Future Generation Computer Systems 95 (2019) 68–78

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

An efficient cloud scheduler design supporting preemptible instances

Álvaro López García *, Enol Fernández del Castillo, Isabel Campos Plasencia

Institute of Physics of Cantabria, Spanish National Research Council — IFCA (CSIC—UC), Avda, los Castros s/n, 39005 Santander, Spain

HIGHLIGHTS

- Discussion on resource allocation in Clouds from the resource provider point of view.
- Novel scheduling algorithm that allows to execute preemptible instances.
- The scheduler does not incur in a noticeable an extra overhead.
- New cloud usage and payment models in current Cloud Management Frameworks.

ARTICLE INFO

Article history:
Received 5 February 2018
Received in revised form 10 December 2018
Accepted 26 December 2018
Available online 3 January 2019

Keywords:
Cloud computing
Scheduling
Preemptible instances
Spot instances
Resource allocation

ABSTRACT

Maximizing resource utilization by performing an efficient resource provisioning is a key factor for any cloud provider: commercial actors can maximize their revenues, whereas scientific and non-commercial providers can maximize their infrastructure utilization. Traditionally, batch systems have allowed data centers to fill their resources as much as possible by using backfilling and similar techniques. However, in an IaaS cloud, where virtual machines are supposed to live indefinitely, or at least as long as the user is able to pay for them, these policies are not easily implementable. In this work we present a new scheduling algorithm for IaaS providers that is able to support preemptible instances, that can be stopped by higher priority requests without introducing large modifications in the current cloud schedulers. This scheduler enables the implementation of new cloud usage and payment models that allow more efficient usage of the resources and potential new revenue sources for commercial providers. We also study the correctness and the performance overhead of the proposed scheduler against existing solutions.

1. Introduction

Infrastructure as a Service (IaaS) Clouds make possible to provide computing capacity as a utility to the users following a pay-per-use model. This fact allows the deployment of complex execution environments without an upfront infrastructure commitment, fostering the adoption of the cloud by users that could not afford to operate an on-premises infrastructure. In this regard, Clouds are not only present in the industrial ICT ecosystem, and they are being more and more adopted by other stakeholders such as public administrations or research institutions.

Indeed, clouds are nowadays common in the scientific computing field [1–4], due to the fact that they are able to deliver resources that can be configured with the complete software needed for an application [5]. Moreover, they also allow the execution of non-transient tasks, making possible to execute virtual laboratories, databases, etc. that could be tightly coupled with the execution environments. This flexibility poses a great advantage against traditional computational models – such as batch systems or even Grid computing – where a fixed operating system is normally imposed and any complimentary tools (such as databases) need to be self-managed outside the infrastructure. This fact is pushing scientific datacenters outside their traditional boundaries, evolving into a mixture of services that deliver more added value to their users, with the Cloud as a prominent actor.

Scientific cloud resource providers must face different resource scheduling challenges when compared with commercial providers. One important aspect is that normally, science cloud users do not usually pay for these resources – or at least they are not charged directly – for their consumption, and normally resources are paid via other indirect methods (like access grants), with users tending to assume that resources are *for free*. On the one hand traditional scientific computing facilities tend to work on a fully saturated manner, aiming at the maximum possible resource utilization level. However, on the other hand, cloud promises on-demand and interactive access to the resources, and as a matter of fact, this is being considered as one of the post promising facts of the cloud computing model [4]. These two aspects seem to be contradictory, as a saturated infrastructure cannot react to on-demand requests with ease. In this context, scheduling mechanisms and strategies that allow for a mixed allocation model become fundamental [6].

* Corresponding author.
E-mail addresses: aloga@ifca.unican.es (A. López García), enolfc@ifca.unican.es (E. Fernández del Castillo), iscampos@ifca.unican.es (I. Campos Plasencia).

https://doi.org/10.1016/j.future.2018.12.057
0167-739X/

Maximizing resource utilization by performing an efficient resource provisioning and workload scheduling is a fundamental for a scientific cloud provider (arguably this is an essential aspect for any resource provider). However, this is not a trivial task, since it is common that compute servers spawned in a scientific cloud infrastructure are not terminated at the end of their lifetime, resulting in idle resources, a state that is are not desirable as long as there is processing that needs to be done [4,7]. As already explained, this situation happens since scientific cloud users are not paying directly for their consumption, in contrast with commercial clouds, where users are being charged for their allocated resources, regardless if there is a real usage or not. Therefore, users tend to take care of their virtual machines, terminating them whenever they are not needed anymore. Moreover in the cases where users leave their resources running forever, the provider is still obtaining revenues for those resources.

