A Hibernation Aware Dynamic Scheduler for Cloud Environments

Luan Teylo  
Fluminense Federal University  
Niterói, Brazil  
luanteylo@ic.uff.br

Pierre Sens  
Sorbonne Université, CNRS, INRIA, LIP6  
Paris, France  
pierre.sens@lip6.fr

Luciana Arantes  
Sorbonne Université, CNRS, INRIA, LIP6  
Paris, France  
luciana.arantes@lip6.fr

Lúcia Maria de A. Drummond  
Fluminense Federal University  
Niterói, Brazil  
lucia@ic.uff.br

ABSTRACT

Nowadays, cloud platforms usually offer several types of Virtual Machines (VMs) which have 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 hour for on-demand VMs while spot VMs are unused instances available for a lower price. Despite the monetary advantages, a spot VM can be terminated or hibernated by EC2 at any moment. In this work, we propose the Hibernation-Aware Dynamic Scheduler (HADS), to schedule applications composed of independent tasks (bag-of-tasks) with deadline constraints in both hibernation-prone spot VMs (for cost sake) and on-demand VMs. We also consider the problem of temporal failures, that occurs when a spot VM hibernates, and does not resume within a time that guarantees the application’s deadline. Our dynamic scheduling approach aims at minimizing the monetary costs of bag-of-tasks applications execution, respecting its deadline even in the presence of hibernation. It is also able to avoid temporal failures, by using task migration and work-stealing techniques. Experimental results with real executions using Amazon EC2 VMs confirm the effectiveness of our scheduling when compared with on-demand VM only based approaches, in terms of monetary costs and execution times. It is also shown that our strategy can tolerate temporal failures.

CCS CONCEPTS

• Computer systems organization → Reliability; Availability; Maintainability and maintenance.

KEYWORDS

Clouds, Temporal failures, Hibernation, Dynamic Scheduling

ACM Reference Format:


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPP 2019, August 5–8, 2019, Kyoto, Japan  
© 2019 Association for Computing Machinery.  
ACM ISBN 978-1-4503-7196-4/19/08  
https://doi.org/10.1145/3339186.3339205

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: the on-demand and the spot markets.

On-demand VMs do not require a long-term commitment and can be deployed at any time. Those VMs have high availability and cannot be interrupted by the provider. On the other hand, spot VMs are unused EC2 capacity offered by Amazon with a huge discount (according to Amazon it can be up to a 90% discount compared to on-demand prices). But it can be interrupted by the provider when it needs the resources back.

In December 2017, Amazon EC2 changed the spot VMs pricing model. Before that, to contract a spot VM the user had to give a bid and the VM was allocated only if its current price was smaller than that bid. If during the execution, the VM price increased and became larger than the bid, the VM was interrupted. Before the new price model, the spot market prices faced constant variations because it was driven by the users’ bids. With the new price model, the client does not have to give a bid anymore to contract spot VMs, and the prices are stable. In addition to prices stability, the number of interruptions decreased and Amazon EC2 also included the hibernation feature on the spot VMs. So, now the provider can hibernate a spot VM instead of terminating that. When a spot VM is hibernated, its memory and context are saved and, when the cloud demand reduces, the VM is resumed by the provider, and all the tasks that were executing at the moment of the hibernation restart from the breakpoint. Note that the user is not charged for the periods of the hibernation.

In this paper, we are interested in applications composed of independent tasks which can be executed in any order and in parallel, called bag-of-task (BoT). Although simple, the BoT approach is used
by several applications such as parameter sweep, chromosome mapping, Monte Carlo simulation and computer imaging applications. Furthermore, they may require deadline-bounds where the correctness of the computation also depends on the time the computation of all tasks ends.

In this work, we propose a dynamic cloud scheduler for Bag-of-Tasks applications using, for cost sake, hibernation-prone spot VMs while also minimizing the monetary costs. Our strategy is able to avoid the problem of temporal failures, that occurs when a spot VM hibernates, and does not resume within a time that guarantees the application’s deadline, by using task migration and work-stealing techniques. Experimental results with real executions using Amazon EC2 VMs confirm the effectiveness of our scheduling when compared with on-demand VM only based approaches, in terms of monetary costs and execution times. It is also shown that our strategy can tolerate temporal failures.

So, the main contributions of this work are:

- Definition of the BoT scheduling problem to hibernation-prone spot VMs
- Development of the Hibernation-aware Dynamic Scheduler (HADS)
- Evaluation of HADS in different scenarios of VMs hibernation and resuming

The remaining of the paper is organized as follows. Section 2 discusses some related work. Section 3 presents the problem definition, and Section 4 describes our proposed dynamic scheduler. Evaluation results conducted in real scenarios are presented in Section 5. Finally, Section 6 concludes the paper and introduces some future directions.

2 RELATED WORK

According to Oprescu and Kielmann [10], applications composed of independent tasks lend themselves well to elastic environments, where computational resources can be added or removed according to the application’s needs. Because of that, many works propose scheduling BoT applications on homogeneous and heterogeneous cloud environments [15]. For instance, Thai et al. [14] evaluate the scheduling of applications in on-demand VMs distributed across different data centers, focusing on the trade-offs between performance and cost, while Yao et al. [18] provide an approach that satisfies job deadlines and minimizes monetary cost. The proposed heuristics use both on-demand and reserved VMs. In [15], Thai et al. present an extensive survey and taxonomy of existing research in the scheduling of bag-of-task applications in clouds.

Gutierrez-Garcia and Sim [8] present an agent-based strategy that uses different scheduling heuristics to schedule concurrent BoTs applications to virtual machines in the cloud. In [10], the authors propose BaTs, a budget-constrained scheduler that uses on-demand VMs to execute BoT applications. Farahadaby et al. [6] propose FPRAS, a scheduler algorithm for BoT applications in multi-cloud environments. In the same line, in [16] a binary integer program to select computational resources in different cloud providers is presented. This work uses a static approach to scheduling non-finite applications whose main objective is to maximize the computational capacity of the infrastructure, respecting a budget constraint. Unlike our approach, those works do not consider the use of VMs spots to minimize the monetary cost of the execution and do not employ techniques to guarantee the continuity of the applications in case of VMs interruptions.

