# Scheduling Bag-of-Tasks in Clouds using Spot and Burstable Virtual Machines

Luan Teylo, Luciana Arantes, Pierre Sens, Lucia Maria de A. Drummond

# Scheduling Bag-of-Tasks in Clouds using Spot and Burstable Virtual Machines

Luan Teylo, Luciana Arantes, Pierre Sens and Lúcia Maria de A. Drummond

**Abstract**—Cloud providers offer several types of Virtual Machines (VMs) in diverse markets, with different guarantees in terms of availability and reliability. Among them, the most popular market models are the on-demand and the spot. On-demand VMs are allocated for a fixed cost per time, and their availability is ensured during the whole execution. On the other hand, in the spot market, VMs are offered with a huge discount, but their availability fluctuates according to cloud's current demand that can terminate or hibernate a spot VM at any time. Furthermore, to cope with workload variations, cloud providers have also introduced the concept of burstable VMs, which can burst up their CPU performance during a limited period of time. In this work, we present the Burst Hibernation-Aware Dynamic Scheduler (Burst-HADS), a framework that executes Bag-of-Tasks applications with deadline constraints by exploiting both spot and on-demand burstable VMs, aiming at minimizing both the monetary cost and the execution time. Performance results on Amazon EC2 show that Burst-HADS reduces the monetary cost and meets the application deadline even in spot hibernation scenarios, when compared to other approaches from the related literature which uses only spot and non-burstable on-demand instances.

**Index Terms**—Cloud Computing, BoT Scheduling, Burstable VMs, Spot VM Hibernation

## 1 Introduction

In the past few years, cloud computing has emerged as an attractive solution to execute different applications. It brings several advantages compared with dedicated infrastructure, as, for example, a significant reduction in operational costs. However, in cloud environments, besides the usual goal of minimizing the application's execution time, it is also essential to reduce the monetary cost because even though in the cloud, computational resources are virtually infinite, the user's budget is not.

Cloud platforms enable users to dynamically acquire computational resources wrapped as Virtual Machines (VMs), that can be selected by the users according to their application requirements (CPU, memory, I/O, etc.) in a pay-as-you-use price model. Furthermore, cloud providers offer VMs in different contract models, with different guarantees in terms of availability and volatility. For instance, in Amazon EC2, there are three main contract models (also called markets): i) *reserved* market, where the user pays an upfront price, guaranteeing long-term availability; ii) *on-demand* market which is allocated for specific periods of time, and incurs a fixed cost per unit time of use, ensuring the availability of the instance during this period; iii) *spot* market in which unused resources are available up to 90% discount when compared to the on-demand model, but such resources can be requested at any time.

In the three markets, there exist a wide range of VM types that suit different user requirements. According to Amazon Web Service (AWS), "Instance types comprise varying combinations of virtual CPUs (vCPUs), memory, storage, and networking capacity and give you the flexibility to choose the appropriate mix of resources for your applications. Each instance type includes one or more instance sizes, allowing you to scale your resources to the requirements of your target workload"[1]. Instances types are grouped into families based on their respective use cases. For example, the compute-optimized instances (C3, C4, and C5) are ideal for compute-bound applications that require high-performance processors.

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

Besides the markets, some leading cloud providers (e.g. Microsoft Azure, Amazon EC2), introduced in the last years the concept of a burstable VM that can sprint its performance during a limited period of time to cope with sudden workload variations. By operating on a CPU credit

- *Luan Teylo and Lúcia Maria de A. Drummond are with the Institute of Computing, Federal Fluminense University, Brazil.*
- *Luciana Arantes and Pierre Sens are with Sorbonne Université, CNRS, Inria, France.*

1. https://aws.amazon.com/ec2/instance-types

regime that controls the processing power offered to users, burstable VM instances can use 100% of the VM's processing power (burst mode) or only a fraction of that processing power (baseline mode), depending on credits. They can then accumulate CPU credits per hour, whose amount also depends on the instance type. If a burstable instance uses fewer CPU resources than required for baseline performance (for example, when it is idle), the unspent CPU credits are accrued in the CPU credit balance of the instance. If the latter needs to burst above the baseline performance level, it spends the accrued credits. The higher the number of credits that a burstable instance has accumulated, the longer it can burst beyond its baseline when higher performance is required. Burstable instances have two main advantages: i) they are offered with an up to 20% discount compared to non-burstable on-demand instances with equivalent computational resources, and ii) contrarily to spot VMs, they are not prone to revocation neither hibernation. On the other hand, to obtain monetary advantages of burstable instances, the user has to control their respective CPU credit usage by monitoring their baseline performance and defining bursting periods. In Amazon EC2, burstable VMs are of type T3, T3a, or T2 [1].

In the current work, we consider Bag-of-Tasks (BoT) applications, which are commonly used in solving various science and engineering problems, such as parameter sweep, chromosome mapping, Monte Carlo simulation and computer imaging applications [2]. As the interconnection network is not the bottleneck for the good performance of this type of applications, it is possible to speed up them by distributing their tasks on a large number of VMs in the cloud. In this context, an important issue is the definition of an efficient initial scheduling for the tasks which minimizes both the execution time and the financial cost, typically conflicting objectives. To this end, we propose to mathematically formulate it as a multi-objective integer programming problem and/or solve it by applying metaheuristics efficiently. Furthermore, for many of such applications, cost optimization algorithms should respect a given deadline, otherwise, a temporal failure takes place. Considering that the application executes on spot VMs that can be revoked/hibernated, we should provide a second scheduler, denoted dynamic scheduler, which would be responsible for provisioning new VMs and migrating tasks between different VMs, if necessary.

Therefore, this article presents a framework for scheduling Bag-of-Tasks (BoT) applications with deadline constraints, exploring hibernate-prone spot instances and on-demand burstable instances aiming at minimizing both the monetary cost and the execution time of applications. The proposed framework, denoted Burst Hibernation-Aware Dynamic Scheduler (Burst-HADS), is an event-driven scheduling framework built in a modular way with two main scheduling modules: i) the Primary Scheduling Module that defines an initial scheduling map of tasks to VMs, and ii) a Dynamic Scheduling Module responsible for task migration in case of hibernation.

In a previous work [3], we have explored the use of hibernation-prone spot instances to minimize the monetary cost of BoT applications execution, respecting their deadline constraints. To meet such a deadline, even in the presence of multiple hibernations, new on-demand VMs, not allocated in the initial mapping, could be dynamically launched. In this case, tasks of the hibernated spot instances and those not executed yet are migrated to on-demand instances. Although the strategy can significantly reduce the monetary costs, always respecting the application deadline, if VM spots hibernate, the application's total execution time may considerably increase when compared to a solution that uses only on-demand instances. Hence to tackle such a problem, in this article, we investigate how burstable instances can be used to reduce the impact on the execution time of the application and the corresponding monetary costs. We also propose a formulation of the task scheduling problem as a multi-objective integer programming problem (execution time and financial cost objectives), and a heuristic based on the Iterated Local Search (ILS) [4] method for generating the initial scheduling map.

Applying different scenarios, we have studied and evaluated our proposed framework. The results show that the burstable instance based approach can optimize the monetary cost without degrading the application execution time, even in the presence of several hibernations. The evaluation of Burst-HADS targeted different issues. Firstly, we focused on the new proposed Iterated Local Search (ILS) based primary scheduler. The ILS approach was compared to optimal solutions given by the exact model of the problem and also with three widely used scheduling heuristics: MinMin, MaxMin and Greedy [5]. Next, we conducted a series of experiments with a real scenario using the Amazon EC2 cloud [6], where Burst-HADS was compared to three other approaches: (i) a baseline case where only on-demand VMs are used, (ii) HADS, proposed in our previous work [3], and (iii) an approach based on AutoBoT [7], a BoT scheduler that also exploits spot VMs to minimize the monetary cost while meeting the deadline defined by the user. To the best of our knowledge, AutoBoT is the closest to our work in the literature. However, since it considers the old EC2 price model, which was based on price history and spot market bid, and its original code is not available, we have implemented a version of AutoBot, denoted AutoBot-like, adapting some procedures of it to the current EC2 price model, defined by the supply and demand of spare resources. Performance results with experiments using both real and synthetic BoT applications in different scenarios subject to several spot VM hibernations and resuming events confirm the effectiveness of Burst-HADS in terms of monetary costs and execution times when compared to the other approaches.

The remainder of this paper is organized as follows. Section 2 discusses some related work, including the description of AutoBot and HADS, used in our analysis. Section 3 presents the mathematical formulation and the primary and dynamic modules of the Burst-HADS framework. The evaluation results of both the ILS based primary scheduling module and the dynamic module are presented in Section 4. Finally, Section 5 concludes the paper and introduces some future directions.