Cloud operators try to solve this problem by setting resource quotas, hence limiting the amount of resources that a user or group is able to consume by doing a static partitioning of the resources [8]. However, this unflexible scheduling strategy automatically leads to an underutilization of the infrastructure since the partitioning needs to be conservative enough so that other users could utilize the infrastructure. Quotas impose hard limits that leading to dedicated resources for a group, even if the group is not using the resources.

On top of this, as already introduced before, cloud users expect to have on-demand resource provisioning, as this has been always promoted as one of the most compelling cloud characteristics [9]. In order to provide such access, an overprovisioning of resources is expected [10] in order to fulfill a user request, leading to an infrastructure where utilization is not maximized, as there should be always enough resources available for a potential request.

Taking into account that some processing workloads executed on the cloud do not really require on-demand access (but rather they are executed for long periods of time), a compromise between these two aspects (i.e. maximizing utilization and providing enough on-demand access to the users) can be provided by using idle resources to execute these tasks that do not require truly on-demand access [10]. This approach indeed is common in scientific computing, where batch systems provide scheduling strategies to maximize the resource utilization through *backfilling* techniques, where opportunistic access is provided to these kind of tasks.

However, unlike in batch processing environments, virtual machines (VMs) spawned in a Cloud do not have fixed duration in time and are supposed to live forever—or until the user decides to stop them. Commercial cloud providers provide specific VM types (like the Amazon EC2 Spot Instances[1] or the Google Compute Engine Preemptible Virtual Machines[2]) that can be provisioned at a fraction of a normal VM price, with the caveat that they can terminated whenever the provider decides to do so. This kind of VMs can be used to backfill idle resources, thus allowing to maximize the utilization and providing on-demand access, since normal VMs will obtain resources by evacuating Spot or Preemptible instances.

In this paper we propose an efficient scheduling algorithm that combines the scheduling of preemptible and non preemptible instances in a flexible way. The proposed solution is modular in order to allow different allocation, selection and termination policies, thus allowing resource providers to easily implement and enforce the strategy that is more suitable for their needs.

In order to evaluate our work we extend the OpenStack Cloud middleware with a prototype implementation of the proposed scheduler, as a way to demonstrate and evaluate the feasibility of our solution. The OpenStack Preemptible Instances Extension (OPIE) [11] has been developed as a prototype plugin for the OpenStack Cloud Management Framework. We have performed an evaluation of the performance of this solution, in comparison with the existing OpenStack scheduler as well as other prototype implementations delivering the same functionality.

The remainder of the paper is structured as follows. In Section 2 we present the related work in this field. In Section 3 we propose a design for an efficient scheduling mechanism for preemptible instances. In Section 4 we present an implementation of our proposed algorithm, as well as an evaluation of its feasibility and performance with regards with a normal scheduler. Finally, in Section 6 we present this work's conclusions.

2. Related work

The resource provisioning from cloud computing infrastructures using Spot Instances or similar mechanisms has been addressed profusely in the scientific literature in the last years [12]. However, the vast majority of this work has been done from the users' perspective. These works involve the usage and exploitation of Spot Instances [13] and few works tackle the problem from the resource provider standpoint.

Due to the unpredictable nature of the Spot Instances, there are several research papers that try to improve the task completion time – making the task resilient against termination – and reduce the costs for the user. Andrzejak et al. [14] propose a probabilistic model to obtain the bid prices so that the costs and performance and reliability can be improved. In [15–18] the task checkpointing is addressed so as to minimize costs and improve the whole completion time.

Related with the previous works, Voorsluys et al. have studied the usage of Spot Instances to deploy reliable virtual clusters [19,20], managing the allocated instances on behalf of the users. They focus on the execution of compute intensive tasks on top of a pool of Spot Instances, in order to find the most effective way to minimize both the execution time of a given workload and the price of the allocated resources. Similarly, in [21] the authors develop a workflow scheduling scheme that reduces the completion time using Spot Instances.

Jain et al. have performed studies in the same line, but focused on using a batch system that leverages the Spot Instances [22], learning from its previous experience – in terms of spot prices and workload characteristics – in order to dynamically adapt the resource allocation policies of the batch system.

Regarding Big Data analysis, several authors have studied how the usage of Spot Instances could be used to execute MapReduce workloads reducing the monetary costs, such as in [23,24]. The usage of Spot Instances for opportunistic computing is another usage that has awaken a lot of interest, especially regarding the design of an optimal bidding algorithm that would reduce the costs for the users [25,26]. There are already existing applications such as the vCluster framework [27] that can consume resources from heterogeneous cloud infrastructures in a fashion that could take advantage of the lower price that the Spot Instances should provide.

In spite of the above works, to the best of our knowledge, there is a lack of research in the feasibility, problematic, challenges and implementation of preemptible instances from the perspective of the IaaS provider. Singh and Chana [6] performed an extensive survey of resource scheduling in cloud environments where it can be seen that there is a clear lack of preemptible scheduling (or any similar mechanism). In spite of the user's interest in exploiting preemptible instances and the large commercial actors providing this alternative payment and access model, it is hard to find open source products or implementations of preemptible instances.

