HAL
open science

A Bag-of-Tasks Scheduler Tolerant to Temporal Failures in Clouds

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

To cite this version:

Luan Teylo, Lúcia Maria de A. Drummond, Luciana Arantes, Pierre Sens. A Bag-of-Tasks Scheduler Tolerant to Temporal Failures in Clouds. SBAC-PAD 2019 - International Symposium on Computer Architecture and High Performance Computing, Oct 2019, Campo Grande, Brazil. hal-02284965

HAL Id: hal-02284965

https://inria.hal.science/hal-02284965v1

Submitted on 12 Sep 2019

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

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

A Bag-of-Tasks Scheduler Tolerant to Temporal Failures in Clouds

Luan Teylo and Lúcia Maria de A. Drummond
Fluminense Federal University Institute of Computing
Niteroi, Brazil
(luanteylo, lucia)@ic.uff.br

Luciana Arantes and Pierre Sens
Sorbonne Université, CNRS, INRIA
LIP6, Paris, France
(luciana.arantes, pierre.sens)@lip6.fr

***Abstract*—Cloud platforms offer different types of virtual machines which ensure different guarantees in terms of availability and volatility, provisioning the same resource through multiple pricing models. For instance, in Amazon EC2 cloud, the user pays per hour for *on-demand* instances while *spot* instances are unused resources available for a lower price. Despite the monetary advantages, a spot instance can be terminated or hibernated by EC2 at any moment. Using both hibernation-prone spot instances (for cost sake) and on-demand instances, we propose in this paper a static scheduling for applications which are composed of independent tasks (bag-of-task) with deadline constraints. However, if a spot instance hibernates and it does not resume within a time which guarantees the application's deadline, a temporal failure takes place. Our scheduling, thus, aims at minimizing monetary costs of bag-of-tasks applications in EC2 cloud, respecting its deadline and avoiding temporal failures. Performance results with task execution traces, configuration of Amazon EC2 virtual machines, and EC2 market history confirms the effectiveness of our scheduling and that it tolerates temporal failures.**

***Index Terms*—Clouds, Temporal failures, Scheduling**

I. Introduction

In the past few years, cloud computing has emerged as an attractive option to run different classes of applications due to several advantages that it brings when compared with a dedicated infrastructure. Clouds provide a significant reduction in operational costs, besides offering a rapid elastic provisioning of computing resources like virtual machines (VMs) and storage. However, in cloud environments, besides the usual goal of minimizing the execution time of the application, it is also important to minimize the monetary cost of using cloud resources, i.e., there exists a trade-off between performance and monetary cost.

Infrastructure-as-a-Service (IaaS) existing cloud platforms (e.g., Amazon EC2, Microsoft Azure, Google Cloud, etc.) enable users to dynamically acquire resources, usually as virtual machines (VMs), according to their application requirements (CPU, memory, I/O, etc,) in a pay-as-you-use price model. They usually offer different classes of VMs which ensure different guarantees in terms of availability and volatility, provisioning the same resource through multiple pricing models. For instance, in Amazon EC2, there are basically three classes: (i) *reserved* VM instances, where the user pays an upfront price, guaranteeing long-term availability; (ii) *on-demand* VM instances which are allocated for specific time periods and incur a fixed cost per unit time of use, ensuring availability of the instance during this period; (iii) *spot* VM instances which are unused instances available for lower price than on-demand price.

The availability of spot VMs instances fluctuates based on the spot market's current demand. If there are not enough instances to meet clients demands, the VM can be interrupted by the cloud provider (temporarily or definitively). Despite the risk of unavailability, the main advantage of spot VMs is that their cost is much lower than on-demand VMs since the user requests unused instances at steep discounts, reducing the costs significantly. With Amazon's more recent announcement, an interrupted spot can either terminate or hibernate[1]. Whenever a spot instance is hibernated by EC2, its memory and context are saved to the root of EC2 Block Storage (EBS) volume and, during the VM's pause, the user is only charged for EBS storage. EC2 resumes the hibernated instance, reloading the saved memory and context, only when there is enough availability for that type of instance with a spot price which is lower than the user's maximum price. Contrarily to terminated instances whose user is warned two minutes before the interruption of them, hibernated instances are paused immediately after noticing the user.

Our proposal in this work is to provide a static cloud scheduler for Bag-of-Tasks applications using, for cost sake, hibernate-prone spot instances as much as possible, respecting the application deadline constraints while also minimizing the monetary costs of bag-of-tasks applications. However, if a spot instance hibernates, it might happen that it will not resume within a time which guarantees the deadline constraints of the application. In this case, a *temporal failure* would take place, i.e., correct computation is performed but too late to be useful (inability to meet deadlines). Thus, in order to avoid temporal failure in case of spot instance hibernation, our scheduler statically computes the time interval that an hibernated instance can stay in this state without violating the application's deadline. If the instance does not resume till the end of this interval, our scheduler will move the execution of the current tasks of the spot instance as well as those not executed yet to on-demand instances, in order to guarantee

[1]https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-interruptions.html

the application's deadline. Note that even after migrating the remaining task execution to on-demand VMs, the scheduler continues to look forward to minimizing monetary costs.