## 2 Related Work

Since AWS introduced the concept of burstable VM, many works exploring its features have been proposed. Many of

them focus on evaluating the performance gain obtained by using burstable VMs. In [8], Leitner and Scheuner presented a first empirical and analytical study about the second generation of AWS burstable instances (T2 family). They considered the T2.micro, T2.small, and T2.medium instances. Their work aimed at answering if, in terms of monetary cost and performance, these instance types are more effective than other ones. The results show that compared to general-purpose and computed-optimized instances (2015 generation), the evaluated T2 instances provide a higher CPU performance-cost ratio as long as the average utilization of instances is below 40%. To figure out the CPU usage limits on-the-fly, considering the dynamic variation of workloads, Ali *et al.* proposed in [9] an autonomic framework that combines light-weight profiling and an analytical model. The objective was to maximize the amount of work done using the burstable capacity of T2 VM instances. The authors state that the framework extends the CPU credits depletion period. Similarly to Leitner and Scheuner's work [8], their results also confirmed the benefits of active CPU usage control when burstable instances are exploited.

Jiang *et al.* [10] analytically modeled the performance of burstable VMs, considering their configurations, such as CPU, memory, and CPU credits parameters. They also showed that providers could maximize their total revenue by finding the optimal prices for burstable instances. Although their work, contrarily to ours, does not focus on application performance, its contribution is interesting since it states that providers can offer burstable instances for low prices without losing revenue while meeting QoS parameters.

Other works take advantage of burstable instance features in scheduling and scaling problems. In [11], for example, Baarzi *et al.* proposed an autoscale tool denoted BurScale, which uses burstable instances, together with on-demand ones, in order to handle transient queuing which arises due to traffic variation. They also presented how burstable instances can mask VM startup/warmup costs when autoscaling, which handles flash crowds, takes place. Using two distinct workloads, a stateless web server cluster and a stateful Memcached caching cluster, the authors showed that a careful combination of burstable and regular instances ensures similar performance for applications as traditional autoscaling systems while reducing up to 50% of the monetary cost. In [12], Wang *et al.* combined on-demand, spot, and burstable instances proposing an in-memory distributed storage approach. Burstable instances were used as a backup to overcome performance degradation resulting from spot instance revocations. According to the authors, those instances' burst capacity makes them ideal candidates for such a backup. Performance results show that the backup using burstable instances presents latency 25% lower than the latency of backup based on regular instances, inducing, therefore, a significant monetary cost saving.

Spot and on-demand instances have received a lot of attention in applications scheduling [7], [13], [14], [15], [16], [17]. Most of those works cope with termination/revocation of spot VMs and do not consider the hibernation feature. Moreover, the majority of them exploit the historical of spot VM price variation to predict spot VMs' revocations. However, since EC2 adopted a new price model (announced in December 2017), prices of VMs in the spot market are quite stable and defined exclusively by the supply and demand of spare capacity and no more by bid prices[2]. Therefore, it is no longer possible to predict the revocation/termination of spot VMs based only on the history of price variations as those works do.

To the best of our knowledge, the hibernation mechanism of the spot VMs is only discussed in our previous works [3], [16] and in Fabra *et al.* [18]. We proposed, in [16], a static heuristic that creates predefined backup maps before the execution of the job tasks themselves. It was the first attempt to cope with spot VMs hibernation, and results from simulations showed that the hibernation problem is better handled with a dynamic approach. In [3], we present a dynamic scheduler, denoted HADS that uses both spot and regular (non-burstable) on-demand VMs to execute BoT applications. It aims at minimizing the execution's monetary cost respecting application deadline. In [18], the authors consider a scenario where hibernation-prone spot VMs can be used and then they show that deadline constraints add complexity to the problem of resource provisioning. However, no performance experiment results are presented neither discussed. Moreover, the work does not focus on the task scheduling problem, but on resource provisioning.

Recently, in [7], Varshney and Simmhan proposed the AutoBoT Scheduler. According to the authors, AutoBoT uses several heuristics to choose a set of spot and on-demand VMs to execute BoT applications subject to a deadline defined by the user. The objective is to reduce the monetary cost of the execution, and the scheduler combines checkpoints and migration procedures to meet the deadline, even with spot revocations. Like Burst-HADS, AutoBoT also has two execution phases: i) the initial mapping, where an initial execution map is built, and ii) the execution phase, where the tasks are executed. Similarly to the Dynamic Scheduling of Burst-HADS, the execution phase of AutoBoT also reacts to the spot revocations by changing the initial execution map, adding on-demand VMs, and migrating the affected tasks. However, there are some crucial differences between AutoBoT and Burst-HADS. Firstly, although it is a recent work, AutoBoT takes some scheduling decisions based on the old spot price model. Moreover, differently from Burst-HADS, AutoBoT does not explore the hibernation feature of spot VMs neither the burst capacity of burstable VMs to reduce the monetary cost and minimize the execution time. Another important point to highlight is that AutoBoT was not evaluated in a real cloud. Instead, the authors presented a comprehensive study in a simulated environment.

## 3 Scheduling with Hibernation-prone Spot and Burstable On-demand Instances

The Burst Hibernation-Aware Dynamic Scheduler (Burst-HADS) has two main scheduling modules: i) the Primary Scheduling Module, that defines an initial scheduling map (Section 3.2); and ii) a Dynamic Scheduling Module responsible for migrating tasks or executing task rollback recovering in idle VMs (Section 3.3). In this section, we present

2. https://aws.amazon.com/pt/blogs/compute/new-amazon-ec2-spot-pricing/

the algorithms and procedures proposed for each one of these modules. In Section 3.1, we formulate the primary scheduling map as a multi-objective integer programming problem.

### 3.1 Mathematical Formulation

The problem of scheduling tasks in distributed computing resources is an NP-complete one [19], even in simple scenarios. Furthermore, some features of clouds render it more difficult. We thus model the primary task scheduling problem as a multi-objective integer programming problem whose objectives are to minimize both the monetary cost and the total execution time of the application. In other words, in our case, it is defined as the problem of creating an initial scheduling strategy, respecting the limit of available virtual cores (vCPUs) and memory capacity of the used VMs while minimizing the makespan and the monetary costs of the execution. Then, a number of burstable VMs, defined in function of the number of selected spot VMs, are included as an additional resource at the last step of the Primary Scheduling Module (see Section 3.2). Therefore, in the mathematical model we consider only spot VMs.

Let $M^s$ be the set of spot VMs that can be used to execute the BoT application and let $D$ be the application deadline, defined by the user. We then consider a set $T = \{1, \ldots, D\}$ of discrete valid time periods. Each $vm_j \in M^s$ has a memory capacity of $m_j$ gigabytes and $nvc_j$ vCPUs. When a new $vm_j$ is launched, the user is charged $c_j$ for each period of time. When the VM terminates or hibernates, the user's charge for this VM immediately stops. Let $B$ be the set of tasks of a BoT application. We assume that each task $t_i \in B$ requires a known amount of memory $rm_i$, and it is executed in only one vCPU of $vm_j$. A multi-core $vm_j$ ($nvc_j > 1$) can execute more than one task simultaneously (one task per core) only when there is enough main memory to allocate them in the VM. We also consider that the execution time $e_{ij}$ of each task $t_i$ in any $vm_j \in M^s$ is known beforehand.

Aiming at ensuring the application deadline $D$ no matter if and when spot VM hibernations happen, in our previous work [3], we define $D_{spot}$ as the worst-case estimated makespan, which guarantees that there will always have enough spare time to migrate tasks of any hibernated spot VM to other VMs and to execute them before the deadline $D$. $D_{spot}$ is computed by considering the deadline $D$ and the longest tasks that might need to be migrated and executed to/in the slowest VMs. Let the binary variable $X_{ij}^v$ indicate whether a *task* $t_i \in B$ allocated to a $vm_j \in M^s$ will start executing ($X_{ij}^v = 1$), or not ($X_{ij}^v = 0$), at time period $v \in T$. Let also $Z_j$ and $ZT$ be continuous variables which respectively keep the last period of execution of a $vm_j \in M^s$ and the total execution time of the application (makespan).

All variables and parameters defined in this section are summarized in Table 1.

TABLE 1: Notation and Variables used in the Mathematical Formulation.

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

The proposed objective function (Equation 1) is a weighted function that minimizes the monetary cost and the makespan, where $\alpha$ is the weight given by the user for the objectives.

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

Note that, in this case, both the monetary cost and the makespan have to be first normalized. The normalization procedure updates the target values to share the same minimum and maximum values, 0 and 1, respectively. Thus, the solution's total monetary cost was divided by the product of the monetary cost of hiring the most expensive spot $vm_j \in M^s$, during $D_{spot}$ periods, times the maximum number of VMs that can be deployed. Similarly, the makespan is divided by $D_{spot}$. When the heuristic finds a solution with makespan longer than the $D_{spot}$ limit, it does not accomplish the normalization, and that solution is discarded by ILS. The objective function is subject to the following constraints.

