij}.

The user also 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 $T = \{1, \ldots, D\}$ as the set that discretizes such an execution in time intervals.

Figure 1a 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.

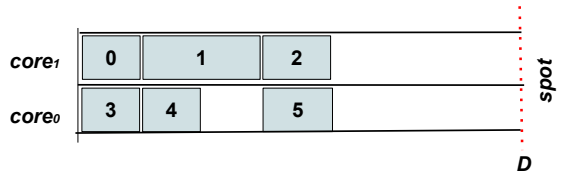
In an environment where spot VMs can hibernate and resume multiple times, let *break-point* of vm_j be the time $p \in T$ when the last hibernation of vm_j started. If vm_j resumes in time to satisfy the application deadline, as shown in Figure 1b, its tasks can go on running from the *break-point*.

All variables and parameters defined of this section are summarized in Table 2.

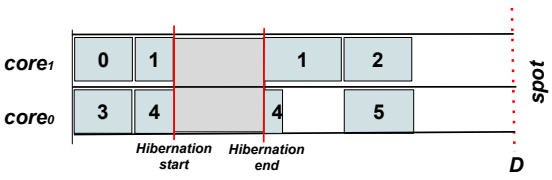
[1] https://cloud.google.com/compute/vm-instance-pricing

Table 2: Variables and parameters of the problem.

Name	Description
B	Set of tasks
$M = M^s \cup M^o$	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
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
$max_ondemand$	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



(a) Execution without hibernation



(b) Execution with hibernation

Fig. 1: Different scenarios of execution with hibernation-prone spot VMs

4 Hibernation-Aware Dynamic Scheduler

We consider BoT applications with deadline constraints that can be scheduled in both hibernation-prone spot VMs (for cost sake) and on-demand VMs. Hence, if a spot VM hibernates in time to satisfy an application deadline, the tasks of that VM can go on running from the *break-point*. However, if it is not the case, a temporal failure will possibly take place, and the application deadline will be violated. Therefore, the aim of our Hibernation-Aware Dynamic Scheduler (HADS) is to offer a dynamic scheduling solution that guarantees the execution of the tasks of applications with deadline constraints, avoiding temporal failures even in the presence of multiple hibernations with minimum monetary cost in regards to allocation prices of VMs. To this end, HADS provides a mechanisms to migrate tasks from a hibernated spot VM to other ones whenever the former does not resume early enough to ensure the deadline constraints of the application. If the existing allocated spot VMs are not enough to execute these tasks, new on-demand VMs should be deployed.

HADS is composed by two main modules: i) the **Primary Scheduling Heuristic Module**, that defines an initial task scheduling map (Section 4.2), and ii) an event-driven **Dynamic Scheduler Module** that, if necessary, migrates tasks to other VMs so that the deadline is respected (see Section 4.3). Furthermore, for reducing costs or load balancing, it may also migrates tasks from busy VMs to idle spot ones by applying a work stealing procedure (see Section 4.3.5). Finally, in order to avoid executing migrated tasks from the beginning, tasks on spot VM take checkpoints periodically (see Section 4.1). Hence, those which were running in a VM that hibernated are migrated to other VMs and start their execution from their respective last checkpoint. On the other hand, checkpointing of a task

induces overhead, increasing the task execution time, which must be considered by the Primary Scheduling Module when mapping tasks to spot VMs. For this reason, in this section, we firstly explain HADS checkpointing approach and then the two scheduling modules. All variables and parameters used by HADS' algorithms are presented in Table 3. The functions and procedures called by the algorithms are summarized in Table 4.

Table 3: Variables and parameters used on Algorithms 1 to 8

Name	Description
IR	Set of *idle* VMs
BR	Set of *busy* VMs
TR	Set of *terminated* VMs
HR	Set of *hibernated* 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
A	Set of VMs selected to execute the tasks (Used on Algorithm 2)
L	Set with n longest tasks (Used only on Algorithm 1)
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_{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)
mtt	Migration trigger time limit
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 intervals to migrate tasks to a new deployed VM

Table 4: Functions and Procedures called by Algorithms 1 to 8

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
$assign(t_i, vm_s)$	emulates the assignment of task t_i to vm_s. In this case task t_i is not actually scheduled to vm_s
$get_makespan(vm_s)$	Renders the worst case makespan as if the n longest tasks had been scheduled to the slowest VM of M
$sort_by_memory(B)$	Sorts the tasks of set B in descending order by memory size demand
$sort_by_price(A)$	Sorts the VMs of set A by price
$enough_mem(rm_i, m_j)$	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
$get_wrr_VM(M^s)$	Selects the next spot VM using the weighted round-robin heuristic
$sort_selected(K, IR, BR)$	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 *busy* 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

4.1 Checkpoint mechanism

Since in BoT application tasks execute independently to each other, an uncoordinated checkpoints approach is very suitable because, in this case, each task decides itself when to take checkpoints to save its state, not requiring any global tasks synchronization. Hence, only the last checkpoint needs to be kept for each task. We consider that, each task t_i scheduled to a spot VM, when executing, will have its checkpoint periodically saved in the Amazon Simple Storage Service (Amazon S3), a resilient storage that can be used to store any amount of data. In accordance with Amazon, S3 is designed for 99,99% availability[2].