II. Related Work

Replication and resubmission of tasks are mechanisms widely used to tolerate failures [1], [2], [3]. 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 on different computing instances [2], [3]. The copies of a task are executed only when the primary task fails. In order to reduce the response time in case of failure, overlapping techniques are proposed [2] in a Grid context where a backup is scheduled for each primary on a different host. A backup is executed when its primary cannot complete execution due to a failure but it does not require fault diagnosis. Zheng et al. [2] propose an algorithm to find an optimal backup schedule for each independent task. Wang et al. [3] 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 the tasks deadlines have enough time for executing backups in case of failure. Then, several works study active replication [4], [5], [6] allowing backups to execute concurrently with its primaries. Al-Omari et al. [4] improve the primary-backup scheme by proposing the primary-backup-overloading technique, in which the primary of a task can be scheduled onto the same or overlapping time interval with the backup of another task on a processor. In [5], authors present 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. [6] 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 latter take the execution control if the former fail. Neither of them use backup tasks to avoid temporal failure. In addition, in our case, they are only executed in case of VM's hibernation and risk of temporal failures.

Some works take into account Amazon spot VMs instance features. Authors of [7] propose to switch to on-demand resources when there is no spot instance available to ensure the desired performance. Using both on-demand and spot VM instances, SpotCheck [8] provides the illusion of an IaaS platform that offers always-available on-demand VMs for a cost near that of spot VMs. Also claiming performance of on-demand VMs, but at a cost near that of the spot market, the authors in [9] present the SpotOn batch service computing, that uses fault-tolerance mechanism to mitigate the impact of spot revocations. To our knowledge no work studies the impact of the new hibernation feature of spot instances on scheduling algorithms.

III. A Static Scheduler of Bag-of-tasks Applications in Clouds

Aiming at reducing monetary costs, our proposed scheduling uses hibernate-prone spot instances. However, due to the possibility of hibernation and also the need to meet the application's deadline, the scheduler might migrate tasks that run on spot instances to on-demand ones, whenever the duration of an instance hibernation would induce a temporal failure. We denote *primary* tasks those which are allocated on VMs (spot or on-demand) that guarantee application's deadline with minimum monetary cost and we denote *backup* tasks those which are allocated on on-demand VMs and were originally primary tasks allocated on spot VMs. Backup tasks are only executed in case the hibernation state remains for such a long period of time that it is impossible to meet the deadline of the application, thus avoiding temporal failures. Therefore, a task might have two versions (primary and backup) which are statically scheduled on two different cores with time exclusion.

Figure 1 shows an example where the hibernation does not require backup task execution. In this example, a spot instance starts hibernating in time p and finishes in y, before the time $start_bkp$, when the backups should be triggered. Then, the deadline D can be met without executing the backups. On the other hand, Figure 2 presents a case where it is necessary to execute the backup tasks in an on-demand virtual machine to meet the deadline, since the hibernation exceeded $start_bkp$.

A. Problem Formulation

Let M be the set of virtual machines, B the set of tasks that compose a bag-of-task application, and $T = \{1, \ldots, D\}$ the set of feasible periods, where D is the deadline defined by the user. Each $vm_j \in M$ has a memory capacity of m_j gigabytes, nc_j virtual cores and can be allocated in one of the markets (spot or on-demand). Besides that, each task $t_i \in B$ requires a know amount of memory rm_i and is executed in only one core of the nc_j cores of vm_j. Let $Queue_j \subset B$ be the set with all tasks scheduled on vm_j, when a task t_i is scheduled to a vm_j, its start time and end time are given by $st_{i,j} \in T$ and $end_{i,j} \in T$, respectively.

When a VM is allocated for a user, he/she pays for a full-time interval called $slot$. That time is usually one hour. Therefore, if a VM is used for 61 minutes the user will be charged for two $slots$ (120 minutes). Note that one slot can correspond to several periods of T. For example, if each period corresponds to one minute, a slot of one hour would correspond to 60 periods. It is, thus, in the user's best interest to maximize the use of a slot already allocated. Let $stSlot_j \in T$ and $endSlot_j \in T$ be the period when the first slot was allocated to vm_j, and the end time of the last allocated slot for this same VM respectively, such that $stSlot_j < endSlot_j$. Whenever the end time $end_{i,j}$ of a task t_i allocated to vm_j exceeds the $endSlot_j$, the user has to pay for another full interval. Thus, if part of that interval is not used by any task, we have a waste of time. To compute the waste time of a vm_j, we define $waste_j$ in Equation 1, that is

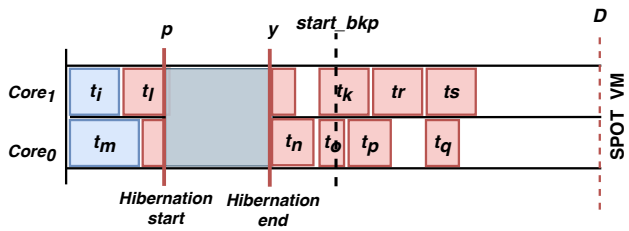

Fig. 1: Hibernation without Backup Execution.