Constraint 2 guarantees that every task $t_i \in B$ must be executed, starting at a time $v \in T$ in a $vm_j \in M^s$. Constraint 3 ensures that tasks' memory demand does not outpace the memory capacity of the VM, while constraint 4 guarantees that the number of parallel tasks allocated to a $vm_j \in M^s$ does not exceed the number of virtual cores of the VM.

$$\sum_{vm_j \in M^s} \sum_{v \in T} X_{ij}^v = 1, \forall i \in B \tag{2}$$

$$\sum_{t_i \in B} \sum_{q=p}^{v} rm_i \times X_{ij}^q \leq m_j, \quad \forall vm_j \in M^s, \forall v \in T, \text{ and } p = \max(v - e_{ij}, 1) \tag{3}$$

$$\sum_{t_i \in B} \sum_{q=p}^{v} X_{ij}^q \leq nvc_j, \quad \forall vm_j \in M^s, \forall v \in T, \text{ and } p = \max(v - e_{ij}, 1) \tag{4}$$

Inequalities 5 and 6 relate the last period of execution of each $vm_j \in M^s$ with the application total execution time (makespan). Finally, constraint 7 ensures that the application makespan does not exceed the $D_{spot}$ value.

$$X_{ij}^v \times (v + e_{ij}) \leq Z_j \quad \forall t_i \in B, \forall vm_j \in M^s \text{ and } \forall v \in T \tag{5}$$

$$X_{ij}^v \times Z_j \leq ZT \quad \forall t_i \in B, \forall vm_j \in M^s \text{ and } \forall v \in T \tag{6}$$

$$Z_j \leq D_{spot}, \forall vm_j \in M^s \tag{7}$$

As shown in Section 4, the execution time to solve the exact model is prohibitive even to small BoT applications. Hence, in order to find good solutions to the scheduling problem in an acceptable time, we adopt the heuristic approach presented in Section 3.2.

### 3.2 Primary Scheduling Module

In order to create the initial scheduling map according to the Mathematical Formulation presented in Section 3.1, we propose an Iterated Local Search (ILS) able to solve the problem in an acceptable time. The ILS [4] is a metaheuristic that aims at improving a final solution by sampling in a broad and distant neighborhood of candidate solutions and then applying a local search technique to refine solutions to their local optima. It explores a sequence of solutions created by perturbations of the current best solution to reach these distant neighborhoods.

Let $M^b$ be the set of burstable on-demand VMs. After finding a scheduling map with the ILS, a second heuristic is applied to include burstable instances from $M^b$ into the solution. In case of spot VM hibernations, the respective instances will be used, in burst mode, by the dynamic scheduler, as an attempt to minimize the impact of these hibernations in the monetary cost and/or the execution time. Therefore, the Primary Task Scheduling algorithm (Algorithm 1) has two parts: i) the Iterated Local Search, that solves the scheduling problem defined in Section 3.1 and ii) the burstable instances allocation, that includes burstable instances to the final solution. Table 2 summarizes the variables and parameters used in the Primary Task Scheduling Algorithms 1, 2 and 3.

Initially, in line 2 of Algorithm 1, an initial solution is generated by calling Algorithm 2, originally proposed in [3]. The latter is a greedy heuristic that schedules the set of tasks $B$ to a set $M^s$ of spot VMs. First, in line 2 of Algorithm 2, all tasks of $B$ are sorted in descending order by their memory size requirements. Then, by calling the procedure *check_schedule*, it verifies if it is possible to schedule task $t_i$ in the selected $vm_j$ of the current phase, i.e., the task is scheduled to the VM if the memory requirements are satisfied and if $D_{spot}$ is not violated. Note that 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 to a $vm_j$ from $A$, the set of already selected VMs (lines 7 to 12). If task $t_i$ cannot be scheduled to a $vm_j$ of $A$, the algorithm tries to select a new spot VM (lines 14 to 23) using a weighted round-robin algorithm (WRR) [20]. In WRR, each spot VM has an associated weight, and the algorithm selects the VMs in a round-robin way, according to such weights.

TABLE 2: Variables and parameters used in Algorithms 1, 2 and 3

| Name | Description |
|---|---|
| $B$ | Set of tasks |
| $M^s$ | Set of spot VMs |
| $M^b$ | Set of on-demand burstable VMs |
| $D$ | Application deadline |
| $D_{spot}$ | Estimated time limit which ensures that there will be enough spare time to migrate tasks |
| $t_i$ | Task $t_i \in B$ |
| $vm_j$ | Virtual machines |
| $max\_iteration$ | Number of iterations of the ILS |
| $max\_failed$ | Tolerated number of iterations without improving the solution |
| $max\_attempt$ | Maximum number of attempts to find a better solution |
| $attempt$ | Current attempt to find a better solution |
| $RD_{spot}$ | Relaxed $D_{spot}$ value |
| $relaxed\_rate$ | Percentage of increment of the $RD_{spot}$ value |
| $it$ | Current iteration of the ILS |
| $it_{best}$ | Last iteration where a best solution was found |
| $swap\_rate$ | Percentage of tasks to be swapped |
| $S$ | A scheduling solution |
| $n$ | Number of burstable VMs added to the final solution |
| $S_{final}$ | The final scheduling solution |
| $burst\_rate$ | Percentage of burstable VMs included into the final solution |
| $A$ | Set of selected VMs |

As shown in Equation 8, 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 time period. In this work, the $Gflops_j$ of a $vm_j$ is used to quantify the computing power of $vm_j$. It is previously estimated using the LINPACK benchmark [21] and allows us to compare the VMs' performance. 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 to increase the availability of spot VM instances. 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 possible VM hibernations. Finally, in line 25, Algorithm 2 calls the function $create\_solution$ that receives the set of the selected VMs $A$ and returns a solution $S$ that defines a scheduling map.

$$weight(vm_j) = Gflops_j / c_j, where\ vm_j \in M \tag{8}$$

After obtaining an initial solution, the ILS algorithm tries to improve it by applying local search and perturbation procedures (Algorithm 1, lines 3 to 20). Thus, let $S$ be a solution that defines a scheduling map of all tasks $t_i \in B$ to a subset of VMs of $M^s$. Let $fitness(S)$ be a weighted function that assigns a value to the quality of $S$. Since this function is equivalent to the objective function presented in Equation 1, we define in Equation 9 the $fitness(S, D_{spot})$

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

**Algorithm 1** *Primary Task Scheduling*

**Input:** $B$, $M$, $M^s$, $M^b$, $max\_iteration$, $max\_attempt$, $max\_failed$, $relaxed\_rate$, $D_{spot}$ **and** $D$

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

function, where $cost$ is the total monetary cost of $S$ and $mkp$ is the total execution time of the application.

$$fitness(S, D_{spot}) = \begin{cases} \infty, \text{if violates } D_{spot} \\ \alpha.cost + (1-\alpha).mkp, otherwise \end{cases} \quad (9)$$

Algorithm 1 executes a local search by calling, in line 3, the $local\_search$ procedure (Algorithm 3), which executes a series of attempts to improve the current solution by swapping tasks between the selected VMs. Algorithm 3 receives as input the current solution $S$, the $max\_attempt$ parameter that determines the number of times the local search will be executed, the set of tasks $B$, the $swap\_rate$ and the $D_{spot}$ value. The $swap\_rate \in \ ]0, 1]$ parameter is tuned before the execution and determines the number of tasks that will be swapped at each iteration. All the parameters used in our experiments, including the $swap\_rate$, $max\_attempt$, $max\_iteration$ and other parameters will be presented in Section 4.

As can be observed in lines 4 and 8 of Algorithm 3, a solution $S$ is composed of two structures: (i) a vector, which controls task allocation, where indexes correspond to tasks, and each element keeps the identity of the VM that will execute the corresponding task, and (ii) a list composed by selected VMs. Firstly, the algorithm computes the number of tasks that will be swapped at each iteration (line 2) and randomly selects a destination VM ($vm_{dest}$, line 4). After that, the algorithm starts the tasks swapping procedure (lines 5 to 14), where $n$ tasks, also randomly selected (line 7), are moved to the $vm_{dest}$. After each swap $movement$, the $local\_search$ procedure checks if the quality of the new generated solution has improved (line 9) and it updates the $S_{best}$ solution, if necessary. In the end, the procedure returns the best solution (line 15).