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 in S3, 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 1. Note that, we consider that the execution time of a task t_i allocated to a vm_j increases *ovh* percent ($e_{ij} = e_{ij}(1 + ovh)$).

$$n_ckp_i = e_{ij} \times ovh/dump(t_i) \tag{1}$$

In Equation 2, we also define $intv_ckp_i$ as the time interval between two consecutive checkpoints of task t_i scheduled to spot vm_j.

$$intv_ckp_i = e_{ij}/n_ckp_i \tag{2}$$

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

4.2 Primary Scheduling Heuristic Module

In order to create an initial task scheduling map, the primary scheduler requires a parameter, denoted D_{spot}, necessary to schedule tasks on spot VM. Thus, we firstly explain how D_{spot} is estimated an then we present the Primary Scheduling Heuristic algorithm.

4.2.1 Estimation of D_{spot}

Aiming at ensuring the application deadline D no matter when hibernations take place, D_{spot} is a worst case estimated time limit which guarantees that there will always have enough spare time to migrate tasks of a hibernated spot VM to other VMs.

Algorithm 1 *compute_D_{spot}*

Input: $B, M, max_ondemand, D, \alpha$
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** $ti \in L$ **do**
5: $\quad assign(t_i, vm_s)$
6: **end for**
7: $mkp_w = get_makespan(vmf)$
8: return $max(D - (mkp_w + \alpha), 0)$

Algorithm 1 renders the estimation value of D_{spot}. The algorithm receives as input set of tasks B, set M of VMs, the maximum number of on-demand VMs that can be allocated simultaneously (*max_ondemand* parameter), the deadline D and the overhead α. The number of tasks n assigned to a VM can be estimated by dividing the number of tasks by the number of VMs (line 1). Then, the n longest tasks are included in set $L \subset B$ (line 2). Therefore, the execution of theses 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 t_i 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, selected in D_{spot} is, then, the difference between D and mkp_w plus α, the overhead to migrate the tasks (line 8). D_{spot} is illustrated in Figure 2. If D_{spot} is equal to zero, the scheduler deploys only on-demand virtual machines.

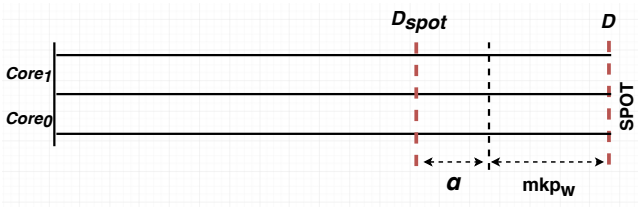


Fig. 2: D_{spot} definition

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 executed 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 com-

[2] https://aws.amazon.com/s3/faqs/

plexity $\mathcal{O}(|B|)$, 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 $|B| >> |M|$, in terms of space complexity we have $\mathcal{O}(|B|)$ as well.

4.2.2 Primary Scheduling Heuristic Algorithm

Algorithm 2 *Primary Scheduling Heuristic*

Input: B, M, M^s, M^o, D, ovh, α, and $max_ondemand$
1: $D_{spot} = compute_D_{spot}(B, M, max_ondemand, D, \alpha)$
2: $sort_by_memory(B)$ {Sort tasks by memory requirement rm_i}
3: $A \leftarrow \emptyset$ {Set of selected VMs}
4: **for all** $ti \in B$ **do**
5: {Phase 1: Try to schedule the task in 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 in 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: $M^s \leftarrow M^s \backslash \{vm_k\}$
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: $M^o \leftarrow M^o \backslash \{vm_k\}$
35: **end if**
36: **end for**
37: $create_primary_map(A)$

Algorithm 2 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 overhead α and the $max_ondemand$ parameter. The primary scheduling heuristic has three distinct phases: i) schedule of a task to an already select VM; ii) schedule of a task to a new spot VM; and iii) schedule of a task to a new on-demand VM.

Algorithm 3 *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** $start_{ij} + e_{ij} < D_k$ **and** $enough_mem(rm_i, m_j)$ **then**
5: *return* **True**
6: **else**
7: *return* **False**
8: **end if**

Initially, the algorithm computes the estimated time limit D_{spot}, used to schedule tasks in spot VMs (line 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 3), the Primary Scheduling algorithm 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 3). 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 induced by the checkpoints (line 2 of Algorithm 3). Furthermore, for spot VMs, the algorithm also computes $intv_ckp$, the interval time between two consecutive checkpoints, Equation 2 (lines 9 and 22).