Checkpointing is widely used to tolerate failures in different environments. Checkpointing [3, 7] approaches periodically save the execution state of the application as an image file. These mechanisms can be costly to the cloud since data centers have limited network resources and may readily become overloaded when a huge number of checkpoint image files need to be stored. Moreover, they are usually not suitable for real-time applications or jobs with deadline constraints since periodically checkpoints and resuming of failed tasks are time-consuming.

Replication and resubmission of tasks are other mechanisms widely used to tolerate failures [11, 17, 19]. Using replication, several copies of the same task are executed to support fault tolerance. Most of the studies use a single primary-backup scheme considering one primary and one or several backup (copy) tasks scheduled in different computing instances [17, 19]. A backup is executed when its primary cannot complete execution due to failure but it does not require fault diagnosis. Zheng et al. [19] propose an algorithm to find an optimal backup schedule for each independent task. Wang et al. [17] extend Zheng’s results in a cloud context and using an elastic resource provisioning. Despite the use of overlapping techniques, primary-backup schemes require that tasks have enough time for executing backups in case of failure. [5] presents a comprehensive study of replication schedulers where all replicas of a task start executing concurrently and the next task is started as soon as one of the previous task replicas finishes. Benoit et al. [4] adopt a more conservative approach where the next task can only start when all the replicas of the previous task finished. Contrarily to our approach, the above solutions need to schedule both the primary and backup tasks and the later takes the execution control if the former fails. Neither of them uses backup tasks to avoid temporal failure.

Different mechanisms and tools have been proposed in the literature to deal with the revocation of spot VMs. The most common objective is to maintain a tradeoff between the monetary cost and the reliability of the execution. SpotOn [13] is a batch service computing that uses checkpointing, migration and replication to mitigate the impact of spot VMs revocations. Lu et al. use hybrid instances, including both on-demand instances for high priority tasks and backup, and spot instances for normal computational tasks.

In [9] a machine learning algorithm to select spot and on-demand VMs to execute batch jobs that arrive over time is proposed. The solution also switches to on-demand resources when there is no spot instance available to ensure the desired performance. SpotCheck [12] uses nested VMs within spot VMs to provide the illusion of a platform that offers always-available VMs. The nested VMs are migrated to an on-demand VM when a spot revocation is detected. Those works use the price variation of the spot market to predict the spot VMs’ revocations. However, with the changes in the spot market announced in December 2017, now all prices are stable, making it no longer possible to predict the termination of the VMs.
To the best of our knowledge, no work that considered the use of spot VMs to execute BoT applications has explored the hibernation-prone spot VMs to minimize the monetary cost of the execution. Besides that, there are no evaluations of the impact of the hibernation in the execution of the applications.

3 PROBLEM DEFINITION

Let $M$ be the user-provided set of VMs that can be used to execute the BoT application, where $M^s \subset M$ is the set of spot instances and $M^o \subset M$ is the set of on-demand ones. Moreover, let $D$ be the deadline defined by the user and $T = \{1, \ldots, D\}$ be a discrete set that represents the periods used during the execution. Each $vm_j \in M$ has a memory capacity of $m_j$ gigabytes, and a set of virtual cores $VC_j$. Whenever a new VM is allocated it takes $\alpha$ periods to be available, which includes the time to answer the client request and the boot time overhead.

Each VM $vm_j \in M$ is offered in only one of the markets (spot or on-demand) with cost $c_j$. When a VM is allocated to a user, he/she pays for a full-time interval called Allocation Cycle (AC), which is usually one hour. Thus, if a VM was used for 61 minutes, the user will be charged for two ACs (120 minutes). Note that an AC can correspond to several periods in $T$. For example, if each period corresponds to one second, an AC of one hour would correspond to 3600 periods.

Let $B$ be the set of tasks of a BoT application. We assume that each task $t_i \in B$ is executed in only one core and requires a known amount of memory $rm_i$, which must be available throughout its execution. Therefore, a multi-core VM can execute more than one task simultaneously (one task per core) only when there is enough main memory to allocate them. We also consider that the execution time of each task $t_i$ in any $vm_j \in M$ is also known and given by $\epsilon_{ij}$. Figure 1a presents the execution of a set of tasks in a spot VM. As can be seen, after $\alpha$ periods each core starts executing a task. However, note that there is a gap between tasks 4 and 5. That happened because the VM does not have enough memory to fulfill the requirement of tasks 1 and 5 at the same time; thus $core_0$ remains idle until task 1 finishes.

In an environment where VMs are hibernation-prone, a hibernated VM can resume in time to meet the deadline or not. In the first situation, shown in Figure 1b, since the VM resumes in time, tasks go on running from the break-point. On the other hand, if the VM does not resume in time, it is necessary to migrate the affected tasks to different VMs to avoid a temporal failure. If the others already deployed spot VMs were not able to execute the affected tasks in time, new on-demand VMs are deployed.

Figure 1c illustrates a hibernation followed by migration. Considering that the spot VM hibernated in time $p \in T$, $Q_{jp} \subset B$ is the set of tasks to be migrated if $vm_j$ does not resume in time, so it contains both the tasks that were running in $vm_j$ at the moment of hibernation and the tasks that were waiting to be executed in that virtual machine. In Figure 1c, $Q_{jp} = \{1, 2, 4, 5\}$.