[1] http://aws.amazon.com/ec2/purchasing-options/spot-instances/.
[2] https://cloud.google.com/preemptible-vms/.

Amazon provides the EC2 Spot Instances,[3] where users are able to select how much they are willing to pay for their resources by *bidding* on their price in market where the price fluctuates accordingly to the demand. Those requests will be executed taking into account the following points:

- The EC2 Spot Instances will run as long as the published Spot price is lower than their bid.
- The EC2 Spot Instance will be terminated when the Spot price is higher than the bid (out-of-bid).
- If the user terminates the Spot Instance, the complete usage will be accounted, but if it gets terminated by the system, the last partial hour will not be accounted.

When an out-of-bid situation happens, the running instances will be terminated without further advise. This rough explanation of the Amazon's Spot Instances can be considered similar to the traditional job preemption based on priorities, with the difference that the priorities are being driven by an economic model instead by the usual fair-sharing or credit mechanism used in batch systems.

Google Cloud Engine (GCE)[4] has released a new product branded as *Preemptible Virtual Machines*.[5] These new Virtual Machine (VM) types are short-lived compute instances suited for batch processing and fault-tolerant jobs, that can last for up to 24 h and that can be terminated if there is a need for more space for higher priority tasks within the GCE.

Marshall et al. [10] delivered an implementation of preemptible instances for the Nimbus toolkit in order to utilize those instances for backfilling of idle resources, focusing on HTC fault-tolerant tasks. However, they did not focus on offering this functionality to the end-users, but rather to the operators of the infrastructure, as a way to maximize their resource utilization. In this work, it was the responsibility of the provider to configure the backfill tasks that were to be executed on the idle resources.

Nadjaran Toosi et al. have developed a Spot Instances as a Service (SIPaaS) framework, a set of web services that makes possible to run a Spot market on top of an OpenStack cloud [28]. However, even if this framework aims to deliver preemptible instances on OpenStack cloud, it is designed to utilize normal resources to provide this functionality. SIPaaS utilizes normal resources to create the Spot market that is provided to the users by means of a thin layer on top of a given OpenStack, providing a different API to interact with the resources. From the CMF point of view, all resources are of the same type, being SIPaaS the responsible of handling them, in different ways. In contrast, our work leverages two different kind of instances at the CMF level, performing different scheduling strategies depending on which kind of resource it is being requested. SIPaaS also delivers a price market similar to the Amazon EC2 Spot Instances market, therefore they also provide the Ex-CORE auction algorithm [29] in order to govern the price fluctuations.

Carvalho et al. have proposed [30] a capacity planning method combined with an admission service for IaaS cloud providers offering different service classes. This method allows providers to tackle the challenge of estimating the minimum capacity required to deliver an agreed Service Level Objective (SLO) across all the defined service classes. In the aforementioned paper Carvalho et al. lean on their previous work [31,32], where they proposed a way to reclaim unused cloud resources to offer a new *economy* class. This class, in contrast with the preemptible instances described here, still offer a SLO to the users, being the work on Carvalho et al. focused on the reduction of the changes that the SLO is violated due to an instance reclamation because of a capacity shortage.

Recent interest has been awaken in the OpenStack Cloud Management Framekwork Scientific Interest Group (SIG) [33] where preemptible instances are seen as an opportunity to increase resource utilization. In this context, the CERN OpenLab has produced a prototype implementation (called Reaper Service[6]) of a service that captures the scheduling errors that are provuced by the scheduler whenever there is a failure. Once the error has been produced, this service checks if there are preemptible instances available to be terminated so that enough free resources are available. This service, although similar in concept to our solution, requires that there is a previous failure in the scheduler, whereas the solution that we will propose in this work does not require a scheduling failure to terminate the required preemptible instances. It is expected that this extra step will incur in a higher scheduling latency, as we will show in the 6.

2.1. Scheduling in the existing cloud management frameworks

Generally speaking, existing Cloud Management Frameworks (CMFs) do not implement full-fledged queuing mechanism as other computing models do (like the Grid or traditional batch systems). Clouds are normally more focused on the rapid scaling of the resources rather than in batch processing, where systems are governed by queuing systems [34]. The default scheduling strategies in the current CMFs are mostly based on the immediate allocation or resources following a first-come, first-served basis. The cloud schedulers provision them when requested, or they are not provisioned at all (except in some CMFs that implement a FIFO queuing mechanism) [35].

However, some users require for a queuing system – or some more advanced features like advance reservations – for running virtual machines. In those cases, there are some external services such as Haizea [36] for OpenNebula or Blazar[7] for OpenStack. Those systems lay between the CMF and the users, intercepting their requests and interacting with the cloud system on their behalf, implementing the required functionality.