Phase 1: Scheduling tasks in 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) [16]. 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 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 [9] 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 recommendations[3] that say that an application should use different types of spot VMs in order to increase the availability of spot VM instances. According to Kumar et al. [18] 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.

$$weight(vm_j) = Gflops_j/c_j, where\ vm_j \in M \quad (3)$$

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 the tasks have been scheduled, the algorithm creates the primary scheduling map (line 37) which describes the initial execution strategy that the Dynamic Scheduler Module of HADS should follow, as explained in the next section.

For each phase of Algorithm 2, we have the following time complexity. Since functions *check_schedule* (see Algorithm 3) and *compute_intv_ckp* have both time complexity $\mathcal{O}(c)$, the phase 1 of Algorithm 2 (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 2 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 $|B| >> |M|$, in terms of space complexity Algorithm 2 is $\mathcal{O}(|B|)$.

4.3 Dynamic Scheduler Module

HADS Dynamic Scheduler Module is an event-driven algorithm (Algorithm 5) 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 cost and meeting the application deadline. They may also change the state of the VM (e.g., from busy to idle, idle to terminated, etc). In order 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 4.3.1.

As previously discussed, if a hibernated spot VM does not resume or does it but too late to avoid a temporal failure, HADS Dynamic Scheduler algorithm should execute a migrate procedure. However, in this case, it is also necessary to define a Migration Trigger Time Limit ($mtt \in T$) to start executing this procedure, otherwise it will be useless to execute it. Hence, in this section, we firstly explain the concept of allocation cycles, then how mtt is computed (Section 4.3.2), and finally the Dynamic Scheduler, Migration, and Work Stealing algorithms.

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 3 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 a cost of α. 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, contrarily to task 4 that does not take any checkpoint, task 1 will start executing in the new VM from its last checkpoint. Note that, in HADS checkpoints are parallel. Therefore, tasks running in the same VM can record checkpoints at the same time.

4.3.1 VM states and Allocation Cycle (AC) concept

Let BR, IR, HR, and TR of M be the set 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 is 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 ($vm_j \in TR$).

In order 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

[3] https://aws.amazon.com/pt/ec2/spot/instance-advisor/

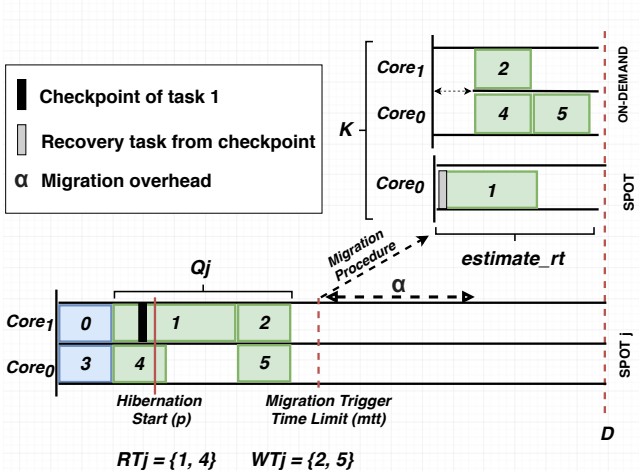


Fig. 3: Execution where v_j hibernates at p, does not resume, and its tasks are migrated at mtt

by second (see Section 3), 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 Scheduler 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 4 shows an example in which the execution of scheduled tasks on 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 terminates at the end of AC_2.

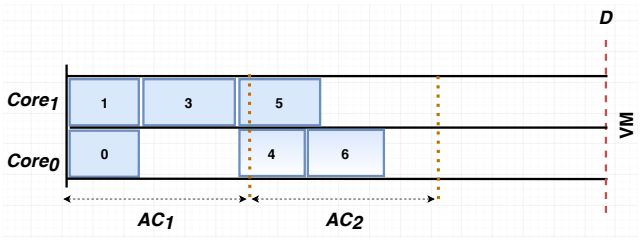


Fig. 4: A scheduling with two logical Allocation Cycles (AC)

4.3.2 Migration Trigger Time Limit (mtt)

Let $K \subset M$ be the set of VMs that will receive the migrated tasks. For instance, in Figure 3, 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 on 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 4.3. 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 4 expresses the migration trigger 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.

$$mtt = D - (estimate_rt(Q_j, IR, BR, M^O, D) + \alpha) \quad (4)$$

4.3.3 Dynamic Scheduler Module Algorithm

Algorithm 4 *estimate_rt*

```
Input: Q_j IR, BR, M^o and D
 1: K ← ∅
 2: for t_i ∈ Q_j do
 3:    for vm_k ∈ sort_selected(K, IR, BR) do
 4:       if check_migration(t_i, vm_k, D) then
 5:          assign(t_i, vm_k)
 6:          K ← K ∪ {vm_k}
 7:          break {Next task}
 8:       end if
 9:    end for
10:    if task was not assign then
11:       for vm_k ∈ M^o do
12:          if check_migration(t_i, vm_k, D) then
13:             assign(t_i, vm_k)
14:             K ← K ∪ {vm_k}
15:             break {Next task}
16:          end if
17:       end for
18:    end if
19: end for
20: return get_makespan(K)
```