**Algorithm 2** *Initial Solution*

**Input:** $B$, $M^s$, $D_{spot}$

1:
2: $sort\_by\_memory(B)$ {Sort Tasks $t_i \in B$ according to $rm_i$}
3: $A \leftarrow \emptyset$ {Set of selected VMs}
4: **for all** $t_i \in B$ **do**
5: {P1: Try to schedule the task in an already selected spot VM}
6: $sort\_by\_price(A)$ {Sort VMs $v_j \in A$ according to $c_j$}
7: **for all** $vm_j \in A$ **do**
8: **if** $check\_schedule(t_i, vm_j, D_{spot})$ **then**
9: $schedule(t_i, vm_j)$
10: $break$ {/*Schedule next task*/}
11: **end if**
12: **end for**
13: {P2: Try to schedule the task in a new spot VM}
14: **if not** $scheduled$ **then**
15: {Select a spot VM using the weighted round-robin heuristic}
16: $vm_k \leftarrow weighted\_RR(M^s)$
17: **if** $check\_schedule(t_i, vm_k, D_{spot})$ **then**
18: $schedule(t_i, vm_k)$
19: $A \leftarrow A \cup \{vm_k\}$ {Update the set of selected VMs}
20: $M^s \leftarrow M^s \backslash \{vm_k\}$
21: $break$ {/*Schedule next task*/}
22: **end if**
23: **end if**
24: **end for**
25: $S \leftarrow create\_solution(A)$
26: **Return** $S$ {Returns a scheduling solution $S$}

**Algorithm 3** *Local Search*

**Input:** $S$, $max\_attempt$, $B$, $swap\_rate$ **and** $D_{spot}$

1: $S_{best} \leftarrow S$
2: $n \leftarrow swap\_rate \times |B|$
3: $attempt \leftarrow 0$
4: $vm_{dest} \leftarrow random\_choice(S.selected\_vms)$
5: **while** $attempt < max\_attempt$ **do**
6: **for** $k \in \{1, ..., n\}$ **do**
7: $t_i \leftarrow random\_choice(B)$
8: $S.allocation\_array[t_i] \leftarrow vm_{dest}$
9: **if** $fitness(S, D_{spot}) < fitness(S_{best}, D_{spot})$ **then**
10: $S_{best} \leftarrow S$
11: **end if**
12: **end for**
13: $attempt \leftarrow attempt + 1$
14: **end while**
15: **return** $S_{best}$

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

Upon finishing the ILS (Part 1), Algorithm 1 executes the $burst\_allocation$ procedure (Part 2, lines 22 to 24). In

this procedure, $n$ burstable VMs are included in the final solution, where $n$ is a percentage, given by the parameter $burst\_rate$, of the selected spot VMs of the best solution found by the ILS. For example, if 20 spot VMs were selected by the ILS, with $burst\_rate = 0.1$, only 2 burstable VMs will be included.

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

### 3.3 Dynamic Scheduling Module

The Dynamic Scheduling Module is responsible for performing some actions in response to events, such as spot VM hibernation, resuming, and idleness, that may occur along with the execution.

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

One of the main goals of the Dynamic Scheduling Module is to decide when a VM should terminate. On the one hand, as VMs are charged for each period of time (see Section 3.1), the user has an interest that a VM terminates as soon it becomes *idle*, reducing cost and avoiding extra ones. On the other hand, in some cases, it might be interesting to keep it because it can receive and execute tasks without incurring deploying overhead. To tackle such a trade-off, Burst-HADS has a termination policy where the allocation time is logically divided into units denoted Allocation Cycles (ACs). A $vm_j$ that reaches the end of its current AC, denoted $AC\_cur_j$, in *idle* state, is terminated.

The events and respective actions handled by the Dynamic Scheduling Module are the following:

- $vm_j$ *becomes idle*: When a $vm_j$ finishes the execution of all tasks scheduled to it, its state is updated to *idle* and, if there exist scheduled tasks that have not completely executed, a work-stealing procedure assigning some or all of these tasks to $vm_j$, whose state is, thus, set to *busy* (see Section 3.5).
- *Non-burstable idle* $vm_j$ *reaches the end of* $AC\_cur_j$: If at the end of $AC\_cur_j$ a non-burstable $vm_j$ remains *idle*, it is terminated and removed from the set $IR$ of *idle* VMs and included into $TR$, the set of *terminated* VMs.
- *Spot* $vm_j$ *hibernates*: The $vm_j$'s state is updated to *hibernated* and if it was $busy$, Burst-HADS starts the *burst migration procedure* (see Section 3.4).
- *Spot* $vm_j$ *resumes*: When a hibernated spot $vm_j$ resumes, it is excluded from $HR$, the set of hibernated VMs, and Burst-HADS starts a work-stealing procedure that tries to move tasks from $busy$ VMs to $vm_j$ (see Section 3.5).

Table 3 summarizes the variables and parameters used on Algorithms of the Dynamic Schedule Module (Algorithms 4 and 5).

TABLE 3: Variables and parameters used on Algorithms 4 and 5

| Name | Description |
|---|---|
| $Q_l$ | Set of unfinished tasks of a hibernated spot $vm_l$ |
| $IR$ | Set of *idle* VMs |
| $BR$ | Set of *busy* VMs |
| $M^o$ | Set of on-demand VMs |
| $D$ | Application deadline |
| $cc_j$ | Current amount of CPU credit |
| $rcc_{ij}$ | Estimated number of required CPU credits |
| $burst\_period$ | Number of periods of $T$ corresponding to the use of one CPU credit |
| $K = IR \cup BR$ | Ordered set of idle and busy VMs |
| $start_{ij}$ | The time period a task $t_i$ will start executing when allocated to a $vm_j$ |
| $e_{ij}$ | Time required to execute task $t_i$ in a $vm_j$ |
| $\omega$ | Time overhead to deploy a new VM |
| $vm_j$ *and* $vm_k$ | Virtual machines |
| $ST_j$ | Set of tasks that can be stolen from $vm_j$ |

### 3.4 Burst Migration Procedure

As previously explained, as soon as a spot $vm_l$ hibernates, Burst-HADS executes the migration procedure, searching for a set of VMs to assign and execute non-finished tasks, that were previously scheduled to $vm_l$. These tasks are denoted affected tasks.

Algorithm 4 presents the migration procedure, which always respects the deadline $D$ when selecting tasks to migrate. It receives as input the set $Q_l \subset B$ of affected tasks, the sets of idle, busy, and non-launched regular on-demand VMs ($IR$, $BR$ and $M^o$, respectively), and the deadline $D$. Note that, in the case of a burstable $vm_j \in M^b$, $e_{ij}$ is the execution time of task $t_i$ in $vm_j$ in burst mode (100% of the $vm_j$ processing power) and each $vm_j \in M^b$ have a current CPU credit amount $cc_j$ that is constantly updated by the cloud provider (in the case of $vm_j \notin M^b$, i.e., non-burstable VMs, $cc_j = \infty$).

Initially, $Q_l$ is ordered, giving priority to those tasks that were executing at the moment of the hibernation and had been previously checkpointed (line 1). Burst-HADS applies the checkpoint/recovery mechanism that we proposed and evaluated in our previous work [22] where checkpoints are taken with the help of Checkpoint Restore In Userspace (CRIU) [23], a widely used tool that records the state of individual applications. In order to avoid the overhead of launching new VMs, the migration procedure gives priority to the use of already launched VMs. For each task $t_i \in Q_l$, the algorithm first tries to migrate the task to an *idle* burstable VM (lines 4 to 13). Otherwise, it tries to schedule $t_i$ to one of the non-burstable *idle* or *busy* VM of

**Algorithm 4** *Burst Migration Procedure*

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

```
 1: Q_l ← sort_tasks(Q_l) {/* Prioritizes tasks with checkpoints */}
 2: for each t_i ∈ Q_l do
 3:    {Attempt 1 - Try to migrate task to a Burstable IDLE VM}
 4:    for each burstable vm_j ∈ IR do
 5:       rcc_ij ← ⌈e_ij/burst_period⌉
 6:       if cc_j > rcc_ij and check_migration(t_i, vm_j, D) then
 7:          {Migrate t_i to burstable vm_j on burst mode}
 8:          migrate(t_i, vm_j)
 9:          IR ← IR\{vm_j}
10:          BR ← BR ∪ {vm_j}
11:          break {/*Migrate next task*/}
12:       end if
13:    end for
14:    {Attempt 2 - Try to migrate task to a NON-burstable Idle or Busy
       VM}
15:    if not migrated then
16:       {/* Prioritizes idle spot VMs */}
17:       K ← sort_by_market(IR ∪ BR)
18:       for each NON-burstable vm_j ∈ K do
19:          if check_migration(t_i, vm_j, D) then
20:             migrate(t_i, vm_j)
21:             if vm_j ∈ IR then
22:                IR ← IR\{vm_j}
23:                BR ← BR ∪ {vm_j}
24:             end if
25:             break {/*Migrate next task*/}
26:          end if
27:       end for
28:    end if
29:    {Attempt 3 - Migrate task to a new NON-burstable on-demand
       VM}
30:    if not migrated then
31:       sort_by_price(M^o)
32:       for each vm_j ∈ M^o do
33:          if start_ij + e_ij + ω < D then
34:             start_vm(vm_j)
35:             migrate(t_i, vm_j)
36:             M^o ← M^o\{vm_j}
37:             BR ← BR ∪ {vm_j}
38:             break {/*Migrate next task*/}
39:          end if
40:       end for
41:    end if
42: end for
```

set $K = IR \cup BR$ (lines 18 to 27). Note that, the set $K$ is created by sorting the idle and busy VMs according to its market, putting the spot VMs in the front of the ordered set $K$ (Algorithm 4 line 17).