We define the set of VMs that will receive the affected tasks by hibernation as $K \subset M$. We also consider that: (i) a VM takes $\alpha$ periods of time to be ready to receive the migrated tasks and (ii) the number of periods required to perform all tasks of $Q_{jp}$ in VMs of $K$ is $rt(Q_{jp}, K)$. Equation 1 calculates the hibernation time limit ($st \in T$), when the migration must be triggered. If the migration does not start at time $st$, it will not be possible to meet the deadline. Thus, we aim at finding a set of instances $K \subset M$, such that $st + rt(Q_{jp}, K) \leq (D - \alpha)$.

$$st = (D - \alpha) - rt(Q_{jp}, K)$$

3.1 Initial Task Allocation Map

(a) Execution without hibernation

(b) Execution with hibernation without migration

(c) Execution with hibernation followed by task migration

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

4 HIBERNATION-AWARE DYNAMIC SCHEDULER

The Hibernation-Aware Dynamic Scheduler (HADS) aims to respect the application deadline even in the presence of hibernation, while minimizing the monetary cost. HADS contains the following main procedures: i) a Primary Scheduling Heuristic that defines an initial task allocation map (Section 4.2), and ii) a Dynamic Scheduler Module that monitors the VMs and eventually moves tasks
among them so that the deadline is respected and the monetary costs minimized (Section 4.3).

4.1 Computing $D_{spot}$

To schedule the primary tasks, it is necessary to estimate $D_{spot}$, that is the primary tasks execution times in the worst case. That parameter defines the maximum limit for the primary tasks makespan, and it is used to ensure that in case of a hibernation there will be enough remaining time to migrate the applications to other VMs and finish it before the deadline $D$.

$D_{spot}$ is estimated using Algorithm 1. In addition to the deadline $D$ and sets $B$ and $M$, the Algorithm also receives $\text{max}_{\text{spot}}$ as input, that is the maximum number of spot VMs that can be allocated simultaneously. This value is defined by the cloud provider. In Amazon EC2, for example, by default, only 20 VMs spots can be allocated simultaneously in the same region.

We can assume that, using Algorithm 2, the number of tasks $n$ allocated to a VM can be estimated as presented in line 1. After that, a set $W \subseteq B$ containing the $n$ longest tasks is created (line 2). Since $W$ contains the $n$ longest tasks, we can consider that the execution of tasks $t_i \in W$ in the slowest VM of $M$ represents the worst makespan case. In line 7, that makespan, called $mkp_w$, is estimated using the procedure $\text{get_makespan}$ by considering the slowest VM selected in line 3 (procedure $\text{get_slowest_vm}$). Thus, we calculate the value of $D_{spot}$ as the difference between $D$ and $mkp_w$ (line 8). If $D_{spot}$ is less than zero, the user is warned to consider another deadline.

Algorithm 1 Compute $D_{spot}$

**Input:** $B, M, \text{max}_{\text{spot}}, D$

1. $n \leftarrow \lceil \frac{n}{\text{max}_{\text{spot}}} \rceil$
2. $W \leftarrow \text{get_longest_tasks}(n, B)$
3. $\text{vmf} \leftarrow \text{get_slowest_vm}(M)$
4. for all $t_i \in W$ do
5. $\text{allocate}(t_i, \text{vmf})$
6. end for
7. $mkp_w = \text{get_makespan}(\text{vmf})$
8. $D_{spot} = D - mkp_w$

4.2 Primary Scheduling Heuristic

Algorithm 2 shows the primary scheduling heuristic which is a greedy algorithm that allocates the set of tasks $t_i \in B$ to a set of VMs spot. Tables 1 and 2 present the used variables and functions respectively. The algorithm receives $B, M'$ and $D_{spot}$ as input parameters.

Initially, tasks are ordered in descending order by the memory size they require (line 1). Then, the algorithm tries to allocate each task in an already allocated virtual machine from set $A$ (lines 5 to 11), since it has enough memory and also ensures that the task insertion will respect $D_{spot}$.

Allocating tasks in an already allocated VM reduces boot time overhead in comparison of allocating a new VM. However, if such an allocation is not possible, the algorithm must allocate a new VM. In this case, the heuristic defines the best type of VM in terms of execution time (line 13). Finally, it updates the primary scheduling map (line 15).

4.3 Dynamic Scheduler Module

The Dynamic Scheduler Module of HADS is an event-driven algorithm that performs some actions in response to some events that may occur along the application execution in order to reduce the monetary cost and meet the application deadline. These events and the corresponding executed procedures are shown in Figure 2.

We consider that a VM is in one of the following states i) busy, if active and executing tasks; ii) idle, if active but has finished all tasks allocated to it so far; iii) hibernated, if was hibernated by the cloud provider or was not available in the beginning of the application execution; and iv) terminated, if was terminated by the Dynamic Scheduler Module.

The events and associated procedures are described in next subsections.

---

Table 1: Variables of Primary Task Scheduling Heuristic

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>Set of tasks</td>
</tr>
<tr>
<td>$M'$</td>
<td>Set of VMs spots</td>
</tr>
<tr>
<td>$D_{spot}$</td>
<td>Parameter that determines the maximum occupation period of a spot VM</td>
</tr>
<tr>
<td>$A$</td>
<td>Set of VMs selected to execute primary tasks</td>
</tr>
<tr>
<td>$t_i$</td>
<td>Task $t_i$</td>
</tr>
<tr>
<td>$\text{allocated}$</td>
<td>Boolean variable that indicates whether task $t_i$ was successfully scheduled</td>
</tr>
<tr>
<td>$\text{vmj}, \text{vmk}$</td>
<td>Virtual machines</td>
</tr>
</tbody>
</table>

Table 2: Functions and Procedures of Primary Task Scheduling Heuristic

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sort}(B)$</td>
<td>Sorts the set of tasks $B$ in descending order by memory size demand</td>
</tr>
<tr>
<td>$\text{check_allocation}(t_i, \text{vmj}, D_{spot})$</td>
<td>Checks if the allocation of task $t_i$ to a core of $\text{vmj}$ respects the $D_{spot}$ limit and if $\text{vmj}$ has available memory to meet the requirement of the task ($r_{mi}$)</td>
</tr>
<tr>
<td>$\text{allocate}(t_i, \text{vmj})$</td>
<td>Allocates task $t_i$ to a core of $\text{vmj}$</td>
</tr>
<tr>
<td>$\text{best_VM}(t_i, M')$</td>
<td>Selects the spot VM that executes task $t_i$ with the minimum number of periods of time</td>
</tr>
<tr>
<td>$\text{create_primary_map}(A)$</td>
<td>Creates the initial execution plan</td>
</tr>
</tbody>
</table>
Update VM state to Hibernate.

were running as the ones that were waiting for execution when the VM hibernated.