Algorithm 5 *Event Handler*

```
Input: event, vm_j, Q_j, IR, BR, TR, HR, M^o, α, and D
 1: switch (event)
 2:  case Q_j = ∅:
 3:    BR ← BR\{vm_j}
 4:    IR ← IR ∪ {vm_j}
 5:    work-stealing_procedure(vm_j, BR) {/*Algorithm 8*/}
 6:  case vm_j ∈ IR and AC_cur_j ended :
 7:    IR ← IR\{vm_j}
 8:    TR ← TR ∪ {vm_j}
 9:  case vm_j hibernates:
10:    if vm_j ∈ BR then
11:       mtt ← D − (estimate_rt(Q_j, IR, BR, M^o, D) + α)
12:       BR ← BR\{vm_j}
13:    else
14:       IR ← IR\{vm_j}
15:    end if
16:    HR ← HR ∪ {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 ← HR\{vm_j}
21:    if resumes before mtt then
22:       BR ← BR ∪ {vm_j}
23:    else
24:       work-stealing_procedure(BR, IR, vm_j, D)
          {/*Algorithm 8*/}
25:    end if
26: end switch
```

The possible events with which it deals and the respective actions are described in the sequence.

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 4.3.5.

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 trigger time limit mtt (Equation 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 4.3.2. If the vm_j was *idle*, the algorithm just changes its state to *hibernated* (lines 12 and 16).

Migration trigger time limit reached (line 17)

Whenever vm_j is *busy* and the migration trigger 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 4.3.4.

Spot vm_j resumes (line 19)

The spot vm_j may resume before the migration trigger 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 trigger 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 4.3.5.

Figure 5 shows the state diagram of spot vm_j and the transition between them according to the presented events.

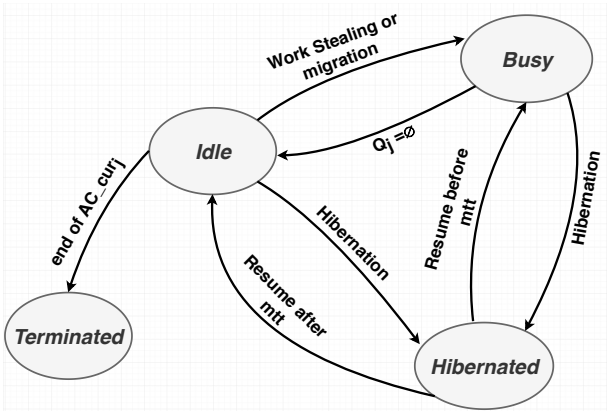


Fig. 5: Spot vm_j state diagram

4.3.4 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 in 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 $t_i \in RT_l$, 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.

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: $IR \leftarrow IR \backslash \{vm_j\}$
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: $M^o \leftarrow M^o \backslash \{vm_j\}$
32: $BR \leftarrow BR \cup \{vm_j\}$
33: *break* {Migrate next task}
34: **end if**
35: **end for**
36: **end if**
37: **end for**

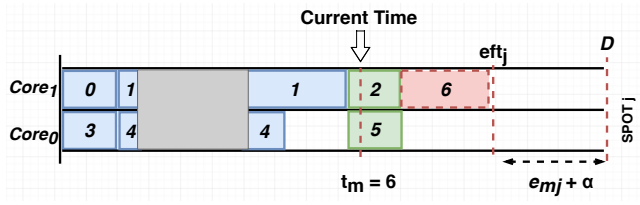
Algorithm 7 *check_migration*

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**

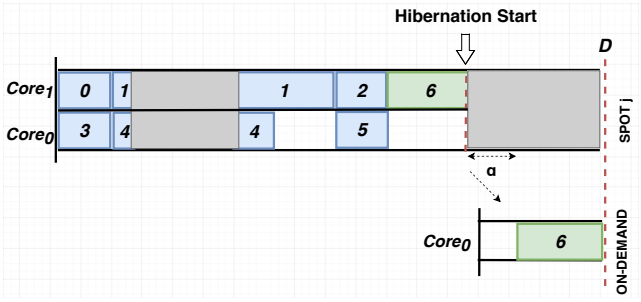
In order to migrate a task t_i to a spot vm_j, Algorithm 6 calls the procedure *check_migration* of Algorithm 7. Task t_i can be migrated to vm_j provided that the latter, after scheduling t_i, has enough spare time for executing a migration procedure of its longest scheduled task, since vm_j itself 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 6a shows a scenario where condition (1) is satisfied. Note that task 6 is the longest one.

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 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 guarantees that all tasks will have enough time to migrate and executed, respecting the application's deadline D, no matter when hibernation events take place. Figure 6b 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.