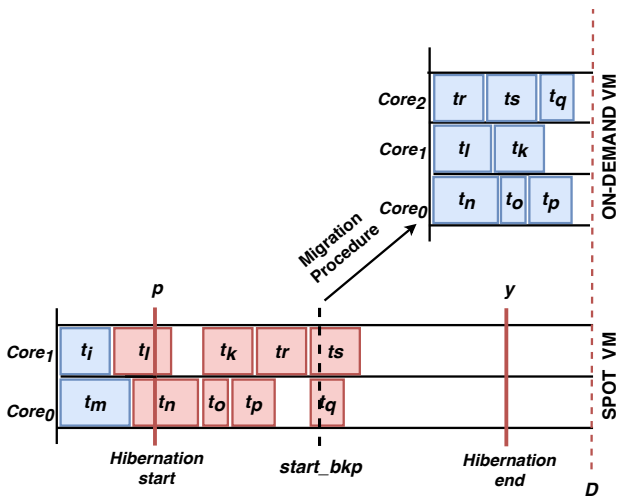

Fig. 2: Hibernation with Backup Execution.

the time interval inside the last contracted $slot$ at which vm_j remains idle after executing all tasks allocated to it.

$$waste_j = endSlot_j - \max_{\forall t_i \in Queue_j} (end_{i,j}) \quad (1)$$

B. Computing D_{spot}

Before scheduling the primary tasks, it is necessary to estimate the D_{spot} value, that is used to ensure that if a hibernation occurs, there will be enough time to execute the backup tasks respecting the deadline D. The D_{spot} is estimated using Algorithm 1. As can be seen, in addition to the deadline D and sets B and M, Algorithm 1 also receives max_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[2].

We can assume that, using the proposed primary scheduling algorithm (Algorithm 2), the number of tasks n allocated to a VM can be calculated as presented in line 1 of Algorithm 1. After that, a set $W \subset B$ 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 a single VM represents the worst primary makespan case. In line 7, that makespan, called mkp_w, is estimated by considering the slowest VM of M. Thus, we calculate the value of D_{spot} as the difference between D and mkp_w (line 8).

Algorithm 1 *Compute D_{spot}*

Input: B, M, max_spot, D
1: $n = \lceil \frac{|B|}{max_spot} \rceil$;
2: $W = get_longest_tasks(n, B)$;
3: $vm_s = get_slowest_vm(M)$;
4: **for all** $t_i \in W$ **do**
5: $\quad allocate(t_i, vm_s)$;
6: **end for**
7: $mkp_w = get_makespan(vm_s)$;
8: $D_{spot} = D - mkp_w$;

C. Primary Task Scheduling

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 and on-demand VMs). Tables I and II present the used variables and functions respectively. The algorithm receives B, M, D, and D_{spot} as input parameters. Since the objective is to respect the application deadline (even in the presence of hibernation) while minimizing monetary costs, all the choices made by the heuristic are guided by the VMs' prices, and by the deadline D defined by the user.

Initially, tasks are ordered in descending order by the memory size they require (line 1). Then, for each task, the algorithm applies a best fit heuristic that tries to allocate it to a virtual core of a VM with already allocated slots. The algorithm first choose the VM with the highest waste o time (lines 6 to 12), that has enough memory and also ensures that the task insertion will respect D_{spot}. If such a VM does not exist, the heuristic tries to allocate new slots in an already allocated VM with enough memory to execute the task, but now with the smallest $waste$ (lines 15 to 22). Similarly to the previous case, the slot allocation must not violate D_{spot} (line 18).

Allocating slots 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 24) and, then, it chooses the market where this VM shall be acquired: on-demand or spot, considering the offered prices (lines 25 to 29). Finally, it updates the primary scheduling map (line 34).

D. Backup Task Scheduling

Let $Succ_{k,j} \subset Queue_j$ be a set containing task t_k and all its successors, i.e., all tasks that are allocated to the same core and where t_k is allocated (vm_j) and that execute after the end of t_k. Let $Parallel_{i,j} \subset Queue_j$ be a set containing all tasks that execute in parallel with t_i in vm_j. In order to avoid temporal failures due to vm_j's hibernation while executing t_i in one of its cores, it is necessary to determine which backup tasks must be executed in this case. To this end, we define $RG_{i,j} \subset Queue_j$, as presented in Equation 2. The set $RG_{i,j}$ is obtained

[2]https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-limits.html

TABLE I: Variables of Scheduling Heuristic

Name	Description
Variables used in the Primary Scheduling Heuristics	
B	Set of tasks
M	Set of VMs
D	Deadline defined by the user, to be respected even in presence of VM hibernation
D_{spot}	Parameter that determines the maximum occupation period of a spot VM
A	Set of VMs selected to execute primary tasks
$allocated$	Boolean variable that indicates whether task t_i was successfully scheduled
$slot$	Minimum contracted time for a VM (for example, in AWS the slot is 1 hour)
n	Number of contracted slots
$endSlot_k$	End of last contracted slot of vm_k
vm_k^{market}	Market where vm_k will be contracted: on-demand or spot
PM	Scheduling map of primary tasks containing VMs and the corresponding execution queues
Variables used in the Backup Scheduling Heuristics	
B_VM	Set of VMs selected to execute backup tasks
$Queue_j$	Set of primary tasks scheduled on a vm_j
$RG_{i,j}$	Set of backup tasks to be executed due to hibernation of vm_j along t_i execution
S_VM_i	Set of VMs selected to execute backup tasks of the $RG_{i,j}$
$start_bkp_{i,j}$	Time when the migration of tasks in vm_j must start due to hibernation along t_i execution
BM	Scheduling map of backup tasks containing VMs and the starting times of backup tasks