Algorithm 2 Primary Task Scheduling

Input: \( B, M^i, D_{spot} \)

1. \( \text{sort}(B) \) \{Sort tasks by \( r_m_k \)\}
2. \( A \leftarrow \emptyset \) \{set of allocated VMs\}
3. for all \( t_i \in B \) do
   4. \( \text{allocated} \leftarrow \text{False} \)
   \{Check if \( v_m_j \) has sufficient time and memory to execute \( t_i \) without violating the limit \( D_{spot} \)\}
5. for all \( v_m_j \in A \) do
6. if check_allocation\((t_i, v_m_j, D_{spot})\) then
   7. allocate\((t_i, v_m_j)\)
   8. allocated \leftarrow True
   9. stop_loop()
10. end if
11. end for
12. if not allocated then
13. \( v_m_k \leftarrow \text{best VM}(t_i, M^i) \)
14. allocate\((t_i, v_m_k)\)
15. \( A \leftarrow A \cup \{v_m_k\} \) \{Update the set of allocated VMs\}
16. \( M^i \leftarrow M^i \setminus \{v_m_j\} \)
17. end if
18. end for
19. create_primary_map\((A)\)

Figure 2: Events and the corresponding procedures of the Dynamic Scheduler Module

4.3.1 Being idle. When a busy VM finishes all tasks allocated to it, its state changes to idle. As we will see in Section 4.4, idle VMs can return to busy states due to the task migration procedure. However, if a VM stays idle for a long time, to avoid additional charges, it is terminated near to the end of the current allocation cycle.

4.3.2 Hibernating. The cloud provider can hibernate busy or idle spot VMs. In this case, the algorithm changes the VM state to hibernated. If the VM was busy, the algorithm also computes the hibernation time limit (Equation 1) considering both the tasks that were running as the ones that were waiting for execution when the VM hibernated.

4.3.3 Reaching hibernation time limit. As seen in Section 3, when a hibernated VM that was busy reaches the hibernation time limit, in order to meet the deadline all affected tasks are migrated to other VMs. Thus, when this event happens the algorithm starts the migration procedure (Section 4.4).

4.3.4 Resuming. A VM may resume before the hibernation time limit or not. As seen in Section 3, if a VM resumes before the time limit, the tasks continue their executions from the break-points and no additional action is taken. On the other hand, when a VM resumes after the time limit, before it happens, the event Reaching hibernation time limit has already happened and consequently the tasks allocated to that VM were already migrated to other VMs. Then, in this case, the algorithm executes a work-stealing procedure that tries to bring back tasks from on-demand VMs to the resumed spot. That procedure is described in Section 4.5.

4.4 Migration Procedure

The migration procedure is presented in Algorithm 3. As can be seen there, the algorithm receives as input the set \( Q_{kp} \subset B \), that contains the remaining tasks of a spot \( v_m_k \in M \) that hibernated at period \( p \in T \). Besides that, the procedure also receives the deadline \( D \) and the computational resources for where those tasks can be migrated: \( IR \subset M \) that contains all idle VMs; \( BR \subset M \) that includes all busy VMs; and \( M^o \subset M \) that contains on-demand VMs that can be deployed. Tables 3 and 4 show the used variables and functions, respectively.

Table 3: Variables of Migration Procedure

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q_{kp} )</td>
<td>Set of tasks to be migrated</td>
</tr>
<tr>
<td>( D )</td>
<td>Deadline defined by the user</td>
</tr>
<tr>
<td>( IR )</td>
<td>Set of idle VMs</td>
</tr>
<tr>
<td>( BR )</td>
<td>Set of busy VMs</td>
</tr>
<tr>
<td>( M^o )</td>
<td>Set of on-demand VMs that can be allocated</td>
</tr>
<tr>
<td>( t_i )</td>
<td>A task from set ( Q_{kp} )</td>
</tr>
<tr>
<td>( v_m_j )</td>
<td>A VM from set ( IR, BR ) or ( M^o )</td>
</tr>
<tr>
<td>( start_{ij} )</td>
<td>The period that a task ( t_i ) will start if it is allocated to a ( v_m_j )</td>
</tr>
<tr>
<td>( e_{ij} )</td>
<td>Execution time of a task ( t_i ) in a ( v_m_j )</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>Number of periods a VM takes to become available for execution</td>
</tr>
<tr>
<td>migrated</td>
<td>Boolean variable that indicates whether task ( t_i ) was successfully migrated</td>
</tr>
</tbody>
</table>

Table 4: Functions and Procedures used in the Migration Procedure

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort_by_market()</td>
<td>Sorts the sets ( IR ) or ( BR ) prioritizing spot VMs over the on-demand ones</td>
</tr>
<tr>
<td>enough_mem((t_i, v_m_j))</td>
<td>Returns TRUE, if there is enough memory to allocate ( t_i ) to ( v_m_j ), otherwise returns FALSE</td>
</tr>
<tr>
<td>migrate((t_i, v_m_j))</td>
<td>Migrates task ( t_i ) to one of the virtual cores of ( v_m_j )</td>
</tr>
<tr>
<td>sort_by_price((M^o))</td>
<td>Sorts the set of on-demand VMs ( M^o ) in ascending order by the VM’s price</td>
</tr>
<tr>
<td>start_vm((v_m_j))</td>
<td>Deploys a new on-demand ( v_m_j ) in the cloud</td>
</tr>
</tbody>
</table>
Algorithm 3 Migration Procedure