(a) Evaluating the migration of task 6 to a spot VM



(b) Migration of task 6 to a new on-demand VM

Fig. 6: Example of task migration in a scenario with successive hibernations

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 (lines 3 to 12) 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 (lines 13 to 23) 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 (line 26).

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:

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

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 $\mathcal{O}(|Q_l|)$.

4.3.5 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.

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 7 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.

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: $IR \leftarrow IR \backslash \{vm_k\}$
14: **end if**

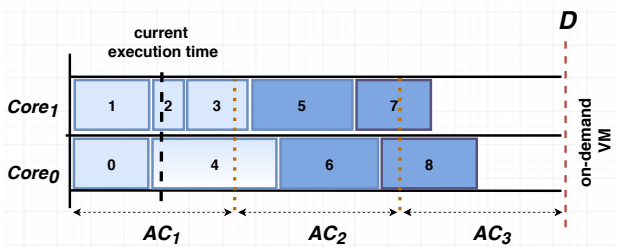


Fig. 7: Example of tasks in an on-demand VM that can be stolen by the work-stealing procedure

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 $\mathcal{O}(|WT_j| + |Q_k|)$.

5 Evaluation Results

This section presents performance results of experiments conducted on Amazon EC2 virtual machines with two BoT applications whose tasks are scheduled by HADS.

We should point out that all evaluation results shown in tables and figures in this section concern the average of three executions.

5.1 BoT applications

In our experiments, we use the two following BoT applications with deadline constraints:

Synthetic: In this case, the jobs are composed by tasks generated with the application template proposed by Alves et al. [2] 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 2.81 MB 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 [5]. By executing ED, we created the job ED200 which is composed of 200 tasks running the largest problem size (class B).

Table 5 shows the characteristics of the four BoT jobs, including their respective number of tasks, memory footprint, and runtime. For all jobs, the execution deadline is 35 minutes ($D = 2100s$).

Table 5: 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

5.2 Experimental Environment

All results presented in this work were obtained from real executions using VMs from 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 [4]. Table 6 shows the computational characteristics of VMs that we used in our experiments as well as its respective prices in on-demand and spot markets in December 2019.

Table 6: VMs attributes

Type	#VCPUs	Memory	Gflops	On-demand price per hour	Spot price 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$

5.2.1 Hibernation Emulation Module (HEM)

Cloud users have no control on spot VMs hibernations since it is the cloud provider that decides when to hibernate and resume a given spot VM according to resource demands variation. Thus, in order to evaluate different patterns of spot VMs hibernation and resuming, we have developed the Hibernation Emulation Module (HEM) that emulates the cloud hibernation feature, using Poisson distribution [1] to model both the hibernation and resuming times 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, HEM emulates events for groups of VMs of identical types. In other words, when a HEM event happens for a VM, it has an impact not only on this VM but also on all VMs of that type. HEM uses 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[5], 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

[4] https://aws.amazon.com/ec2/spot/instance-advisor/
[5] https://www.criu.org/

5.2.2 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 specified by Amazon EC2 [6], which means that up to five VMs of each type in each market can be allocated. The checkpoint overhead ovh, was set to 10%. As said in section 5.1, $D = 2100s$ (35 minutes). Table 7 summarize the used setup parameters and the set of VMs.

Table 7: Experimental Setup Parameters (time in seconds)

AC	α	M^o	M^s	D
900	180	5-c3.large 5-c4.large 5-c4.xlarge 5-c4.xlarge	5-c3.large 5-c4.large 5-c4.xlarge 5-c4.xlarge	2100

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 10, discussed later. Table 8 presents seven different scenarios by varying k_h and k_r.

Figure 8 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,

[6] https://docs.aws.amazon.com/AWSEC2/UserGuide/ec2-resource-limits.html

Table 8: Different execution scenarios generated by varying parameters λ_h and λ_r

ID	*hibernating*	*resuming*	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

the expect number of hibernation events is $k_h = 2$ while in scenario sc_5, it is $k_h = 3$.

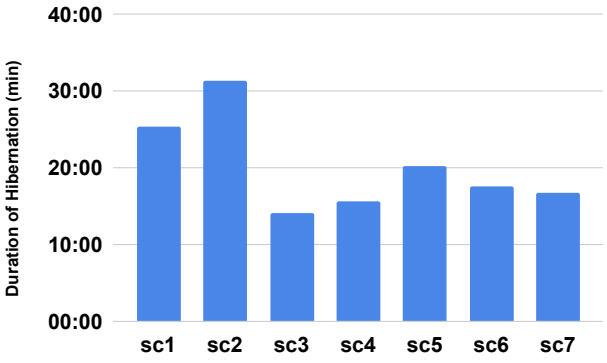


Fig. 8: Average duration of hibernation in different scenarios

5.3 BoT Jobs Evaluation

In this section, for evaluation comparison sake, we firstly define two baseline cases for the execution of the four jobs. Then, we evaluate and discuss the results of experiments on EC2 conducted with the four jobs and the seven generated scenarios.