TABLE II: Functions and Procedures of Scheduling Heuristic

Name	Description
Functions and Procedures used in Primary Scheduling Heuristic	
$sort(B)$	Sorts the set of tasks B in descending order by memory size demand
$sort_by_max_waste(A)$	Sort the set of VMs A in descending order by the waste size
$sort_by_min_waste(A)$	Sort the set of VMs A in ascending order by the waste size
$check_allocation(t_i, vm_j, D_{spot})$	Checks if vm_j has an already contracted slot with enough idle time, and if the allocation of t_i to a core of vm_j respects the D_{spot} limit and if vm_j has available memory to meet the requirement of the task (rm_i)
$allocate(t_i, vm_j)$	Allocates task t_i to a core of vm_j
$number_of_slots(t_i, vm_j)$	Compute the number of slots necessary to execute task t_i in vm_j
$bestVM_time(t_i, M)$	Selects the VM that executes task t_i with the minimum number of periods of time
$allocate_slots(vm_k, n)$	Allocate (contract) n slots in vm_k
$create_primary_map(A)$	Create the scheduling map of primary tasks
Functions and Procedures used in Backup Scheduling Heuristic	
$create_RG(t_i, Queue_j)$	Create $RG_{i,j}$ using Equation 2
$bestVM_cost(t_k, S_VM_i)$	Select a VM of S_VM_i that executes t_k with minimum monetary cost
$allocate(t_k, vm_{bkp})$	Insert backup task t_k into the execution queue of vm_{bkp}
$runtime(RG_{i,j}, S_VM_i)$	Calculate the number of periods necessary to execute all tasks in $RG_{i,j}$ using VMs of S_VM_i
$update(B_VM, S_VM_i)$	Include VMs of S_VM_i into B_VM
$create_backup_map(B_VM)$	Create the scheduling map of backup tasks

by the union of all $Succ_{k,j}$, such that $t_k \in Parallel_{i,j}$ or $t_k = t_i$. We also define the set $S_VM_i \subset M$, that contains all VMs that will be used to execute backup tasks of $RG_{i,j}$, if a migration occurs. Figure 3 shows an example of $Succ_{k,j}$ and $RG_{k,j}$ sets of task t_k allocated to vm_j.

$$RG_{i,j} = \bigcup_{t_k \in (Parallel_{i,j} \cup \{t_i\})} Succ_{k,j} \tag{2}$$

We also define the backup start time, $start_bkp_{i,k}$, as presented in Equation 3. It defines how long the hibernation state of vm_j can be tolerated before any action of migrating tasks of $RG_{i,j}$ to backup ones is triggered.

$$start_bkp_{i,j} = D - runtime(RG_{i,j}, S_VM_i) \tag{3}$$

Such that $runtime(RG_{i,j}, S_VM_i)$ is the number of periods necessary to execute all tasks of $RG_{i,j}$ in the VMs of S_VM_i plus the number of periods necessary to boot the VMs of S_VM_i.

The proposed backup scheduling algorithm is presented in Algorithm 3, where Table I shows the used variables and Table II describes the used procedures and functions. As can be seen in line 4, $RG_{i,j}$ is created for each task $t_i \in Queue_j$.

This algorithm employs a scheduling strategy similar to that presented in Algorithm 2, in which tasks are scheduled on different VMs using a best-fit heuristic. However, unlike the Algorithm 2, in Algorithm 3, the VMs selection prioritizes the on-demand VM with the cheapest monetary cost, resulting from the product of its price and the execution time of a backup task on it.

Note that the backup scheduling has to ensure that if a migration event occurs, the number of periods required to perform the backup tasks respects the deadline. Thus, the VMs chosen in the function get_best_VM (lines 9 and 11) guarantees that $end_i + runtime(RG_{i,j}, S_VM_i) < D$, where end_i is the end time of the primary task t_i .

After scheduling all backup tasks of $RG_{i,j}$, the period when the migration of tasks will have to start to meet the deadline, $start_bkp_{i,j}$, is computed (line 16).

IV. Experimental Results

This section presents execution times and monetary costs of accomplished with real BoT applications, using the configuration of Amazon EC2 virtual machines, and considering a real VMs market history. According to the information on Amazon Web Server (AWS)[3], only the VMs of families C3, C4, C5,

[3]https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-interruptions.html

Algorithm 2 *Primary Task Scheduling*

Input: B, M, D, D_{spot}
$sort(B)$;
$A \leftarrow \emptyset$; {set of allocated VMs}
for all $t_i \in B$ **do**
  $sort_by_max_waste(A)$; {using Equation 1}
  $allocated \leftarrow False$; {Check if vm_j has sufficient time and memory in an already allocated slot to execute t_i without violating the limit D_{spot}}
  for all $vm_j \in A$ **do**
    if $check_allocation(t_i, vm_j, D_{spot})$ **then**
      $allocate(t_i, vm_j)$;
      $allocated \leftarrow True$;
      $stop_loop$;
    end if
  end for
  {Check if it will be necessary to allocate a new slot on an already allocated VM or if it will be necessary to allocate a new VM}
  if not $allocated$ **then**
    $vm_k \leftarrow \emptyset$;
    $sort_by_min_waste(A)$;
    for all $vm_j \in A$ **with enough memory do**
      $n \leftarrow number_of_slots(t_i, vm_j)$; {Get the number of slots necessary to execute t_i on vm_j}
      if $endSlot_j + (n * slot) < D_{spot}$ **then**
        $vm_k \leftarrow vm_j$;
        $stop_loop$;
      end if
    end for
    if vm_k **is** $\emptyset$ **then**
      $vm_k \leftarrow bestVM_time(t_i, M)$;
      if $spotPrice(vm_k) < on\text{-}demandPrice(vm_k)$ **then**
        $vm_k^{market} \leftarrow spot$;
      else
        $vm_k^{market} \leftarrow on\text{-}demand$;
      end if
    end if
    $n \leftarrow number_of_slots(t_i, vm_k)$;
    $allocate_slots(vm_k, n)$; {allocate the number of slots required to execute t_i in vm_k}
    $allocate(t_i, vm_k)$ {Update the VMs sets}
    $update(A, vm_k)$;
  end if
end for
$PM \leftarrow create_primary_map(A)$;