Tasks migrated to burstable VMs are executed in burst mode, i.e., using 100% of the VM's CPU processing power. Therefore, it is necessary to ensure that a selected burstable VM will have enough CPU credits to execute all tasks assigned to it. Let $burst\_period$ be the number of periods of $T$ corresponding to one credit consumption in burst mode. Since we consider that a task $t_i$ is executed in only one core (see Section 3.1), it is possible to estimate (line 5) the number of CPU credits consumed by task $t_i$, where $e_{ij}$ is the execution time of $t_i$ on $vm_j$ in burst mode, and $rcc_{ij}$ is the estimated number of required CPU credits. Then, the algorithm checks if $vm_j$ has enough CPU credits and call the $check\_migration$ function to guarantee that $vm_j$ has enough memory and will execute $t_i$ before the deadline $D$ (line 6). If both conditions are satisfied, $t_i$ is migrated to $vm_j$ in burst mode (line 8). In this case the burstable $vm_j$ is removed from set $IR$ and included in set $BR$ (lines 9 and 10). Note that the cloud provider continuously updates the number of credits, $cc_j$, of every burstable VM.

Similarly, when Algorithm 4 migrates a task $t_i$ to a non-burstable *idle* or *busy* $vm_j$, it also calls the function $check\_migration$ to verify if the VM has enough memory and will be able to finish the task, respecting the deadline $D$ (line 19). However, if $vm_j$ is a spot VM, the function $check\_migration$ should also verify if there will be enough spare time in $vm_j$ between the end of the execution of $vm_j$ tasks (including task $t_i$) and the deadline $D$ since, in this case, a *busy* or *idle* spot $vm_j$ is also subject to hibernation. The spare time has to be greater than the execution time of the longest task scheduled to $vm_j$, ensuring, therefore, that if a hibernation occurs, there will be enough time to migrate and execute all affected tasks before the deadline $D$.

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

### 3.5 Work-Stealing

The work-stealing procedure, presented in Algorithm 5, aims at reducing the allocation time duration of regular on-demand VMs as well as balancing the load of spot VMs. It is triggered when a hibernated spot VM resumes or when a VM (spot or on-demand) becomes *idle*. Basically, the procedure tries to move tasks from non-burstable *busy* VMs to the *idle* VM.

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

Similarly to the migration procedure, for each selected task of a $vm_j$, the work-stealing procedure also verifies, by calling the function $check\_migration$, if the task migration would result in deadline violation (line 5). Since tasks migrated to burstable VMs by the work-stealing procedure are executed in the baseline mode, after verifying if the *idle* $vm_k$ is burstable (line 6), the algorithm sets up the execution to the baseline mode (line 7). In this work, to set up a burstable VM to the baseline mode means that the task cannot run using 100% of the CPU processing power, but uses only the baseline performance defined by the provider. In order to limit the use of the CPU, Burst-HADS exploits the cpulimit tool[4], an open-source program that limits the CPU usage of a process in GNU/Linux. Moreover, if the *idle* $vm_k$ is burstable, only one task is moved to it, to avoid a queue of tasks to be executed in the baseline mode, which could increase the application total execution time. Since stolen tasks are executed in baseline mode on burstable VMs non consumed CPU credits consuming keep available if a spot VM hibernates. Thus, after moving one task to the burstable $vm_k$, the algorithm stops the loop (line 9). Finally, if at least one task is migrated to $vm_k$, its state changes to *busy* and,

4. http://cpulimit.sourceforge.net/

consequently, it is included into the set of *busy* VMs and removed from the *idle* set (lines 17 and 18).

**Algorithm 5** *Burst Work-Stealing Procedure*

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

## 4 Experiments and Results

In this section, we first evaluate the ILS, part one of Algorithm 1, in terms of quality of solution and execution time, by comparing it with the solutions given by the mathematical formulation presented in Section 3.2 (solved with Gurobi 9.1) and with three commonly used BoT scheduling heuristics [24], [25], [26]: MinMin, MaxMin and Greedy (Section 4.1). We also present a evaluation in AWS EC2 (Section 4.2), where the jobs were executed in a real cloud environment. In the experiments, we have considered two distinct workloads: i) synthetic BoT applications composed by tasks generated with the application template proposed in Alves *et al.* [27], and ii) the ED application available in the GRIDNBP 3.1 suite of NAS benchmark [28]. The tasks of the synthetic applications execute vector operations whose execution times depend on the size of the vectors.

According to AWS, only spot VMs of types C3, C4, C5, M4, M5, R3, and R4 with memory amount smaller than 100 GB are hibernation-prone. Therefore, in our experiments, we used spot VMs C3 and C4, which are compute-optimized and have high availability in the spot market. Moreover, we also use the T3.large instances that are the newest generation of the general proposed burstable instances of EC2. Table 4 shows the computational characteristics and the corresponding prices in on-demand and spot markets in November 2020.

TABLE 4: VMs attributes

| Type | #VCPUs | Memory | Price per Hour (USD) | | Baseline performance |
|---|---|---|---|---|---|
| | | | on-demand | spot | |
| C3.large | 2 | 3.75 GB | 0.105$ | 0.0299$ | - |
| C4.large | 2 | 3.75 GB | 0.100$ | 0.0366$ | - |
| C3.xlarge | 4 | 7.50 GB | 0.199$ | 0.0634$ | - |
| T3.large | 2 | 8 GB | 0.0832$ | - | 20% |

For the execution of Algorithms 1 and 3, the following input parameters were used: $\alpha = 0.5$, $max\_iteration = 200$; $max\_attempt = 50$; $swap\_rate = 0.10$; $max\_failed = 20$; $relaxed\_rate = 0.25$; $burst\_rate = 0.2$. Moreover, we defined $AC = 900$ seconds. Except for $\alpha = 0.5$, which was used to give the same weights to both objectives, the parameters' values were defined by executing a set of empirical tests. As mentioned in Section 3.4, Burst-HADS uses a checkpoint approach proposed in [3] where the user defines the parameter $ovh$ as the maximum percentage of time overhead that the checkpoint mechanism is allowed to add to the original execution time of a task. From this parameter, Burst-HADS defines the time interval between checkpoints. In this work, $ovh = 10\%$ for all experiments.

### 4.1 Evaluation of the ILS Primary Task Scheduling

To evaluate the quality of the solution given by the ILS executed in the first part of the primary scheduling heuristic (Algorithm 1), we compared it with the optimal solution given by the exact model and also with the solutions given by three widely used benchmark scheduling heuristics [5]: (i) MinMin, where the scheduling gives priority to VMs with the minimum earliest completion time; (ii) MaxMin, where the scheduling gives priority to VMs with the maximum earliest completion time; and (iii) Greedy, where the shortest execution time task is scheduled to the cheapest available VM.

For the comparison with the exact model, we create six small synthetic jobs (from 5 to 30 tasks), composed of tasks with execution times varying from 5 to 15 seconds. We have chosen this set with up to 30 tasks because the exact model took more than ten hours to solve the problem for jobs with more than 30 tasks. In these experiments, the deadline was fixed in 15 minutes ($D = 900$) for all jobs, and we used as input the spot VMs presented in Table 4, except for the burstable VM, since we are evaluating just the ILS heuristic. The experiments were executed on a computer with a processor Intel Core i7-3770 CPU 3.40 GHz with 12GB of memory and Ubuntu 18.04 was used. The mathematical formulation was solved by using the Gurobi Solver 9.0[5].

Table 5 summarizes the results with both the exact approach and the ILS-based heuristic. The first column identifies the number of tasks. The three columns that follow correspond to the results achieved by ILS: makespan, monetary cost, and the execution time to obtain the solution. The next three columns present the same results for the exact approach. The values shown for the ILS are averages of three executions. The ILS standard deviations were zero for all jobs.

TABLE 5: Results of the ILS-based Primary Scheduling Heuristic and the Exact Approach

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

As we can observe in Table 5, ILS obtains the same solutions of the exact approach for these small jobs, taking significantly less time than the mathematical formulation.

5. http://www.gurobi.com

On average, the ILS takes $0.048$s against $2101.43$s of the exact approach. Although these results are very encouraging, the experiments were limited to jobs with few tasks (up to 30 tasks) due to the huge required time to obtain the exact solution for jobs with more tasks.

For the evaluation with the other heuristics, six synthetic jobs were created with sizes varying from 50 to 300 tasks. The tasks execution times varied from $15$ to $35$ seconds, and the same deadline of 15 minutes ($D = 900$) was applied. Table 6 presents the makespan and monetary cost obtained by the ILS, MinMin, MaxMin, and Greedy heuristics. As shown in Table 6, the ILS heuristic outperforms MinMin in terms of monetary cost but increases the makespan. On average, the ILS makespan is increased of 18.11%, while the average monetary cost is reduced by more than 38.00%.