5.3.1 Baseline job execution cases

We have considered two baseline cases: i) *spot VMs without hibernation*, which is the case where the initial scheduling defined by the Algorithm 2 is followed without the need of migration; and ii) *on-demand only*, which uses the same scheduling, but only with on-demand VMs.

Table 9 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 6). Note also that, in Table 9 the jobs makespan are below the 35 minutes of the jobs deadline. This occurs because Algorithm 2 respects the D_{spot} limit presented in Section 4.2.1.

Table 9: 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%

5.3.2 Monetary cost evaluation of the jobs execution

Table 10 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 2 in respecting applications deadline.

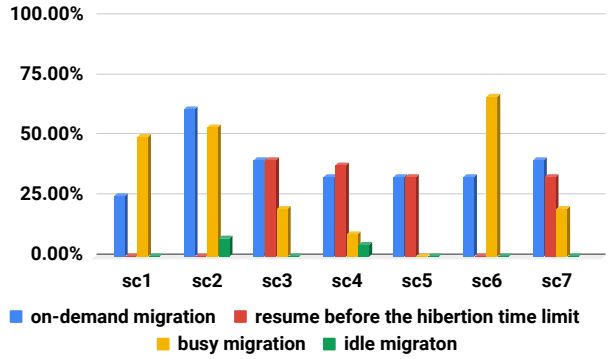
Table 10: Execution of HADS in scenarios sc_1 to sc_7. The table shows the probabilistic mass function of the hibernation (λ_h) and the resume events (λ_r) for each scenario, the average number of hibernations, the number of used on-demand VMs, the average makespan, and the average monetary cost

	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%**

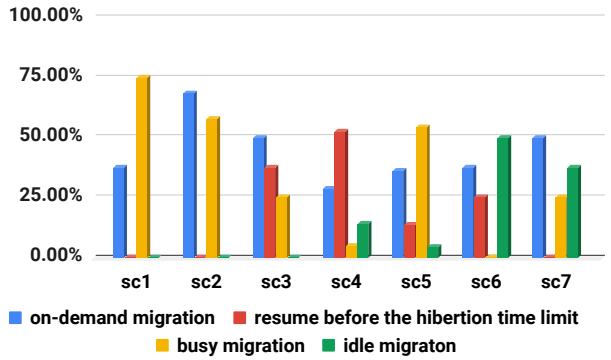
Figures 9 and 10 show the percentage of time that a procedure is executed in regards to the total number of hibernations occurred along with the job execution. For instance, if 4 hibernations occurred and in 2 of them the on-demand migration procedure took place, there will be a bar with 50%. In the 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 10 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 9 and 10 show that in sc_2 the migration to on-demand VMs occurs in more than 50% of the hibernation cases. The impact of the

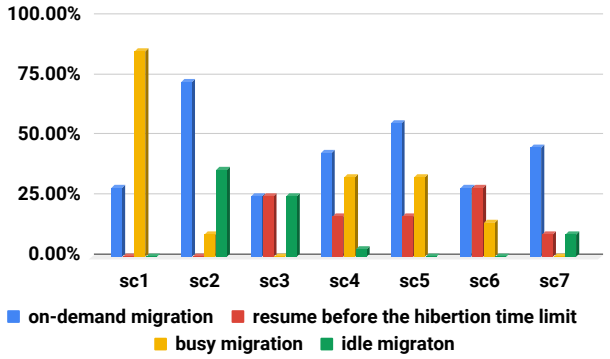


(a) Job J60

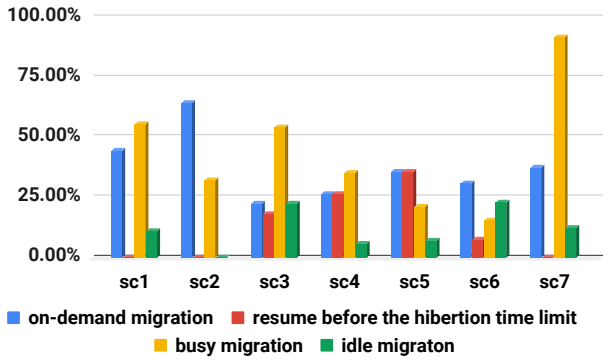


(b) Job J80

Fig. 9: Percentage distribution of the procedures used by HADS during the execution of jobs J60 and J80



(a) Job J100



(b) Job ED200

Fig. 10: Percentage distribution of the procedures used by HADS during the execution of jobs J100 and ED200

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 9 and 10, 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 rate ($k_r = 5$). Figures 9 and 10 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%.

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 9 and 10 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.