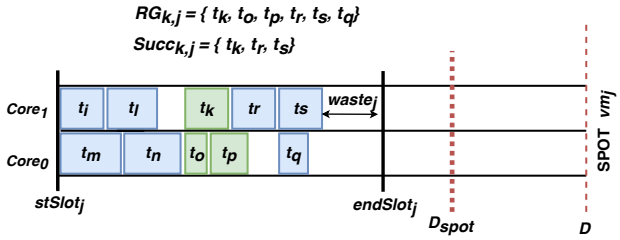

Fig. 3: Example of a task t_k, its successors and which backup tasks might be executed to prevent vm_j from a temporal failure while executing task t_k ($RG_{k,j}$).

Algorithm 3 *Backup Task Scheduling*

Input: A, M, D
$B_VM \leftarrow \emptyset$;
for all $vm_j \in A$ **such that** $vm_j^{market} = spot$ **do**
  for all $t_i \in Queue_j$ **do**
    $RG_{i,j} \leftarrow create_RG(t_i, Queue_j)$
    {Schedule each $t_k \in RG_{i,j}$ on a set of VMs. The VMs choice is guided by the monetary cost resulting from the product of price and execution time}
    $S_VM_i \leftarrow \emptyset$;
    for all $t_k \in RG_{i,j}$ **do**
      $vm_{bkp} \leftarrow \emptyset$;
      {Select a VM able to execute t_k, without violating the deadline, with the smallest monetary cost}
      $vm_{bkp} \leftarrow bestVM_cost(t_k, S_VM_i)$;
      if vm_{bkp} **is** $\emptyset$ **then**
        $vm_{bkp} \leftarrow bestVM_cost(t_k, M)$;
      end if
      $allocate(t_k, vm_{bkp})$;
      $update(S_VM_{t_i}, vm_{bkp})$;
    end for
    $start_bkp_{i,j} \leftarrow D - runtime(RG_{i,j}, S_VM_i)$;
    $update(B_VM, S_VM_i)$;
  end for
end for
$BM = create_backup_map(B_VM)$;

M4, M5, R3, and R4 with memory below 100 GB, running in the spot market, are able to hibernate if an interrupt occurs. Therefore, for the purposes of this work, the fourth generation general purpose VMs (M4) and the third and fourth generation VMs optimized for computation (C3 and C4) were used. By choosing the third and fourth generation VMs, it was possible to compute the VM slowdown using the data from [10]. A VM slowdown is defined as $\frac{P_B}{P_j}$, where P_B is the processing capacity of the machine used to calculate the basis time, and P_j is the processing capacity of the vm_j. Thus, the slowdown represents the processing capacity of a VM when compared with the machine used to compute the basis time duration.

The workload used in the evaluation were obtained from [11], a database that contains the execution traces of jobs submitted to Google's servers throughout the month of March 2011. Based on these traces, we have defined: (i) the number of tasks of a job; (ii) the execution time of each task of the job; and (iii) the average memory footprint. For the experiments, four BoT-type jobs were chosen from the first 10 days of the traces. Table III summarizes the main characteristics of these jobs, followed by the corresponding deadlines considering the virtual machines used in our tests. We adopted the shortest deadlines which enable the generation of valid primary and backup scheduling, for each job. These values were computed iteratively starting with $D = 1(h)$ in increments of 1 hour, stopping at the first valid scheduling given by the algorithms 2 and 3.

The execution times were obtained from Google machines used in 2011. As the hardware information and computational capacity of these machines are not provided, we assumed that these times were obtained with the VM with the lowest compu-

tational power, whose memory capacity was sufficient to meet the requirements of the tasks. As we can observe in Table IV, among the VMs, the ones containing VCPUs with the lowest computational power are c3.large and m4.large. Therefore, they are considered our baseline regarding processing capacity. Spot and on-demand VM prices were obtained on September 10, 2018, considering us-east-1 region. Table IV shows the characteristics of these VMs, along with the corresponding slowdown values of their VCPUs. Based on the latter and considering, as mentioned above, that the duration of the tasks, extracted from Google traces, were obtained from execution them on the slowest VMs (base time duration), the duration of each task in the other VMs was obtained through the product of the respective slowdown value by its base duration.