Besides simplistic scheduling policies like first-fit or random chance node selection [35], current CMF implement a scheduling algorithm that is based on a rank selection of hosts, as we will explain in what follows:

OpenNebula [8] uses by default a `match making` scheduler, implementing the Rank Scheduling Policy [36]. This policy first performs a filtering of the existing hosts, excluding those that do not meet the request requirements. Afterwards, the scheduler evaluates some operator defined rank expressions against the recorded information from each of the hosts so as to obtain an ordered list of nodes. Finally, the resources with a higher rank are selected to fulfill the request. OpenNebula implements a queue to hold the requests that cannot be satisfied immediately, but this queuing mechanism follows a FIFO logic, without further priority adjustment.

OpenStack [9] implements a Filter Scheduler [37], based on two separated phases. The first phase consists on the filtering of hosts that will exclude the hosts that cannot satisfy the request. This filtering follows a modular design, so that it is possible to filter out nodes based on the user request (RAM, number of vCPU), direct user input (such as instance affinity

3 http://aws.amazon.com/ec2/purchasing-options/spot-instances/.
4 https://cloud.google.com/products/compute-engine.
5 https://cloud.google.com/preemptible-vms/.
6 https://gitlab.cern.ch/ttsiouts/ReaperServicePrototype.
7 https://launchpad.net/blazar.
8 http://opennebula.org/.
9 http://www.openstack.org.

or anti-affinity) or operator configured filtering. The second phase consists on the weighing of hosts, following the same modular approach. Once the nodes are filtered and weighed, the best candidate is selected from that ordered set.

CloudStack [10] utilizes the term *allocator* to determine which host will be selected to place the new VM requested. The nodes that are used by the allocators are the ones that are able to satisfy the request.

Eucalyptus [11] implements a greedy or round robin algorithm. The former strategy uses the first node that is identified as suitable for running the VM. This algorithm exhausts a node before moving on to the next node available. On the other hand, the later schedules each request in a cyclic manner, distributing evenly the load in the long term.

Algorithm 1: Scheduling Algorithm.

```
1: function SCHEDULE REQUEST(req, H)
INPUT: req: user request
INPUT: H: all host states
2:     hosts ← [ ]                                   ▷ empty list
3:     for all h_i ∈ H do
4:         if FILTER(h_i, req) then
5:             Ω_i ← 0
6:             for all r, m in ranks do   ▷ r is a rank function, m the rank multiplier
7:                 Ω_i ← Ω_i + m_j * r_j(h_i, req)
8:             end for
9:             hosts ← hosts + (h_i, Ω_i)            ▷ append to the list
10:        end if
11:    end for
12:    return hosts
13: end function
```

All the presented scheduling algorithms share the view that the nodes are firstly filtered out – so that only those that can run the request are considered – and then ordered or ranked according to some defined rules. Generally speaking, the scheduling algorithm can be expressed as the pseudo-code in the Algorithm 1.

3. Preemptible instances design

The initial assumption for a *preemptible aware* scheduler is that the scheduler should be able to take into account two different instance types – preemptible and normal – according to the following basic rules:

- If it is a normal instance and there are no free resources for it, it must check if the termination of any running preemptible instance will leave enough space for the new instance.
  - If this is true, those instances should be terminated – according to some well defined rules – and the new VM should be scheduled into that freed node.
  - If this is not possible, then the request should continue with the failure process defined in the scheduling algorithm—it can be an error, or it can be retried after some elapsed time.
- If it is a preemptible instance, it should try to schedule it without other considerations.

[10] https://cloudstack.apache.org.
[11] https://www.eucalyptus.com/.

It should be noted that the preemptible instance selection and termination does not only depend on pure theoretical aspects, as this selection will have an influence on the resource provider revenues and the service level agreements signed with their users. Taking this into account, it is obvious that modularity and flexibility for the preemptible instance selection and is a key aspect here. For instance, an instance selection and termination algorithm that is only based on the minimization of instances terminated in order to free enough resources may not work for a provider that wish to terminate the instances that generate less revenues, event if it is needed to terminate a larger amount of instances.

Therefore, the aim of our work is not only to design an scheduling algorithm, but also to design it as a modular system so that it would be possible to create any more complex model on top of it once the initial preemptible mechanism is in place.

The most evident design approach is a retry mechanism based on two selection cycles within a scheduling loop. The scheduler will take into account a scheduling failure and then perform a second scheduling cycle after preemptible instances have been evacuated—either by the scheduler itself or by an external service. However, this two-cycle scheduling mechanism would introduce a larger scheduling latency and load in the system. This latency is something perceived negatively by the users [38] so the challenge here is how to perform this selection in a efficient way, ensuring that the selected preemptible instances are the less costly for the provider.