Input: $Q_{kp}$, $D$, $IR$, $BR$, $M^o$

1. for each $t_i \in Q_{kp}$ do
2.  migrated $\leftarrow$ False
3.  ["Attempt 1"]
4.  sort_by_market(IR) ["Prioritizes spot VMs "]
5.  for each $vm_j \in IR$ do
6.    if $start_{ij} + e_{ij} < D$ and enough_mem($t_i$, $vm_j$) then
7.      migrate($t_i$, $vm_j$)
8.      $IR \leftarrow IR - \{vm_j\}$
9.      $BR \leftarrow BR \cup \{vm_j\}$
10.     migrated $\leftarrow$ True
11.     stop_loop()
12.   end if
13. end for
14. if migrated is False then
15.  ["Attempt 2"]
16.  sort_by_market(BR) ["Prioritizes spot VMs "]
17.  for each $vm_j \in BR$ do
18.    if $start_{ij} + e_{ij} < D$ and enough_mem($t_i$, $vm_j$) then
19.      migrate($t_i$, $vm_j$)
20.     migrated $\leftarrow$ True
21.     stop_loop()
22.   end if
23. end for
24. end if
25. if migrated is False then
26.  ["Attempt 3"]
27.  sort_by_price($M^o$)
28.  for each $vm_j \in M^o$ do
29.    if $start_{ij} + e_{ij} < (D - \alpha)$ and enough_mem($t_i$, $vm_j$) then
30.      start_vm($vm_j$)
31.      migrate($t_i$, $vm_j$)
32.      $M^o \leftarrow M^o - \{vm_j\}$
33.      $BR \leftarrow BR \cup \{vm_j\}$
34.      stop_loop()
35.  end if
36. end for
37. end if
38. end for

For each task $t_i$ in $Q_{kp}$, the algorithm at first tries to allocate $t_i$ to one of the idle VMs of set $IR$ (lines 5 to 13). If it is not possible, it tries busy VMs of set $BR$ (lines 17 to 23). In the last case, the task is allocated to a new on-demand VM of set $M^o$ (lines 28 to 36). That approach tries to minimize the monetary costs, since the client pays for ACs and the current AC of idle and busy VMs may have already been charged by the provider.

Note that the algorithm always gives priority to spot VMs, both in the case of idle VMs (line 4) and busy VMs (line 16) as well, to reduce monetary costs.

To migrate a task $t_i$ to a virtual machine $vm_j$, the algorithm checks if the start period of $t_i$ in $vm_j$ ($start_{ij}$) plus the corresponding execution time ($e_{ij}$) is smaller than the deadline. Since idle and busy VMs are already active, the overhead $\alpha$ is not taken into account in these cases (lines 6 and 18). But when a new VM has to be deployed (line 30), the overhead $\alpha$ has to be considered not to violate the deadline (line 29).

When a valid VM is found, the procedure updates the VM state. That is shown in lines 7 and 11, where the selected idle VM is removed from set $IR$ and included in the set $BR$, and from line 31 to 34, where the on-demand VM is removed from set $M^o$.

4.5 Work-Stealing Procedure

The work-stealing procedure, presented in Algorithm 4, is used to reduce the number of contracted ACs of on-demand VMs, and is executed when a spot VM resumes after the hibernation time limit is reached. When this occurs, the algorithm tries to migrate tasks from on-demand VMs to the resumed VM. All used variables and functions are showed in Tables 5 and 6, respectively.

Table 5: Variables of Work-Stealing Procedure

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$BR$</td>
<td>Set of busy VMs</td>
</tr>
<tr>
<td>$IR$</td>
<td>Set of idle VMs</td>
</tr>
<tr>
<td>$vm_k$</td>
<td>A resumed spot VM</td>
</tr>
<tr>
<td>$D$</td>
<td>Deadline defined by the user</td>
</tr>
<tr>
<td>$vm_j$</td>
<td>A busy VM from set $BR$</td>
</tr>
<tr>
<td>$W$</td>
<td>A set of tasks that can be stolen from $vm_j$</td>
</tr>
<tr>
<td>$t_i$</td>
<td>A task from set $W$</td>
</tr>
<tr>
<td>$start_{ik}$</td>
<td>The period that a task $t_i$ will start if it is allocated to the spot $vm_k$</td>
</tr>
<tr>
<td>$e_{ik}$</td>
<td>Execution time of a task $t_i$ in the spot $vm_k$</td>
</tr>
</tbody>
</table>

Table 6: Functions and Procedures used by the Work-Stealing Procedure

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$selectTasks(vm_j)$</td>
<td>Returns all tasks that can be stolen from an on-demand $vm_j$</td>
</tr>
<tr>
<td>$migrate(t_i, vm_k)$</td>
<td>Migrates task $t_i$ to one of the virtual cores of the spot $vm_k$</td>
</tr>
</tbody>
</table>

The Algorithm receives as input the sets of busy and idle VMs ($BR \subset M$ and $IR \subset M$, respectively), the resumed spot $vm_k \in M$, and deadline $D$. For each on-demand $vm_j$ in $BR$, the procedure selects the tasks that can be stolen from $vm_j$ (line 3) and tries to migrate them to $vm_k$ (lines 4-8).

The task selection considers only tasks that are not running and that will not start in the current AC. Figure 3 shows an example of that selection. In this example, tasks are spread over three ACs ($AC_1$, $AC_2$, and $AC_3$). Since the current Allocation Cycle is $AC_1$, tasks 5, 6, 7 and 8 are candidates to be stolen and start in the next cycles.

As in the migration procedure, for each selected task the work-stealing procedure also verifies if the migration would result in the deadline violation, as can be seen in line 5, where the algorithm checks if the start time of the task in the spot $vm_k$ ($start_{ik}$) plus its execution time $e_{ik}$ is smaller than the deadline. If the deadline is not violated, the task is migrated (line 6). Finally, if any task is migrated to the spot $vm_k$, its state is changed to busy, see line 12,
Figure 3: Example of tasks in an on-demand VM that can be stolen by the work-stealing procedure

and it is included in the set of busy VMs. Otherwise, it is updated with idle state in line 14.