TABLE 6: Results of the ILS-based Primary Scheduler, MinMin, MaxMin and Greedy Heuristics.

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

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

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

### 4.2 Evaluation of Burst-HADS on AWS EC2

For the experiments, several synthetic tasks were created, each one with a memory footprint between 2.81 MB and 13.19 MB, resulting in execution times which vary from 102 to 330 seconds. Then, we conceived three synthetic BoT applications, J60, J80, and J100 by randomly selecting the tasks. Concerning the ED application, a real embarrassingly distributed application, the job ED200 comprises 200 tasks solving the largest problem size (class B), taking 211 seconds per task on average.

Table 7 shows the four jobs' features, including their respective number of tasks, memory footprint and tasks runtime. In our experiments, we considered the shortest deadline in which Burst-HADS can find a feasible solution for all evaluated jobs, $D = 2700$ seconds (45 minutes), respecting the constraints. For some jobs, it was a tight deadline, while for other ones it was not. This approach allowed us to observe the behavior of the framework in those different scenarios.

TABLE 7: Characteristics of the Jobs

| job | # tasks | memory footprint | | | task runtime(s) | | |
|---|---|---|---|---|---|---|---|
| | | min | avg | max | min | avg | max |
| J60 | 60 | 2.85MB | 4.69MB | 12.20MB | 102 | 198 | 323 |
| J80 | 80 | 2.91MB | 4.71MB | 13.19MB | 103 | 199 | 322 |
| J100 | 100 | 2.81MB | 4.49MB | 10.86MB | 107 | 190 | 330 |
| ED200 | 200 | 153.74MB | 168.68MB | 177.77MB | 161 | 211 | 354 |

We have firstly evaluated Burst-HADS in a scenario without hibernation, comparing it with (1) the schedule given by the proposed ILS using only on-demand VMs; (2) the schedule given by HADS, and (3) the schedule given by AutoBoT-like. The aim of these experiments is to measure the impact of including burstable on-demand VMs into the scheduling procedure in both the monetary cost and execution time. Table 8 presents the average of three executions of the synthetic jobs J60, J80, and J100, and the real application ED200, for each case.

AutoBoT-like is inspired by the work presented in [7] (see Section 2). AutoBoT-like uses the same heuristics, procedures and parameters defined in [7], but the price history and the spot market bid are not considered in that adapted version. In our implementation, we use the current price model of AWS, which is more stable.

Initially, AutoBoT-like, as in the original version, considers a critical time point beyond which it is not possible to complete the remaining tasks without violating the deadline constraint, in case of a spot VM revocation. Thus, when a task, running on a spot VM, reaches this point, it is immediately migrated to an on-demand VM. The number of launched on-demand VMs in this case is defined in accordance with a sweep heuristic, based on the following parameters: the number of available on-demand VMs, the deadline and the total number of tasks of the application.

In the second step, along with the execution of the application, AutoBot-like, as in the original version, takes checkpoints and migrate tasks to on-demand VMs. Although the authors propose three different strategies for checkpointing, we adopted the optimistic one, where checkpoints are recorded just before the critical time point, because the other ones depended on the price history and they did not present significant advantages.

In comparison with HADS, Table 8 shows that Burst-HADS reduces the makespan in 44.37%, 42.09%, 28.82%, and 11.82%, for jobs J60, J80, J100, and ED200, respectively. However, the average monetary cost increases by 66.34%, 44.54%, 57.55%, and 33.71%, for the same comparison. It happens because Burst-HADS already starts by using some burstable on-demand VMs. Moreover, the ILS based primary scheduling uses more VMs to reduce the execution time, while in HADS, the initial scheduling aims at minimizing only the monetary cost.

As can also observe in in the same table, in the executions without spot revocations, Burst-HADS presents an average reduction of 20.92% in the monetary cost and 31.11% in the makespan, when compared to AutoBoT-like. Such a result is expected since AutoBoT-like launches additional on-demand instances even without spot revocations when the BoT execution gets the critical time point.

On the other hand, compared to the ILS on-demand strategy, on average, Burst-HADS reduces the monetary cost by more than 52.00%, with an average increase of 15%

TABLE 8: Cost and Makespan of Burst-HADS, HADS (without hibernation), AutoBoT-like (without spot interruptions) and ILS On-demand only.

| JOB | Burst-HADS w/o Hibernation | | HADS w/o Hibernation | | AutoBoT-like w/o Interruptions | | ILS On-demand | |
|---|---|---|---|---|---|---|---|---|
| | cost | makespan | cost | makespan | cost | makespan | cost | makespan |
| J60 | $0.112 | 1274 | $0.067 | 2290 | $0.166 | 2221 | $0.271 | 1112 |
| J80 | $0.151 | 1329 | $0.104 | 2295 | $0.199 | 2266 | $0.312 | 1190 |
| J100 | $0.176 | 1660 | $0.112 | 2332 | $0.218 | 2342 | $0.371 | 1462 |
| ED200 | $0.357 | 2275 | $0.267 | 2580 | $0.387 | 2566 | $0.698 | 1887 |

in the makespan. The ILS on-demand strategy uses the scheduling plan given by the ILS proposed in Algorithm 1, which includes neither spot nor burstable VMs, but only regular on-demand VMs. The longer makespan is due to the execution of tasks in the baseline mode of burstable VMs, which does not occur in the ILS on-demand strategy.

Since cloud users have no control over spot VM hibernations, we have emulated different patterns of the events of spot hibernations and resume by using a Poisson distribution function. We have applied the Poisson function to modeling several scenarios where *hibernating* and *resuming* events have different probability mass functions defined by the parameters $\lambda_h$ and $\lambda_r$, respectively. Let the $\lambda$ 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, $\lambda_h$ and $\lambda_r$ parameters are given by $\lambda_h = k_h/D$ and $\lambda_r = k_r/D$. Table 9 presents five different scenarios by varying $k_h$ and $k_r$. Note that, since AutoBoT-like does not explore the hibernation feature of the spot VMs, instead of emulating the hibernation, the spot instances are terminated according to the Poisson function with probability mass functions also defined by the parameters $\lambda_h$.

TABLE 9: Different execution scenarios generated by varying parameters $\lambda_h$ and $\lambda_r$

| ID | *hibernating* | *resuming* | $\lambda_h$ | $\lambda_r$ |
|---|---|---|---|---|
| $sc_1$ | $k_h = 1$ | $k_r = 0$ | $1/2700$ | $0/2700$ |
| $sc_2$ | $k_h = 5$ | $k_r = 0$ | $5/2700$ | $0/2700$ |
| $sc_3$ | $k_h = 1$ | $k_r = 5$ | $1/2700$ | $5/2700$ |
| $sc_4$ | $k_h = 5$ | $k_r = 5$ | $5/2700$ | $5/2700$ |
| $sc_5$ | $k_h = 3$ | $k_r = 2.5$ | $3/2700$ | $2.5/2700$ |

Table 10 presents the averages of three executions of jobs J60, J80, J100, and ED200 using Burst-HADS, HADS and AutoBoT-like in each of the five execution scenarios. For each job and scenario, the table shows the average number of hibernations/revocations followed by resume events. It also includes the number of non-burstable on-demand VMs launched during the frameworks' execution and the average of monetary cost and makespan. Finally, the last four columns present the percentage differences related to the monetary cost and the makespan between Burst-HADS and HADS (Diff HADS), and between Burst-HADS and AutoBoT-like (Diff AutoBoT-like).

As shown in Table 10, in comparison to HADS, Burst-HADS minimizes the makespan in all execution scenarios, presenting an average reduction of 25.87%. As explained in Section 3.4, whenever a spot VM hibernates, Burst-HADS immediately migrates the interrupted tasks to other VMs. Thus, the increase in the makespan is due to the overhead of this procedure, which might include the launch of new VMs. However, in these scenarios, the overhead has a lower impact in small jobs, i.e., jobs with fewer tasks, than in the ones with a higher number of tasks. For example, while for job J60, the average makespan reduction, considering all scenarios, is 40.10%, for job ED200, that reduction is only 10.24%. Such a behavior can be explained because, in our experiments, we have fixed the same deadline for all jobs and, therefore, small jobs have more spare time between its expected makespan defined by the ILS and the deadline. Consequently, in this case, Burst-HADS benefits more from the burst mode of the burstable VMs since it has more idle time to earn CPU credits. Moreover, it also executes the work-stealing more frequently, which also reduces the makespan. On the other hand, independently of the scenario or job, HADS's makespan gets closer to the deadline whenever a hibernation occurs. That happens because the HADS framework postpones as much as possible the execution of the migration procedure. Since HADS gives priority to the monetary cost saving, its central idea is to wait as much as possible for the resume of hibernated VMs in order to avoid the launch of new VMs.