3.1. Preemptible-aware scheduler

Our proposed algorithm (depicted in Fig. 1) addresses the preemptible instances scheduling within one scheduling loop, without introducing a retry cycle, bur rather performing the scheduling taking into account different host states depending on the instance that is to be scheduled. This design takes into account the fact that all the algorithms described in Section 2.1 are based on two complimentary phases: filtering and raking., but adds a final phase, where the preemptible instances that need to be terminated are selected. The algorithm pseudocode is shown in 2 and will be further described in what follows.

As we already explained, the filtering phase eliminates the nodes that are not able to host the new request due to its current state – for instance, because of a lack of resources or a VM anti-affinity –, whereas the raking phase is the one in charge of assigning a rank or weight to the filtered hosts so that the best candidate is selected.

I our preemptible-aware scheduler, the filtering phase only takes into account preemptible instances when doing the filtering phase. In order to do so we propose to utilize two different states for the physical hosts:

h_f This state will take into account all the running VM inside that host, that is, the preemptible and non preemptible instances.

h_n This state will not take into account all the preemptible instances inside that host. That is, the preemptible instances running into a particular physical host are not accounted in term of consumed resources.

Whenever a new request arrives, the scheduler will use the h_f or h_n host states for the filtering phase, depending on the type of the request:

- When a normal request arrives, the scheduler will use h_n.
- When a preemptible request arrives, the scheduler will use h_f.

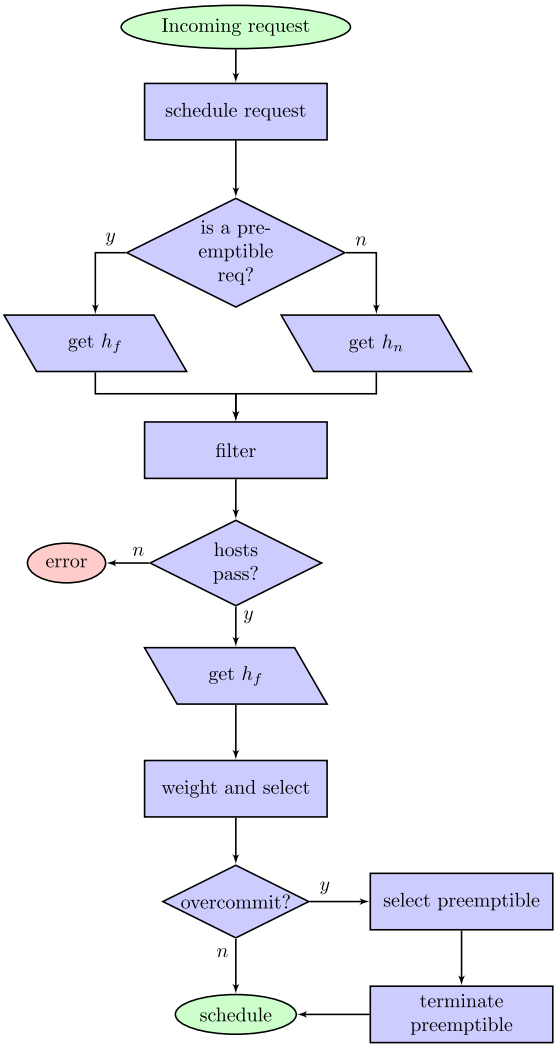


Fig. 1. Preemptible instances scheduling algorithm.

This way the scheduler ensures that a normal instance can run regardless of any preemptible instance occupying its place, as the h_n state does not account for the resources consumed by any preemptible instance running on the host. After this stage, the resulting list of hosts will contain all the hosts susceptible to host the new request, either by evacuating one or several preemptible instances or because there are enough free resources.

Once the hosts are filtered out, the ranking phase is started. However, in order to perform the correct ranking, it is needed to use the full state of the hosts, that is, h_f. This is needed as the different rank functions will require the information about the preemptible instances so as to select the best node. This list of filtered hosts may contain hosts that are able to accept the request because they have free resources and nodes that would imply the termination of one or several instances.

In order to choose the best host for scheduling a new instance new ranking functions need to be implemented, in order to prioritize the *costless* host. The simplest ranking function based on the number of preemptible instances per host is described in Algorithm 3.