5.3.3 Impact of hibernation and resuming

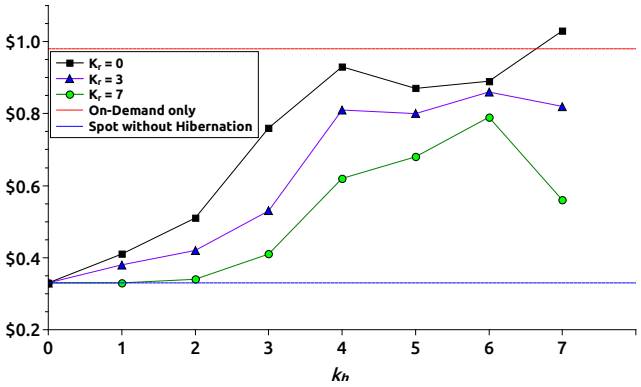


Fig. 11: Impact of variation of k_h in the execution costs of job ED200

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 11, we could say that in scenarios where spots hibernate, monetary costs variation has a 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 event increases becoming, therefore, cheaper than the former.

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 a task, requiring, therefore, on-demand VMs to meet application deadline. On the other hand, with 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.

5.3.4 Work-stealing occurrence

Considering the execution of job ED200, this section studies the behavior of the work-stealing procedure proposed by HADS. Figure 12 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 12 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 a behavior is expected since the work-stealing procedure only steals tasks which are waiting to be executed and which would start executing in the next VM's AC (see Section 4.3.5). In addition, 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 higher work-stealing successful rate in regards to the other scenarios.

Figure 13 illustrates the variation in the number of scheduled tasks (left axis) of two 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 13b) and 6 tasks (Figure 13a) were

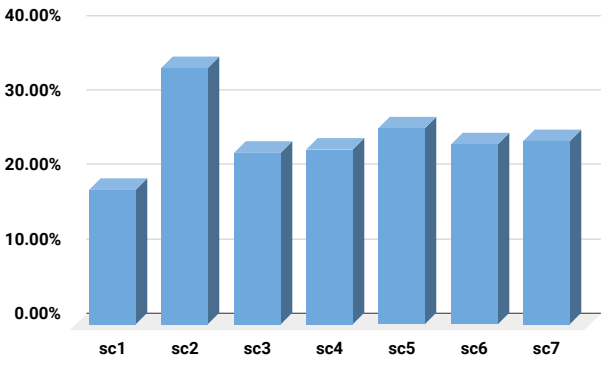
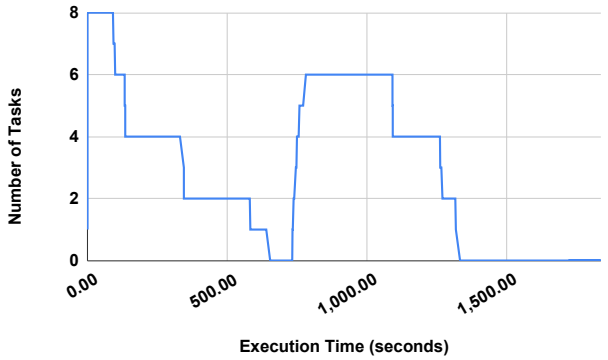
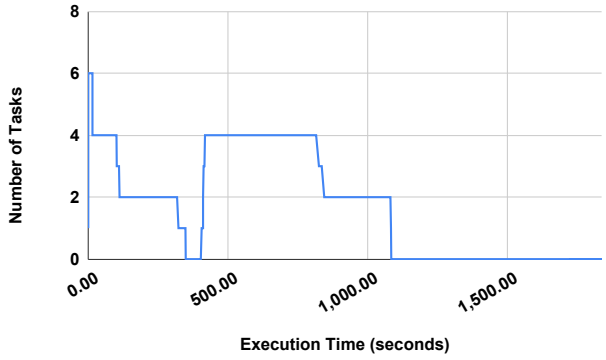


Fig. 12: Work-stealing distribution of Job ED200



(a) c4.large



(b) c3.large

Fig. 13: Tasks progress of two spot VMs during the execution of Job ED200 in scenario sc_2

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 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 Figure 13 was also observed for the other VMs used to execute job ED200. In general, if the tasks of the initial list was completed during the first VM's AC, the work-stealing was performed. Such a behavior confirms that the size of the AC, as well as the initial distribution of tasks, has an impact in work-stealing successful rate.

5.3.5 Impact of checkpointing

In order to study the impact of checkpointing in the execution of 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 14, 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 regards to 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.

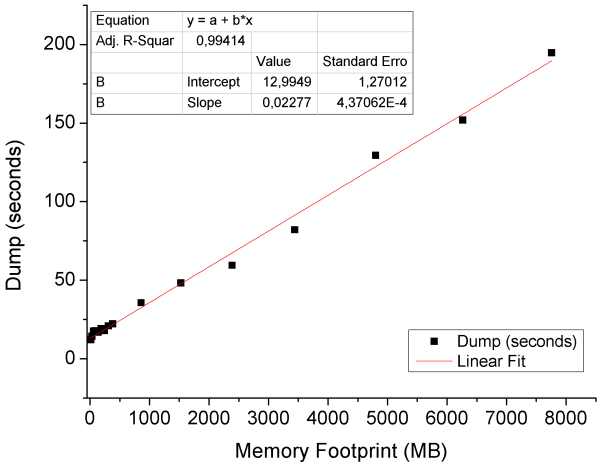


Fig. 14: Dump time variation

Considering the execution of the ED200 job in each of the seven scenarios desribed in Table 8, Table 11 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 11: 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