TABLE III: BoT attributes and deadlines

JOB ID	#Tasks	Memory	Execution Time of a Task			Deadline
			min	avg	max	
J207	31	6.10 GB	7.03 (h)	19.75 (h)	49.31 (h)	17.00 (h)
J402	103	2.90 GB	7.87 (h)	29.83 (h)	94.04 (h)	29.00 (h)
J819	68	3.97 GB	6.42 (h)	18.87 (h)	51.53 (h)	16.00 (h)
J595	97	3.14 GB	7.15 (h)	45.80 (h)	120.39 (h)	40.00 (h)

TABLE IV: VMs attributes

Type	#VCPUs	Memory	On-demand price	Spot price	slowdown
m4.large	2	8.00 GB	0.1$	0.0324$	1.000
c3.large	2	3.75 GB	0.10$	0.0294$	1.000
c4.large	2	3.75 GB	0.1$	0.0308$	0.655
m4.2xlarge	8	32.0 GB	0.4$	0.1326$	0.672
m4.xlarge	4	16.0 GB	0.2$	0.0648$	0.477
m4.4xlarge	16	64.0 GB	0.8$	0.3257$	0.513
c3.xlarge	4	7.50 GB	0.21$	0.0588$	0.323
c4.xlarge	4	7.50 GB	0.199$	0.0617$	0.332
c3.2xlarge	8	15.0 GB	0.42$	0.1175$	0.475
c4.2xlarge	8	15.0 GB	0.398$	0.1262$	0.447
c3.4xlarge	16	30.0 GB	0.84$	0.2350$	0.163
c4.4xlarge	16	30.0 GB	0.796$	0.2535$	0.162
c4.8xlarge	36	60.0 GB	1.591$	0.4986$	0.162
c3.8xlarge	32	60.0 GB	1.68$	0.4700$	0.161

A. Experimental results in different hibernation scenarios

In order to evaluate the effectiveness of our scheduling solution in terms of makespan and monetary cost, we compared it with a strategy (*On-demand*) that uses only on-demand virtual machines, while for evaluating the impact of hibernation, we compared it with a strategy that migrates tasks as soon as the VM, where the tasks have been allocated, hibernates (*Immediate Migration*), i.e., the latter does not consider the possibility that the VM might resume. Furthermore, we also consider two possible scenarios of execution of our scheduling: (1) no spot VM hibernates (*No Hibernation*) and (2) a spot VM hibernates and, in this case, either the tasks need to be migrated (*Hibernation with Migration*) or the VM resumes in time to not violate deadline (*Hibernation*). In case of hibernation, the latter initiates two hours after the job starts. For the *Hibernation with Migration* execution, the duration of hibernation is set to 1000 hours, thus forcing task migration; for *Hibernation*, hibernation duration is just 3 hours and, therefore, task migration is not carried out. Aiming a more accurate analysis of the results, only one spot VM can hibernate in *Hibernation* and *Hibernation with Migration* executions. In addition, only one backup migration takes place in *Hibernation with Migration* execution. Finally, the experiments randomly select the spot VM that should hibernate.

1) Hibernation without Migration: Figure 4 presents the monetary costs in the *Hibernation* scenario, i.e., our scheduling does not migrate tasks because the hibernated spot VM instance resumes in time to meet the application's deadline. As we can observe in the figure, its monetary cost is similar to the one without hibernation, represented by the *Hibernation* and *No Hibernation* bars respectively. Such a result is expected since, according to the new pricing policy defined by AWS in December 2017, the user only pays for the time the spots are running, and during hibernation, the user is charged only for storage, whose price on September 10, 2018 was 0.10 per GB per month. As, in the experiments, the maximum hibernation time is shorter than 30 hours, it is, thus, negligible. When our solution is compared with the one that migrates tasks as soon as the hibernation occurs (*Immediate Migration*), we observe that the latter is more expensive than the second in 59.97%, 26.74%, 55.15% and 40.51%, for J207, J402, J819 and J595, respectively. Such a difference in price can be explained since in the *Immediate Migration*, the user was charged for the two hours of execution of the spot VMs as well as for the on-demand VMs used for migration. On the other hand, it is worth mentioning that the *Immediate Hibernation* monetary cost is, for the four jobs, on average, 57.31% lower than the (*On-demand*) one. This happens because part of the tasks were executed as primary ones in spot VM with high computational power, and, therefore, fewer slots were needed to complete execution on the on-demand VMs.

Figure 5 shows that our solution, *Hibernation*, has a makespan longer than the *Immediate Migration* strategy. This occurs because the former has an additional 3 hours due to hibernation, while the latter migrates immediately, continuing running the job's tasks within this 3 hours.

In contrast, since the VMs usually chosen by the *Immediate Migration* strategy are low cost ones, performing poorly, its makespan can be longer than *On-demand* and *No Hibernation* ones, that allocate VMs with higher computation power. Moreover, when the virtual machine hibernates, the executing task is re-started from the beginning in another VM. So, its execution time can be computed (in the makespan) almost twice in the worst case.

2) Hibernation with Migration: Figures 6 and 7 respectively show the monetary costs and makespan in the scenario where our scheduling (*Hibernation with Migration*) migrates tasks of a hibernated spot VM.