This function assigns a negative value if the free resources are not enough to accommodate the request, detecting an overcommit produced by the fact that it is needed to terminate one or several

Algorithm 2: Preemptible-aware Scheduling Algorithm.

```
1: function SELECT HOSTS(req, H_f, H_n)
INPUT: req: user request
INPUT: H_f: host full-states
INPUT: H_n: host normal-instances states
2:     hosts ← [ ]                                   ▷ empty list
3:     for all h_fi, h_ni ∈ H_f, H_n do
4:         if IS PREEMPTIBLE(req) then
5:             h_i ← h_fi
6:         else
7:             h_i ← h_ni
8:         end if
9:         if FILTER(h_i, req) then
10:            Ω_i ← 0
11:            for all r, m in ranks do   ▷ r is a rank function, m the
    rank multiplier
12:                Ω_i ← Ω_i + m_j * r_j(h_fi, req)
13:            end for
14:            hosts ← hosts + (h_fi, Ω_i)           ▷ append to the list
15:        end if
16:    end for
17:    return hosts
18: end function
19: function SCHEDULE REQUEST(req, H_f, H_n)
INPUT: req: user request
INPUT: H_f: host full-states
INPUT: H_n: host normal-instances states
20:    hosts ← SELECT HOSTS(req, H_f, H_n)
21:    host ← BEST HOST(hosts)
22:    SELECT AND TERMINATE(req, host)
23:    return host
24: end function
```

Algorithm 3: Ranking function detecting overcommit of resources.

```
1: function OVERCOMMIT RANK(req, h_f)
INPUT: req: user request
INPUT: h_f: host state
2:     if req.resources > h_f.free_resources then
3:         return −1
4:     end if
5:     return 0
6: end function
```

preemptible instances. However, this basic function only establishes a naive ranking based on the termination or not of instances. In the case that it is needed to terminate various instances, this function does not establish any rank between them, so more appropriate rank functions need to be created, depending on the business model implemented by the provider. Our design takes this fact into account, allowing for modularity of these cost functions that can be applied to the raking function.

For instance, commercial providers tend to charge by complete periods of 1 h, so partial hours are not accounted. A ranking function based in this business model can be expressed as Algorithm 4, ranking hosts according to the preemptible instances running inside them and the time needed until the next complete period.

Once the ranking phase is finished, the scheduler will have built an ordered list of hosts, containing the best candidates for the new request. Once the best host selected it is still needed to select which individual preemptible instances need to be evacuated from that host, if any. Our design adds a third phase, so as to terminate the preemptible instances if needed.

Algorithm 4: Ranking function based on 1 h consumption periods.

```
1: function PERIOD RANK(req, h_f)
INPUT: req: user request
INPUT: h_f: host state
2:     weight ← 0
3:     for all instance ∈ get_instances(h_f) do
4:         if (is_spot(instance) then
5:             if (instance.run_time mod 3600) > 0 then
6:                 weight ← weight + instance.run_time mod 3600
7:             end if
8:         end if
9:     end for
10:    return −weight
11: end function
```

This last phase will perform an additional raking and selection of the candidate preemptible instances inside the selected host, so as to select the less costly for the provider. This selection leverages a similar ranking process, performed on the preemptible instances, considering all the preemptible instances combination and the costs for the provider, as shown in Algorithm 5.

Algorithm 5: Preemptible instance selection and termination.

```
1: procedure SELECT AND TERMINATE(req, h_f)
INPUT: req: user request
INPUT: h_f: host state
2:     selected_instances ← [ ]
3:     for all instances ∈ get__all_preemptible_combinations(h_f) do
4:         if ∑(instances.resources) > req.resources then
5:             if cost(instances) < cost(selected_instances)0 then
6:                 selected_instances ← instances
7:             end if
8:         end if
9:     end for
10:    TERMINATE(selected_instances)
11: end procedure
```

4. Evaluation

In the first part of this section 4.2 we will describe an implementation – done for the OpenStack Compute CMF –, in order to evaluate our proposed algorithm. We have decided to implement it on top of the OpenStack Compute software due to its modular design, that allowed us to easily plug our modified modules without requiring significant modifications to the code core.

Afterwards we will perform two different evaluations. On the one hand we will assess the algorithm correctness, ensuring that the most desirable instances are selected according to the configured weighers (Section 4.4). On the other hand we will examine the performance of the proposed algorithm when compared with the default scheduling mechanism (Section 4.5).

4.1. OpenStack compute filter scheduler

The OpenStack Compute scheduler is called Filter Scheduler and, as already described in Section 2, it is a rank scheduler, implementing two different phases: filtering and weighting.

Filtering The first step is the filtering phase. The scheduler applies a concatenation of filter functions to the initial set of available hosts, based on the host properties and state – e.g. free RAM or free CPU number – user input – e.g. affinity or anti-affinity with other instances – and resource provider defined configuration. When the filtering process has concluded, all the hosts in the final set are able to satisfy the user request.