Algorithm 4 Work-Stealing Procedure

Input: BR, IR, vmk, D
1. for each vmk ∈ BR do
2. if vmk is on-demand then
3. \( W \leftarrow \text{selectTasks}(vmk) \) /* get tasks that can be stolen */
4. for each \( t_i \in W \) do
5. if \( \text{start}_{ik} + e_{ik} < D \) and enough_mem(\( t_i, vmk \)) then
6. \( \text{migrate}(t_i, vmk) \)
7. end if
8. end for
9. end if
10. end for
11. if any task was stolen then
12. \( BR \leftarrow BR \cup \{ vmk \} \)
13. else
14. \( IR \leftarrow IR \cup \{ vmk \} \)
15. end if

5 EXPERIMENTAL RESULTS

This section presents execution times and monetary costs of different BoT applications executed in the Amazon EC2 virtual machines.

According to the information of Amazon Web Server (AWS), 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, for the purposes of this work, we chose to use the third and fourth generation VMs of families C3 and C4. Those VMs are optimized for computation and have high availability in the spot market. Table 7 shows the computational characteristics of the used VMs with their prices in on-demand and spot markets in April 2019.

The jobs used in our tests are composed of synthetic tasks generated with the application template proposed by Alves et al. [2]. That application template is based on vector operations, and the execution time depends on the size of the used vectors. Thus, we created several synthetic tasks, each one with memory footprint between 2.81 MB and 13.19 MB, which resulted in execution times varying from 1:42 to 5:30 minutes, as can be seen in Table 8. After that, we created three BoT applications by randomly selecting those tasks.

Table 7: VMs attributes

<table>
<thead>
<tr>
<th>Type</th>
<th>#VCPUs</th>
<th>Memory</th>
<th>On-demand price per AC</th>
<th>Spot price per AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>c3.large</td>
<td>2</td>
<td>3.75 GB</td>
<td>0.105$</td>
<td>0.0294$</td>
</tr>
<tr>
<td>c4.large</td>
<td>2</td>
<td>3.75 GB</td>
<td>0.100$</td>
<td>0.0308$</td>
</tr>
<tr>
<td>c3.xlarge</td>
<td>4</td>
<td>7.50 GB</td>
<td>0.210$</td>
<td>0.0588$</td>
</tr>
<tr>
<td>c4.xlarge</td>
<td>4</td>
<td>7.50 GB</td>
<td>0.199$</td>
<td>0.0617$</td>
</tr>
</tbody>
</table>

Since the provider determines the hibernation and resuming times of the VMs based on internal factors of the cloud such as demand variation, the client cannot control that process. Thus, to evaluate different patterns of VMs hibernation and resuming, we developed a Hibernation Emulation Module (HEM) that emulates the hibernation process, using Poisson distribution [1] to model the hibernation and resuming times.

In our tests, when the hibernation (emulated) occurs, the VM state is saved by using the checkpoint tool CRIU, and all tasks allocated to it are paused. In this way, if later the VM resumes, those tasks can be recovered and continue their execution from the checkpoint. Note that HEM uses distinct Poisson functions for the hibernation and resuming modeling, which allows the creation of scenarios where the hibernating and resuming events have different probability mass functions defined by the parameters \( \lambda_h \) and \( \lambda_r \), respectively.

To verify the impact of hibernating and resuming events on the monetary cost and execution times, in different scenarios, we considered allocation cycles of 15 minutes (\( AC = 900s \)) and a fixed deadline of 35 minutes (\( D = 2100s \)) for all tests. We also considered the VM allocation overhead of 3 minutes (\( r = 180s \), based on empirical tests) and the sets \( M^s \) and \( M^r \) were built considering the allocation limits specified by Amazon EC2. So, up to five VMs of each type in each market could be allocated.

Let the \( \lambda \) parameter of Poisson distribution be the number of expected events divided by a time interval. Since in our experiments we considered the deadline \( D \) as the time interval and variables \( k_h \) and \( k_r \), as the expected number of hibernating and resuming events, respectively, \( \lambda_h \) and \( \lambda_r \) parameters are given by \( \lambda_h = k_h / D \) and \( \lambda_r = k_r / D \).

In the initial tests, we consider that a VM may hibernate only once. So, in practice the values for \( k_h \) and \( k_r \) determine the chance of a VM hibernation and resuming, respectively. Table 9 presents five created scenarios, considering the minimal values for hibernating and resuming events, as \( k_h = 1 \) and \( k_r = 0 \), respectively. So, we consider that at least one hibernation can occur during an execution

---

Figure 4 shows the average duration of a hibernation in each scenario. As expected, scenarios c1 and c2 present the longest hibernation times (18:47 and 23:50 minutes, respectively), since in these scenarios the resuming event does not occur ($k_r = 0$). The smallest times are seen in c3 and c4 (10:27 and 11:18 minutes, respectively) that present the highest value of resuming ($k_h = 5$), which reduces the average duration of hibernation. On the other hand, in scenario c5 where $k_r = 2.5$, the hibernation time is longer than in c3 and c4 (13:25 minutes).