The monetary cost of *Hibernation with Migration* strategy is equal to the *Immediate Migration* since the backup map used by both of them are similar. These costs are higher (on average, 30.74% in our experiments) than the one required by the primary scheduling alone (*No Hibernation*), since the former use spot VMs within the first two hours of execution, as well as on-demand VMs for backup migration. On the

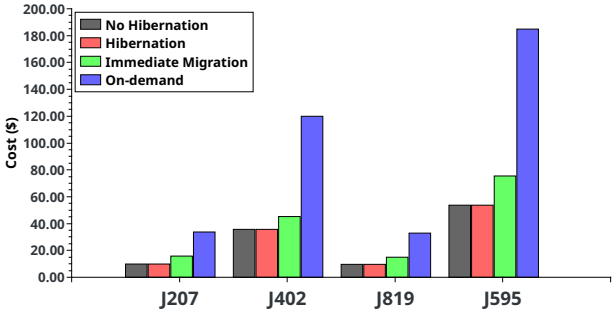

Fig. 4: Monetary costs considering that the spot VM resumes.

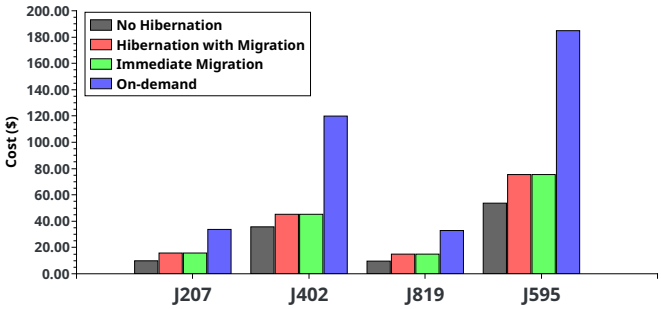

Fig. 6: Monetary costs with tasks migration.

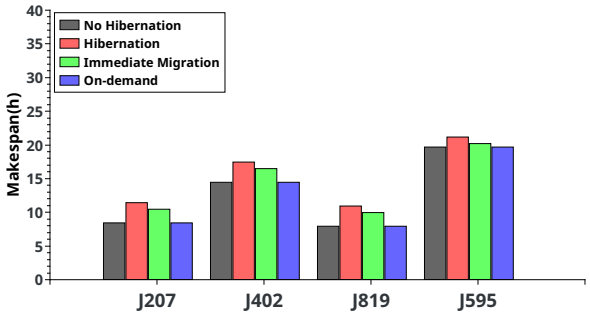

Fig. 5: Makespans considering VM the spot VM resumes.

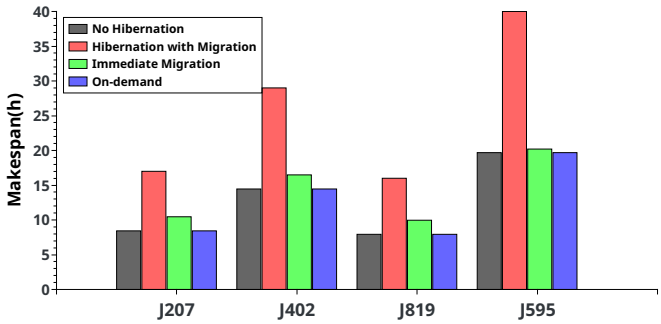

Fig. 7: Makespans with tasks migration.

other hand *On-demand* strategy costs are 136.00% higher than *Hibernation with Migration* costs. In terms of makespan, the *Hibernation with Migration* makespans is close to the deadlines defined in the Table III. Such a behaviour is expected since our approach waits till the $start_bkp$, which is the latest time that hibernation can be tolerated without exceeding the deadline. Note that in the case of the *Immediate Migration* strategy, the makespan is shorter than the *Hibernation with Migration* one. In our experiments, this difference was, on average, 74.26%.

Note that, although in some cases, the tasks can migrate to VMs of equivalent processing powers (see the case of J207), even in *Immediate Migration* strategy, the makespan increases. As pointed out in the previous section, it happens because the execution time of a task, initially started in a VM that hibernates along its execution, can be computed almost twice, when it migrates, in the worst case.

When comparing *Hibernation* with *Hibernation with Migration*, the duration of hibernation has an impact in both makespans due to the duration of the execution itself. However, in the case of *Hibernation*, where the hibernated spot VM resumes in time to respect the deadline, the monetary cost is lower than *Hibernation with Migration*, as we can confirm in Figures 4 and 6.

B. Experimental Results with hibernation based on the variation of spots price

The results presented in this section are from experiments that consider spot price variations for regions us-east-1 and zone us-east-1a between March and April of 2017, defining hibernation traces for the VMs of Table IV. That history of price variation predates the changes in AWS pricing policies, occurred in December 2017, which stabilized the prices of VMs such that peaks of variation ceased to occur[4].

The hibernation traces were generated considering a fixed threshold of $\$0.4$, which represents the average price value in the first 24 hours of the history. Thus, the onset of hibernation is the period in which the VM price is higher than this value. Analogously, when the price drops to a value below the threshold, we consider that the VM resumes execution. The generated traces have two hibernation points: (1) c4.8xlarge VMs hibernation at 4.21 hours after the start of execution and lasting 43.51 minutes; (2) c3.4xlarge VMs hibernation at 23.7 minutes after the start of execution and lasted 1.22 hours.

The number of VMs affected by hibernation is not the same for all evaluated jobs. While in the J207 and J819 jobs only 2 VMs hibernate, in Job J402 there are 8 hibernations of different VMs. This variation is expected, since different job tasks are scheduled to VMs of different types.