Weighing Once the filtering phase returns a list of suitable hosts, the weighting stage starts so that the best host – according to the defined configuration – is selected. The scheduler will apply all hosts the same set of weigher functions $w_i(h)$, taking into account each host state h. Those weigher functions will return a value considering the characteristics of the host received as input parameter, therefore, total weight Ω for a node h is calculated as follows:

$$\Omega = \sum^{n} m_i \cdot \mathrm{N}(w_i(h))$$

Where m_i is the multiplier for a weigher function, $\mathrm{N}(w_i(h))$ is the normalized weight between [0, 1] calculated via a rescaling like:

$$\mathrm{N}(w_i(h)) = \frac{w_i(h) - \min W}{\max W - \min W}$$

where $w_i(h)$ is the weight function, and $\min W$, $\max W$ are the minimum and maximum values that the weigher has assigned for the set of weighted hosts. This way, the final weight before applying the multiplication factor will be always in the range [0, 1].

After these two phases have ended, the scheduler has a set of hosts, ordered according to the weights assigned to them, thus it will assign the request to the host with the maximum weight. If several nodes have the same weight, the final host will be randomly selected from that set.

4.2. Implementation evaluation

We have extended the Filter Scheduler algorithm with the functionality described in Algorithm 6. We have also implemented the ranking functions described in Algorithms 3 and 4 as weighers, using the OpenStack terminology.

Moreover, the Filter Scheduler has been also modified so as to introduce the additional select and termination phase (Algorithm 5). This phase has been implemented following the same modular approach as the OpenStack weighting modules, allowing to define and implement additional cost modules to determine which instances are to be selected for termination.

As for the cost functions, we have implemented a module following Algorithm 4. This cost function assumes that customers are charged by periods of 1 h, therefore it prioritizes the termination of Spot Instances with the lower partial-hour consumption (i.e. if we consider instances with 120 min, 119 min and 61 min of duration, the instance with 120 min will be terminated).

This development has been done on the OpenStack Newton version,[12] and was deployed on the infrastructure that we describe in Section 4.3.

4.3. Configurations

In order to evaluate our algorithm proposal we have set up a dedicated test infrastructure comprising a set of 26 identical IBM HS21 blade servers, with the characteristics described in Table 1. All the nodes had an identical base installation, based on an Ubuntu Server 16.04 LTS, running the Linux 3.8.0 Kernel, where we have deployed OpenStack Compute as the Cloud Management Framework. The system architecture is as follows:

[12] https://github.com/indigo-dc/opie.

Algorithm 6: Preemptible Instances Scheduling Algorithm.

```
1: function SELECT DESTINATIONS(req)
INPUT: req: user request
2:     host ← SCHEDULE(req)
3:     if host is overcommitted then
4:         SELECT AND TERMINATE(req,host)
5:     end if
6:     return host
7: end function

8: function SCHEDULE(req)
INPUT: req: user request
9:     H_f ← host_states(full)
10:    H_p ← host_states(partial)
11:    if is_spot(req) then
12:        H_filtered ← filter(req, H_f)
13:    else
14:        H_filtered ← filter(req, H_p)
15:    end if
16:    H_weighted ← weight(req, H_f)
17:    best ← select_best(H_weighted)
18:    return best
19: end function

20: procedure SELECT AND TERMINATE(req, h_f)
INPUT: req: user request
INPUT: h_f: host state
21:    selected_instances ← [ ]
22:    for all instances ∈ get_all_preemptible_combinations(h_f) do
23:        if ∑ instances.resources > req.resources then
24:            if cost(instances) < cost(selected_instances)0 then
25:                selected_instances ← instances
26:            end if
27:        end if
28:    end for
29:    TERMINATE(selected_instances)
30: end procedure
```

Table 1
Test node characteristics.

CPU	2 x Intel Xeon Quad Core E5345 2.33 GHz
RAM	16 GB
Disk	140 GB, 10 000 rpm hard disk
Network	1 Gbit Ethernet

- A Head node hosting all the required services to manage the cloud test infrastructure, that is:
  - The OpenStack Compute API.
  - The OpenStack Compute Scheduler service.
  - The OpenStack Compute Conductor service.
  - The OpenStack Identity Service (Keystone)
  - A MariaDB 10.1.0 server.
  - A RabbitMQ 3.5.7 server.
- An Image Catalog running the OpenStack Image Service (Glance) serving images from its local disk.
- 24 Compute Nodes running OpenStack Compute, hosting the spawned instances.

The network setup of the testbed consists on two 10 Gbit Ethernet switches, interconnected with a 10 Gbit Ethernet link. All the hosts are evenly connected to these switches using a 1 Gbit Ethernet connection.

Table 2
Configured VM sizes.

Name	vCPUs	RAM (MB)	Disk (GB)
small	1	2000	20
medium	2	4000	40
large	4	8000	80

Table 3
Test-1, preemptible instances evaluation using the same VM size. The label marked with ([a]) indicate the terminated instance. Time is expressed in minutes.

Host	Instances		Preemptible Instances	
	ID	Time	ID	Time
host-A	A1	272	AP1	96
	A2	172	AP2	207
host-B	B1	136	BP1([a])	71
	B2	200	BP2	91
host-C	C1	97	CP1	210
	C2	275	CP2	215
host-D	D1	16	DP1	85
			DP2	199
			DP3	152

[a]Selected instance.