Figure 4: Average duration of hibernation in different scenarios

The results obtained through the execution of jobs in different scenarios presented in Table 9 are discussed in Section 5.1, where the costs were compared with two baseline cases: i) 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: Different execution scenarios generated by varying parameters $\lambda_h$ and $\lambda_r$

<table>
<thead>
<tr>
<th>ID</th>
<th>$k_h$</th>
<th>$k_r$</th>
<th>$\lambda_h$</th>
<th>$\lambda_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>1</td>
<td>0</td>
<td>1/2100</td>
<td>0/2100</td>
</tr>
<tr>
<td>c2</td>
<td>5</td>
<td>0</td>
<td>5/2100</td>
<td>0/2100</td>
</tr>
<tr>
<td>c3</td>
<td>1</td>
<td>5</td>
<td>1/2100</td>
<td>5/2100</td>
</tr>
<tr>
<td>c4</td>
<td>5</td>
<td>5</td>
<td>5/2100</td>
<td>5/2100</td>
</tr>
<tr>
<td>c5</td>
<td>3</td>
<td>2.5</td>
<td>3/2100</td>
<td>2.5/2100</td>
</tr>
</tbody>
</table>

As can be seen in Table 11, when compared to the on-demand only approach, the proposed solution presented cost reductions in almost all cases, except only for job J552 in scenario c2 where there was an increase of 11.25% in the cost. In the other cases, the cost reduction varied between 13.33% and 70.55%.

Table 11: Average results of three executions of jobs J551, J552 and J553 in different scenarios

<table>
<thead>
<tr>
<th>scenario</th>
<th>$\lambda_h$</th>
<th>$\lambda_r$</th>
<th>makespan (min)</th>
<th>cost ($)</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>2.00</td>
<td>1.50</td>
<td>32:16</td>
<td>$0.531$</td>
<td>61.89%</td>
</tr>
<tr>
<td>c2</td>
<td>2.67</td>
<td>4.00</td>
<td>34:62</td>
<td>$0.801$</td>
<td>19.49%</td>
</tr>
<tr>
<td>c3</td>
<td>1.33</td>
<td>0.67</td>
<td>28:36</td>
<td>$0.360$</td>
<td>63.85%</td>
</tr>
<tr>
<td>c4</td>
<td>2.67</td>
<td>1.33</td>
<td>30:29</td>
<td>$0.538$</td>
<td>45.95%</td>
</tr>
<tr>
<td>c5</td>
<td>2.33</td>
<td>2.00</td>
<td>31:30</td>
<td>$0.560$</td>
<td>43.75%</td>
</tr>
</tbody>
</table>

5.1 Evaluation of the proposed solution using different test cases.

Table 10: Baseline executions

<table>
<thead>
<tr>
<th>job</th>
<th># VMs</th>
<th>makespan without hibernation</th>
<th>on-demand only</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>J551</td>
<td>3</td>
<td>24:30</td>
<td>0.29$</td>
<td>0.995$</td>
</tr>
<tr>
<td>J552</td>
<td>4</td>
<td>24:50</td>
<td>0.41$</td>
<td>1.395$</td>
</tr>
<tr>
<td>J553</td>
<td>4</td>
<td>26:02</td>
<td>0.47$</td>
<td>1.592$</td>
</tr>
</tbody>
</table>

Table 10 presents the average costs of executing the three evaluated jobs J551, J552 and J553 for the baseline cases. It also presents the number of used VMs, the average makespans 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 70% for all executions, which is basically the same difference in the price between the used spots and on-demand VMs (see Table 7). Note also that, in Table 10 the jobs makespans are around 25 minutes instead of the expected 35 minutes of the deadline. This occurs because Algorithm 2 respects the $D_{spot}$ limit presented in Section 4.1.

Figure 4 shows the average duration of hibernation in each scenario. As expected, scenarios c1 and c2 present the longest hibernation times (18:47 and 23:50 minutes, respectively), since in these scenarios the resuming event does not occur ($k_r = 0$). The smallest times are seen in c3 and c4 (10:27 and 11:18 minutes, respectively) that present the highest value of resuming ($k_h = 5$), which reduces the average duration of hibernation. On the other hand, in scenario c5 where $k_r = 2.5$, the hibernation time is longer than in c3 and c4 (13:25 minutes).

The results obtained through the execution of jobs in different scenarios presented in Table 9 are discussed in Section 5.1, where the costs were compared with two baseline cases: i) 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 10 presents the average costs of executing the three evaluated jobs (J551, J552, and J553) for the baseline cases. It also presents the number of used VMs, the average makespans 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 70% for all executions, which is basically the same difference in the price between the used spots and on-demand VMs (see Table 7). Note also that, in Table 10 the jobs makespans are around 25 minutes instead of the expected 35 minutes of the deadline. This occurs because Algorithm 2 respects the $D_{spot}$ limit presented in Section 4.1.
of spot VMs in this scenario, the work-stealing procedure never executes. Moreover, some tasks initially allocated to the spots had to migrate since the beginning of the execution to on-demand VMs. Particularly, in J552 execution, the spot VMs were hibernated successively, resulting in an allocation of up to eight on-demand VMs and an average increase of 11.25% in monetary costs. Figure 5 shows that in c2 the migration to on-demand VMs occurs in more than 75% of the hibernation cases.

However, note that although there is also no possibility of resuming in scenario c1, the cost reduced in more than 50% for all evaluated jobs. That happens because in this scenario the hibernation rate is low \( (k_h = 1) \), in general, less than half of the spot VMs hibernates. Thus, the tasks are migrated to busy or idle VMs instead of being allocated to new on-demand VMs. That can be seen in Figure 5, where migrations to idle and busy VMs happen in more than 50% of the hibernation cases.

On the other hand, scenario c3 presents the best results in terms of cost reduction (more than 60% for all jobs). This scenario represents the best execution case because it has the lowest hibernation rate \( (k_h = 1) \) and the highest resuming rate \( (k_r = 5) \). Figure 5 shows that a maximum gain of 13.33% in c2 (job J553). Thus, although the cost gain in c4 was more than 30% for all evaluated Jobs, against a maximum gain of 13.33% in c2 (job J553). Thus, although the hibernation rate is the same in both scenarios, we see in Figure 5 that migrations to on-demand VMs occurred in less than 50% in c4, but more than 75% in c2. Also, as the resuming rate is high in c4, we see recoveries, work-stealing, idle and busy migration in all executions.