6 Concluding Remarks and Future Work

This paper has proposed the Hibernation-Aware Dynamic Scheduler (HADS), a dynamic scheduler for bag-of-task applications with deadline constraints that uses both hibernation-prone spot VMs (for cost sake) and on-demand VMs. Our scheduling aims at minimizing the monetary costs of bag-of-tasks, respecting the application's deadline and avoiding temporal failures. The proposed strategy was evaluated using the VMs of AWS EC2 with real executions of synthetic applications and a real embarrassingly distributed application from NAS benchmark, considering seven emulated scenarios with different spot VM hibernation and resuming rates. Our results confirm the effectiveness of our scheduling in terms of monetary costs and that it avoids temporal failures even in the presence of multiple hibernations. Furthermore, they show that the resuming rate of hibernated spot VMs has an impact in monetary costs.

In the near future, we aim at conducting new experiments with different allocation cycles values as well as the size and number of tasks, since our results show that they have an impact in performance, specially in the work-stealing procedure whose effectiveness might be improved (see Section 5.3.4) .

As another research direction, we intend to adapt the HADS by considering that there is no previous knowledge about the characteristics of the applications such as tasks execution time or memory requirements. We would also like to extend HADS solution to other classes of deadline constraint applications, i.e., non BoT applications.

Acknowledgment

This research was supported by FAPERJ (process number E-26 200.752/2019 242697) and *Programa Institucional de Internacionalização* (PrInt) from CAPES as part of the project REMATCH (process number 88887. 310261/2018-00).

References

1. Ahrens, J.H., Dieter, U.: Computer methods for sampling from gamma, beta, poisson and bionomial distributions. Computing **12**(3), 223–246 (1974)
2. Alves, M.M., de Assumpção Drummond, L.M.: A multivariate and quantitative model for predicting cross-application interference in virtual environments. Journal of Systems and Software **128**, 150 – 163 (2017). DOI https://doi.org/10.1016/j.jss.2017.04.001
3. 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, pp. 1–8 (2013)
4. AWS, A.: 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 15 December 2019
5. 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)
6. 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 15 December 2019
7. Chakravarthi, K.K., Shyamala, L., Vaidehi, V.: Budget aware scheduling algorithm for workflow applications in iaas clouds. Cluster Computing pp. 1–15 (2020)
8. Chhabra, A., Singh, G., Kahlon, K.S.: Multi-criteria hpc task scheduling on iaas cloud infrastructures using meta-heuristics. Cluster Computing pp. 1–34 (2020)
9. Dongarra, J.J., Luszczek, P., Petitet, A.: The linpack benchmark: past, present and future. Concurrency and Computation: practice and experience **15**(9), 803–820 (2003)
10. Fabra, J., Ezpeleta, J., Álvarez, P.: Reducing the price of resource provisioning using ec2 spot instances with prediction models. Future Generation Computer Systems **96**, 348–367 (2019)
11. 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, pp. 423–428. IEEE (2012)

12. 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), e3770 (2020)
13. 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, pp. 455–462 (2010)
14. 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), 1682–1699 (2013)
15. 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 pp. 1–11 (2019)
16. 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), 1265–1279 (1991)
17. 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**, 1–21 (2017)
18. 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), 809–856 (2018)
19. 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, pp. 657–662 (2013)
20. 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), 513–520 (2019)
21. 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., pp. 177–187 (2014)
22. Oprescu, A.M., Kielmann, T.: Bag-of-tasks scheduling under budget constraints. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, pp. 351–359. IEEE (2010)
23. 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 15 December 2019
24. 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, pp. 16:1–16:15 (2015)
25. 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, pp. 329–341 (2015)
26. Tang, X., Liao, X., Zheng, J., Yang, X.: Energy efficient job scheduling with workload prediction on cloud data center. Cluster Computing **21**(3), 1581–1593 (2018)
27. Teylo, L., Arantes, L., Sens, P., d. A. Drummond, L.M.: A bag-of-tasks scheduler tolerant to temporal failures in clouds. In: 2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 144–151 (2019)
28. Teylo, L., Arantes, L., Sens, P., de A Drummond, L.M.: A hibernation aware dynamic scheduler for cloud environments. In: Proceedings of the 48th International Conference on Parallel Processing: Workshops, p. 24. ACM (2019)
29. 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), 1512–1527 (2019)
30. 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, pp. 337–344 (2014)
31. Yi, S., Andrzejak, A., Kondo, D.: Monetary cost-aware checkpointing and migration on amazon cloud spot instances. IEEE Transactions on Services Computing **5**(4), 512–524 (2011)