Still compared to HADS, Burst-HADS improves the monetary cost for all jobs in scenarios $sc_2$ and $sc_5$. The $sc_2$ is the worst case scenario, since it has the highest rate of hibernation ($k_h = 5$) and no resume ($k_r = 0$), while $sc_5$ is the average case scenario where the rate of hibernation is $k_h = 3.0$, and the rate of resume is $k_h = 2.5$ (see Table 9). In these scenarios, Burst-HADS uses fewer regular on-demand VMs than HADS. Moreover, in both cases, the number of hibernations is higher than the number of resumes. Hence, in those cases, by migrating tasks to busy and idle VMs and burstable VMs, exploring the burst mode, Burst-HADS is more effective in minimizing the impact of spot hibernations than HADS. It is also worth pointing out that, considering all executions, the average increase of Burst-HADS monetary cost is 1.92% compared to HADS.

Compared to AutoBoT-like, Burst-HADS can reduce the monetary cost for all jobs in almost all scenarios, presenting an average reduction of 11.65%. In our evaluation, AutoBoT-like reduced the monetary cost only in scenario $sc_2$ and for the jobs J60 and ED200. As explained before, this scenario has the highest rate of hibernation ($k_h = 5$), but there are no resume events ($k_r = 0$). Therefore, in this case, HADS and Burst-HADS cannot explore the resumed spot VMs to minimize the monetary cost. Although Burst-HADS is more effective than HADS to minimize that impact by using burstable VMs, for these two jobs, AutoBoT-like migrated the majority of tasks just before the VMs interruptions and selected a cheap set of on-demand VMs able to execute the remaining tasks. However, we should point out that the AutoBot-like approach is unsuitable for the other scenarios since it does not take advantage of the hibernation feature. In terms of makespan, Burst-HADS presented a better result in almost all cases when compared to AutoBoT-like, with an average reduction of 21.08%. The only exception for the makespan was scenario $sc_4$ for job J80, where there were many hibernations and resumes, which increased the

TABLE 10: Comparison between Burst-HADS, HADS and AutoBoT-like in terms of monetary cost and makespan in scenarios $sc_1$ to $sc_5$

| Job | scenario | # hibernations | # resumes | # used regular on-demand VMs | | | Burst-HADS | | HADS | | AutoBoT-like | | Diff HADS (%) | | Diff AutoBoT-like (%) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Burst-HADS | HADS | AutoBoT-like | cost | makespan | cost | makespan | cost | makespan | cost | makespan | cost | makespan |
| J60 | $sc_1$ | 0.66 | 0.00 | 0.00 | 0.00 | 1 | $0.119 | 1274 | $0.091 | 2620 | $0.166 | 2221 | -30.77% | 51.37% | 28.31% | 42.64% |
| | $sc_2$ | 3.33 | 0.00 | 1.33 | 2.33 | 1 | $0.204 | 1277 | $0.257 | 2549 | $0.173 | 2228 | 20.54% | 49.90% | -17.92% | 42.68% |
| | $sc_3$ | 2.33 | 2.33 | 1.33 | 0.00 | 1 | $0.127 | 1752 | $0.101 | 2539 | $0.178 | 2237 | -26.07% | 31.00% | 28.46% | 21.68% |
| | $sc_4$ | 5.33 | 4.00 | 1.67 | 0.00 | 1 | $0.142 | 1857 | $0.119 | 2634 | $0.180 | 2252 | -19.90% | 29.50% | 21.07% | 17.54% |
| | $sc_5$ | 2.66 | 1.00 | 1.33 | 2.00 | 1 | $0.150 | 1445 | $0.169 | 2359 | $0.170 | 2236 | 11.44% | 38.75% | 11.96% | 35.38% |
| J80 | $sc_1$ | 1.00 | 0.00 | 1.33 | 0.33 | 2 | $0.167 | 1419 | $0.150 | 2581 | $0.206 | 2273 | -11.33% | 45.03% | 18.93% | 37.57% |
| | $sc_2$ | 5.00 | 0.00 | 1.00 | 3.00 | 2 | $0.210 | 2267 | $0.298 | 2591 | $0.211 | 2278 | 29.48% | 12.50% | 0.47% | 0.48% |
| | $sc_3$ | 3.00 | 1.00 | 1.67 | 1.00 | 2 | $0.164 | 1367 | $0.147 | 2602 | $0.214 | 2276 | -11.34% | 47.46% | 23.52% | 39.94% |
| | $sc_4$ | 9.66 | 7.66 | 1.00 | 2.00 | 3 | $0.244 | 2488 | $0.212 | 2607 | $0.301 | 2460 | -15.25% | 4.56% | 18.83% | -1.14% |
| | $sc_5$ | 3.00 | 1.00 | 1.33 | 3.00 | 2 | $0.195 | 1589 | $0.246 | 2529 | $0.228 | 2266 | 20.47% | 37.17% | 14.25% | 29.88% |
| J100 | $sc_1$ | 2.00 | 0.00 | 0.00 | 0.00 | 2 | $0.191 | 1798 | $0.157 | 2332 | $0.226 | 2350 | -21.76% | 22.90% | 15.49% | 23.49% |
| | $sc_2$ | 7.00 | 0.00 | 1.33 | 3.00 | 2 | $0.212 | 1900 | $0.353 | 2518 | $0.222 | 2342 | 39.94% | 24.54% | 4.50% | 18.87% |
| | $sc_3$ | 6.00 | 3.00 | 1.67 | 1.00 | 2 | $0.201 | 1925 | $0.166 | 2636 | $0.224 | 2360 | -21.08% | 26.97% | 10.27% | 18.43% |
| | $sc_4$ | 11.00 | 9.00 | 1.00 | 0.00 | 3 | $0.286 | 2453 | $0.278 | 2591 | $0.312 | 2677 | -2.88% | 5.33% | 8.33% | 8.37% |
| | $sc_5$ | 3.66 | 2.00 | 1.00 | 2.50 | 2 | $0.166 | 1547 | $0.189 | 2543 | $0.227 | 2366 | 12.49% | 39.15% | 26.98% | 34.62% |
| ED200 | $sc_1$ | 3.00 | 0.00 | 1.00 | 0.33 | 3 | $0.388 | 2327 | $0.314 | 2680 | $0.414 | 2630 | -23.57% | 13.17% | 6.28% | 11.52% |
| | $sc_2$ | 8.00 | 0.00 | 2.00 | 5.00 | 3 | $0.482 | 2448 | $0.512 | 2676 | $0.430 | 2661 | 5.86% | 8.52% | -12.09% | 8.00% |
| | $sc_3$ | 6.66 | 4.00 | 2.33 | 1.00 | 3 | $0.427 | 2345 | $0.387 | 2672 | $0.457 | 2675 | -10.34% | 12.24% | 6.56% | 12.34% |
| | $sc_4$ | 9.00 | 6.00 | 2.00 | 1.00 | 3 | $0.411 | 2560 | $0.389 | 2690 | $0.447 | 2667 | -5.66% | 4.83% | 8.05% | 4.01% |
| | $sc_5$ | 4.33 | 2.33 | 1.67 | 3.00 | 3 | $0.367 | 2342 | $0.467 | 2674 | $0.411 | 2765 | 21.41% | 12.42% | 10.71% | 15.30% |

overhead of Burst-HADS due to migrations and task work-stealing. But even in this case, the makespan difference was only 1.14%.

Finally, compared to the ILS On-demand, both Burst-HADS and HADS minimize the monetary cost for all execution cases, presenting an average reduction of 41.80% and 39.65%, respectively. For job ED200, for example, in the worst execution scenario, $sc_2$, Burst-HADS reduces the monetary cost by 30.96%, while HADS presents an average reduction of 26.66%.

Since cloud environments can face random fluctuation in terms of performance, we also carried out some experiments considering an increase of 20% in the execution time of the tasks of job J60. We observed in Table 11 that both frameworks, AutoBoT-like and Burst-HADS, were able to meet the deadlines for all scenarios. However, just Burst-HADS kept the monetary cost below the on-demand only solution ($0.271). However, for tight deadlines and depending on the range of the fluctuations, the frameworks might not meet the deadlines.

TABLE 11: Comparison of Burst-HADS and AutoBoT-like with an unexpected increment of 20% in the execution time of the tasks of J60

| With Time Fluctuation (20%) | | | | Without Fluctuation | | | |
|---|---|---|---|---|---|---|---|
| Burst-HADS | | AutoBoT-like | | Burst-HADS | | AutoBoT-like | |
| cost | makespan | cost | makespan | cost | makespan | cost | makespan |
| $0.199 | 1429 | $0.269 | 2408 | $0.112 | 1274 | $0.166 | 2221 |
| $0.231 | 1493 | $0.274 | 2424 | $0.119 | 1274 | $0.166 | 2221 |
| $0.254 | 2595 | $0.295 | 2453 | $0.204 | 1277 | $0.173 | 2228 |
| $0.227 | 1834 | $0.282 | 2457 | $0.127 | 1752 | $0.178 | 2237 |
| $0.225 | 1833 | $0.276 | 2439 | $0.142 | 1857 | $0.180 | 2252 |
| $0.195 | 1648 | $0.279 | 2483 | $0.150 | 1445 | $0.170 | 2236 |

## 5 Conclusion and Future Work

This paper proposes the Burst Hibernation-Aware Dynamic Scheduler (Burst-HADS) framework which uses regular on-demand VMs, hibernation-prone spot VMs, and burstable VMs for executing deadline constrained bag-of-task applications. The framework creates a primary scheduling plan using an ILS based heuristic and then executes a Dynamic Scheduling Module that monitors the execution, applying both task migration and work-stealing techniques whenever necessary. The objective of Burst-HADS is to minimize both the makespan and the monetary cost of the execution, respecting applications deadline even when spot VMs in use hibernate.