In c5, which represents the average case \( (k_h = 3 \text{ and } k_r = 2.5) \), HADS migrates task to on-demand VMs in all evaluated jobs, and the cost reduction was between 30% and 55%. With this, we see that the algorithm’s efficiency depends on a balance between the hibernation and resuming rates. Thus, to better understand the impact of these rates, in Section 5.2 we performed several tests varying the frequency of hibernation considering different resuming rates.

### 5.2 Impact of hibernation and resuming on the execution costs of the jobs
To observe the cost evolution concerning the hibernation rates, we submit job J553, which contains the largest number of tasks, to several scenarios increasing the hibernation rate of each execution progressively. Three cases of tests were considered in this evaluation: i) execution without resuming \( (k_r = 0) \); ii) execution with medium chance of resuming \( (k_r = 3) \); and iii) execution with high chance of resuming \( (k_r = 7) \).

As can be seen in Figure 6, the cost variation appears to have an intrinsic relationship with the resuming rate. The monetary costs approach the on-demand only cost when \( k_r = 0 \), and distances from it as the rate of resuming event increases.
wait for the resuming event. On the other hand, in the case where
the effectiveness of our scheduling and that it tolerates temporal
failures. The proposed strategy was evaluated
regardless of the resuming rate.

However, at the point next to it, the hibernation rate of $k_h = 7$
causes VMs to always hibernate in the first few seconds of the
execution. Because of this, in the scenarios where $k_r > 0$, the
execution costs are slightly affected, since there is sufficient time to
wait for the resuming event. On the other hand, in the case where
$k_r = 0$, the execution cost is higher than the "on-demand only"
approach since the user pays for the first allocation cycle of the
spot VMs and for the on-demand VMs used in the migration.

6 CONCLUDING REMARKS AND FUTURE WORK

This paper proposed the Hibernation-Aware Dynamic Scheduler,
a dynamic scheduler for bag-of-task applications with deadline con-
straints that uses both hibernation-prone spot VMs (for cost sake)
and on-demand VMs. Our scheduling aims at minimizing mone-
tary costs of bag-of-tasks, respecting application’s deadline and
avoiding temporal failures. The proposed strategy was evaluated
using the VMs of AWS EC2 with real executions of synthetic appli-
cations with five scenarios of hibernation. Our results confirmed
the effectiveness of our scheduling and that it tolerates temporal
failures.

As future work, we intend to test our approach with several real
BoT applications and work in a version of the proposed scheduling
with checkpoints of the tasks, so that, in the migration case, the
tasks can start their executions from the last checkpoints, instead
of being re-started from the beginning. Besides that, we also intend
to include load balance techniques to increase the occupancy of
idle VMs.

ACKNOWLEDGMENTS

This research was supported by Programa Institucional de Interna-
cionalização (PrInt) from CAPES (process number 88887.310261/2018-
00).

REFERENCES

variate and quantitative model for predicting cross-application interference
https://doi.org/10.1016/j.jss.2017.04.001
Robert. 2013. Energy-aware checkpointing of divisible tasks with soft or hard
deadlines. In International Green Computing Conference, IGCC 2013, Arlington,
precedence task graphs on heterogeneous platforms. In 22nd IEEE International Symposium
on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA,
[5] Walfredo Cirne, Francisco Vilar Brasiliero, Daniel Paradanos da Silva, Luis Fab-
clairvoyant assignment of bag-of-tasks applications across multiple clouds. In 2012 13th International Conference on Parallel and Distributed Computing, Ap-
lications and Technologies. IEEE, 423–428.
fault-tolerant infrastructure for virtualized service providers. In IEEE/IPIP Network
Operations and Management Symposium, NOMS 2010, 19-25 April 2010, Osaka,
Japan, 455–462.
agent-based elastic cloud bag-of-tasks concurrent scheduling. Future Generation
[9] Ishai Menache, Ohad Shamir, and Navendu Jain. 2014. 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,
Workflow Systems. In Grid and Services Evolution, Proceedings of the 3rd CoreGRID
2015. SpotCheck: designing a derivative IaaS cloud on the spot market. In
Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015,
J. Shenoy. 2015. SpotOn: a batch computing service for the spot market. In Proceed-
ings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast,
[14] Long Thai, Blesson Varghese, and Adam Barker. 2014. Executing Bag of Dis-
tributed Tasks on the Cloud: Investigating the Trade-Offs between Performance and Cost.
In IEEE 6th International Conference on Cloud Computing Technology and Science,
Future Generation Comp. Syst. 82 (2018), 1–11.
[16] Johan Tordsson, Rubén S Montero, Rafael Moreno-Vozmediano, and Ignacio M
Llorente. 2012. Cloud brokering mechanisms for optimized placement of virtual
machines across multiple providers. Future generation computer systems 29, 2
2015. FESTAL: Fault-Tolerant Elastic Scheduling Algorithm for Real-Time Tasks in
[18] Min Yao, Peng Zhang, Yin Li, Jie Hu, Chuan Li, and Xiang-Yang Li. 2014. Cutting
International Conference on Web Services, ICWS, 2014. Anchorage, AK, USA, June
of Fault-Tolerant Scheduling Strategies Using Primary-Backup Approach for Computational

Figure 6: Impact of variation of $k_h$ in the execution costs of job J553

When $k_h = 6$, all three lines approach to the same cost. At this point, all spot VMs hibernate over different periods of the execution. Some VMs hibernate in the last minutes of the execution, making necessary to allocate on-demand VMs to meet the deadline, regardless of the resuming rate.

As future work, we intend to test our approach with several real
BoT applications and work in a version of the proposed scheduling
with checkpoints of the tasks, so that, in the migration case, the
tasks can start their executions from the last checkpoints, instead
of being re-started from the beginning. Besides that, we also intend
to include load balance techniques to increase the occupancy of
idle VMs.

ACKNOWLEDGMENTS

This research was supported by Programa Institucional de Interna-
cionalização (PrInt) from CAPES (process number 88887.310261/2018-
00).