As can be observed in Figure 8, our solution, *Hibernation*, presents the lowest monetary cost, with an average difference of 167.43 %, relative to the *Immediate Migration*'s one, and 240.94 % in relation to the *On-demand*'s one. It is noteworthy that in Job J402, *Immediate Migration* has a cost which is 6.73 % higher than the *On-demand*'s one. The former used 8 on-demand VMs for migration, which raised the monetary

[4]https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/

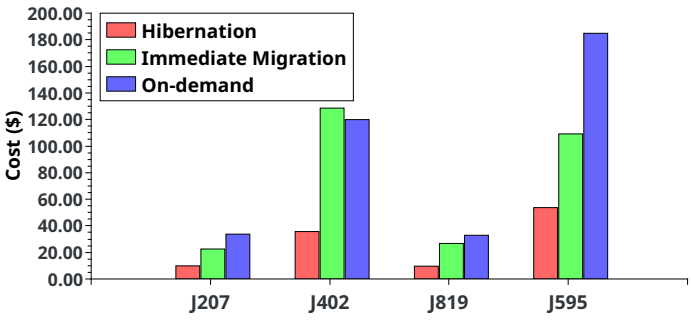

Fig. 8: Monetary costs with hibernation based on AWS price history.

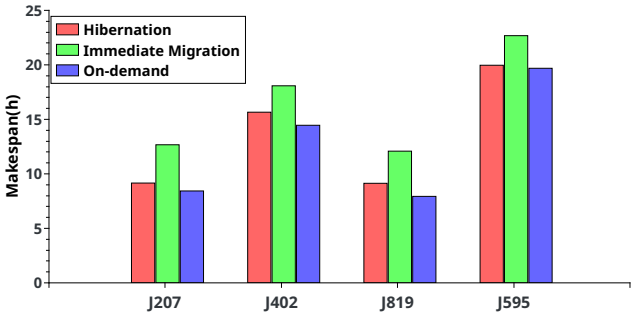

Fig. 9: Makespans with hibernation based on AWS price history.

cost, added to the costs of the VMs spots used until the beginning of their hibernation. On the other hand, for Job J402, our approach presents a significantly lower cost than the *Immediate Migration*'s one (that is 260.26 % higher than our approach), since the duration of none of the hibernation of the corresponding spot VMs triggered the migration of their tasks.

Regarding makespan, shown in Figure 9, our approach is 7.52 % longer than *On-demand*'s one and 24.92 % shorter than *Immediate Migration*'s one. These difference can be explained since the duration of VMs hibernation is up to 1.22 hours, it is not necessary to start the task migration process in any of the evaluated jobs. Therefore, this increase is due only to the hibernation of the VMs. On the other hand, in *Immediate Migration* the scheduling of backup tasks chooses firstly cheaper on-demand VMs, usually with lower computational power.

Although our approach increases the makespan when compared *On-demand*'s one, the monetary costs are lower than the two other approaches. Thus, the results from the experiments with the hibernation trace confirm those from the previous experiments.

V. Concluding Remarks and Future Work

This paper proposed a static scheduling for bag-of-task applications with deadline constraints, using both hibernation-prone spot VMs (for cost sake) and on-demand VMs. Our scheduling aims at minimizing monetary costs of bag-of-tasks, respecting application's deadline and avoiding temporal failures. We evaluate the proposed strategy using the information of BoT applications from real scenarios, and the results confirmed the effectiveness of our scheduling and that it tolerates temporal failures.

As future work, we intend to work in a dynamic version of the proposed scheduling which periodically takes 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.

Acknowledgment

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

References

[1] K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertész, and P. Kacsuk, "Fault detection, prevention and recovery in current grid workflow systems," in *Grid and Services Evolution, Proceedings of the 3rd CoreGRID Workshop on Grid Middleware, June 5-6, 2008, Barcelona, Spain*, 2008, pp. 1–13.

[2] Q. Zheng, B. Veeravalli, and C. Tham, "On the design of fault-tolerant scheduling strategies using primary-backup approach for computational grids with low replication costs," *IEEE Trans. Computers*, vol. 58, no. 3, pp. 380–393, 2009.

[3] J. Wang, W. Bao, X. Zhu, L. T. Yang, and Y. Xiang, "FESTAL: fault-tolerant elastic scheduling algorithm for real-time tasks in virtualized clouds," *IEEE Trans. Computers*, vol. 64, no. 9, pp. 2545–2558, 2015.

[4] R. Al-Omari, A. K. Somani, and G. Manimaran, "Efficient overloading techniques for primary-backup scheduling in real-time systems," *J. Parallel Distrib. Comput.*, vol. 64, no. 5, pp. 629–648, 2004.

[5] W. Cirne, F. V. Brasileiro, D. P. da Silva, L. F. W. Góes, and W. Voorsluys, "On the efficacy, efficiency and emergent behavior of task replication in large distributed systems," *Parallel Computing*, vol. 33, no. 3, pp. 213–234, 2007.

[6] A. Benoit, M. Hakem, and Y. Robert, "Fault tolerant scheduling of precedence task graphs on heterogeneous platforms," in *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, 2008, pp. 1–8.

[7] 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.

[8] 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.

[9] 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.

[10] L. Gillam and J. O'Loughlin, "Should infrastructure clouds be priced entirely on performance? an ec2 case study," *International Journal of Big Data Intelligence*, vol. 1, no. 4, pp. 215–229, 2014.

[11] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.