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

Burst-HADS was evaluated in a real environment using VMs of AWS EC2, considering scenarios with different hibernation and resuming rates. When compared to the ILS On-demand approach that uses only regular on-demand VMs, Burst-HADS reduces the monetary cost for all execution scenarios at the expense of slightly longer makespans due to the migration overhead. Moreover, compared to HADS, our previous proposed framework that uses only spot and on-demand VMs, Burst-HADS reduces the makespan by more than 25%, with an average increase of only 1.92% in the monetary cost. Burst-HADS was also compared to AutoBoT-like, an approach inspired by Vashney and Simmhan's work. AutoBoT-like also combines spot and on-demand VMs but does not explore the hibernation feature. Our results show that, in this case, Burst-HADS reduces the average monetary cost and the makespan by 11.65% and 21.08%, respectively.

As future work, we would like to include in the migration decision of Burst-HADS some prediction of the future state of the spot market, considering, for example, the historic of VMs hibernation. A second research direction would focus on the selection of burstable VMs, with a deeper analysis of the most suitable one, based on the tradeoff between price and performance, in different execution scenarios. Finally, we intend to propose a more realistic scheduler which does not require precise knowledge of the execution time of the application as input, and that may launch or eliminate VMs in accordance with the real application behavior.

## Acknowledgment

This research is supported by project CNPq/AWS 440014/2020-4, Brazil and by the *Programa Institucional de Internacionalização* (PrInt)

from CAPES as part of the project REMATCH (process number 88887.310261/2018-00).

## REFERENCES


[1] A. W. Services, "Burstable performance instances," https://docs.aws.amazon.com/burstable-performance-instances.html, 2021, accessed June 2021.

[2] L. Thai, B. Varghese, and A. Barker, "A survey and taxonomy of resource optimisation for executing bag-of-task applications on public clouds," *Future Generation Comp. Syst.*, vol. 82, pp. 1–11, 2018.

[3] L. Teylo, L. Arantes, P. Sens, and L. M. Drummond, "A dynamic task scheduler tolerant to multiple hibernations in cloud environments," *Cluster Computing*, pp. 1–23, 2020.

[4] H. R. Lourenço, O. C. Martin, and T. Stützle, "Iterated local search," in *Handbook of metaheuristics*. Springer, 2003, pp. 320–353.

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

[6] A. W. Services. (2018) Amazon ec2 instance types. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[7] P. Varshney and Y. Simmhan, "Autobot: Resilient and cost-effective scheduling of a bag of tasks on spot vms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 7, pp. 1512–1527, 2019.

[8] P. Leitner and J. Scheuner, "Bursting with possibilities–an empirical study of credit-based bursting cloud instance types," in *IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, 2015, pp. 227–236.

[9] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Cedule: A scheduling framework for burstable performance in cloud computing," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2018, pp. 141–150.

[10] Y. Jiang, M. Shahrad, D. Wentzlaff, D. H. Tsang, and C. Joe-Wong, "Burstable instances for clouds: Performance modeling, equilibrium analysis, and revenue maximization," in *IEEE INFOCOM Conference on Computer Communications*, 2019, pp. 1576–1584.

[11] A. F. Baarzi, T. Zhu, and B. Urgaonkar, "Burscale: Using burstable instances for cost-effective autoscaling in the public cloud," in *ACM Symposium on Cloud Computing*, 2019, pp. 126–138.

[12] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang, "Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud," in *Twelfth European Conference on Computer Systems*, 2017, pp. 620–634.

[13] S. Subramanya, T. Guo, P. Sharma, D. E. Irwin, and P. J. Shenoy, "Spoton: a batch computing service for the spot market," in *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, 2015, pp. 329–341.

[14] I. Menache, O. Shamir, and N. Jain, "On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud," in *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014.*, 2014, pp. 177–187.

[15] P. Sharma, S. Lee, T. Guo, D. E. Irwin, and P. J. Shenoy, "Spotcheck: designing a derivative iaas cloud on the spot market," in *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, 2015, pp. 16:1–16:15.

[16] L. Teylo, L. Arantes, P. Sens, and L. M. d. A. Drummond, "A bag-of-tasks scheduler tolerant to temporal failures in clouds," in *31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2019, pp. 144–151.

[17] T.-P. Pham and T. Fahringer, "Evolutionary multi-objective workflow scheduling for volatile resources in the cloud," *IEEE Transactions on Cloud Computing*, 2020.

[18] J. Fabra, J. Ezpeleta, and P. Álvarez, "Reducing the price of resource provisioning using ec2 spot instances with prediction models," *Future Generation Computer Systems*, vol. 96, pp. 348–367, 2019.

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

[20] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted round-robin cell multiplexing in a general-purpose atm switch chip," *IEEE Journal on selected Areas in Communications*, vol. 9, no. 8, pp. 1265–1279, 1991.

[21] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[22] L. Teylo, R. C. Brum, L. Arantes, P. Sens, and L. M. d. A. Drummond, "Developing checkpointing and recovery procedures with the storage services of amazon web services," in *49th International Conference on Parallel Processing-ICPP: Workshops*, 2020, pp. 1–8.

[23] P. EMELYANOV, "Criu: Checkpoint/restore in userspace, july 2011," *URL: https://criu. org*, 2011.

[24] N. Fujimoto and K. Hagihara, "A comparison among grid scheduling algorithms for independent coarse-grained tasks," in *2004 International Symposium on Applications and the Internet Workshops. 2004 Workshops.* IEEE, 2004, pp. 674–680.

[25] K. Li, "Job scheduling and processor allocation for grid computing on metacomputers," *Journal of Parallel and Distributed Computing*, vol. 65, no. 11, pp. 1406–1418, 2005.

[26] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *Journal of parallel and distributed computing*, vol. 59, no. 2, pp. 107–131, 1999.

[27] M. M. Alves and L. M. de Assumpção Drummond, "A multivariate and quantitative model for predicting cross-application interference in virtual environments," *Journal of Systems and Software*, vol. 128, pp. 150 – 163, 2017.

[28] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.


**Luan Teylo** Luan Teylo received his Ph.D. in Computer Science in 2021 from Fluminense Federal University (Brazil). He graduated in Computer Science in 2015 and obtained a master's degree in 2017. In 2019 he did a doctorate sandwich in the LiP6 lab at Sorbonne Université, where he worked with scheduling and fault tolerance on the cloud. His research interests include I/O, distributed algorithms, cloud computing, and scheduling problems.

**Luciana Arantes** Luciana Arantes received her Ph.D. in Computer Science in 2000 from Paris 6 University (UPMC), France. She has been working as an associate professor at Sorbonne University (ex-UPMC) since 2001. She was a research member of INRIA/LIP6 Regal project-team from 2005–2017 and now research member of INRIA/LIP6 Delys group. She was member of the Program Committee of several conferences in the area of distributed systems and parallelism (ICDCS, EDCC, ISSRE, NCA, SBAC-PAD, DAIS, etc.) and local organized chair of SBAC-PAD 2014 and EDCC 2015.

**Pierre Sens** Pierre Sens obtained his Ph. D. in Computer Science in 1994, and the "Habilitation à diriger des recherches" in 2000 from Paris 6 University, France. Currently, he is a full Professor at Sorbonne Université. He was a member of the Program Committee of major conferences in the areas of distributed systems and parallelism (DISC, ICDCS, IPDPS, OPODIS, ICPP, Europar,...) and serves as General chair of SBAC and EDCC. Overall, he has published over 150 papers in international journals and conferences and has acted for advisor of 24 PhD theses.

**Lúcia M. A. Drummond** Lúcia M. A. Drummond obtained her D.Sc. in Systems Engineering and Computer Science from the Federal University of Rio de Janeiro, Brazil, in 1994, where she took part of the group which developed the first Brazilian parallel computer. She is in the Department of Computer Science of the Fluminense Federal University (UFF) since 1989, where she is now Full Professor. She was member of the Program Committee of major conferences in the areas of high performance computing and parallelism (ICS, CLUSTER, CCGRID, Europar, SBAC,...) and served as program chair of SBAC twice, and guest editor of JPDC.