We have considered the VM sizes described in Table 2, based on the default set of sizes existing in a default OpenStack installation.

4.4. Algorithm evaluation

The purpose of this evaluation is to ensure that the proposed algorithm is working as expected, so that:

- The scheduler is able to deliver the resources for a normal request, by terminating one or several preemptible instances when there are not enough free idle resources.
- The scheduler selects the best preemptible instance for termination, according to the configured policies by means of the scheduler weighers.

4.4.1. Scheduling using same virtual machine sizes

For the first batch of tests, we have considered same size instances, to evaluate if the proposed algorithm chooses the best physical host and selects the best preemptible instance for termination. We generated requests for both preemptible and normal instances – chosen randomly –, of random duration between 10 min and 300 min, using an exponential distribution [39] until the first scheduling failure for a normal instance was detected.

The compute nodes used have 16 Gbit of RAM and eight CPUs, as already described. The VM size requested was the *medium* one, according to Table 2, therefore each compute node could host up to four VMs.

We executed these requests and monitored the infrastructure until the first scheduling failure for a normal instance took place, thus the preemptible instance termination mechanism was triggered. At that moment we took a snapshot of the nodes statuses, as shown in Tables 3 and 4. These tables depict the status for each of the physical hosts, as well as the running time for each of the instances that were running at that point. The shaded cells represents the preemptible instance that was terminated to free the resources for the incoming non preemptible request.

Considering that the preemptible instance selection was done according to Algorithm 5 using the cost function in Algorithm 4, the chosen instance has to be the one with the lowest partial-hour period. In Table 3 this is the instance marked with ([a]): *BP1*. By chance, it corresponds with the preemptible instance with the lowest run time.

Table 4
Test-2, preemptible instances evaluation using the same VM size. The label marked with ([a]) indicate the terminated instance. Time is expressed in minutes.

Host	Instances		Preemptible Instances	
	ID	Time	ID	Time
host-A			AP1	247
			AP2	463
			AP3	403
			AP4	410
host-B	B1	388	BP1	344
	B2	103	BP2	476
host-C	C1	481	CP1 ([a])	181
	C2	177	CP2	160
host-D	D1	173	DP1	384
			DP2	168
			DP3	232

[a]Selected instance.

Table 5
Test-3, preemptible instances evaluation using different VM sizes. The labels marked with ([a]) indicate the terminated instances. Time is expressed in minutes. S, M, L stand for small, medium and large respectively.

Host	Instances			Preemptible Instances		
	ID	Time	Size	ID	Time	Size
host-A				AP1	298	L
				AP2 ([a])	278	M
				AP3 ([a])	190	S
				AP4 ([a])	187	S
host-B	B1	494	L	BP1	178	L
host-C				CP1	297	L
				CP2	296	M
				CP3	296	S
host-D	D1	176	M			
	D2	200	M			
	D3	116	L			

[a]Selected instances.

Table 4 shows a different test execution under the same conditions and constraints. Again, the selected instance has to be the one with the lowest partial-hour period. In Table 4 this corresponds to the instance marked again with ([a]): *CP1*, as its remainder is 1 min. In this case this is not the preemptible instance with the lowest run time (being it *CP2*).

4.4.2. Scheduling using different virtual machine sizes

For the second batch of tests we requested instances using different sizes, always following the sizes in Table 2. Table 5 depicts the testbed status when a request for a *large* VM caused the termination of the instances marked with ([a]): *AP2*, *AP3* and *AP4*. In this case, the scheduler decided that the termination of these three instances caused a smaller impact on the provider, as the sum of their 1 h remainders (55) was lower than any of the other possibilities (58 for *BP1*, 57 for *CP1*, 112 for *CP2* and *CP3*).

Table 6 shows a different test execution under the same conditions and constraints. In this case, the preemptible instance termination was triggered by a new VM request of size *medium* and the selected instance was the one marked with ([a]): *BP3*, as *host-B* will have enough free space just by terminating one instance.

4.5. Performance evaluation

As we have already said in Section 3, we have focused on designing an algorithm that does not introduce a significant latency in the system. This latency will introduce a larger delay when delivering the requested resources to the end users, something that is not desirable by any resource provider [4].

In order to evaluate the performance of our proposed algorithm we have done a comparison with the default, unmodified OpenStack Filter Scheduler [37]. Moreover, for the sake of comparison, we have implemented a scheduler based on a *retry loop* as well. This scheduler is a simple modification of the original Filter Scheduler that performs a normal scheduling loop, and if there is a scheduling failure for a normal instance, it performs a second pass taking into account the existing preemptible instances. The preemptible instance selection and termination mechanisms remain the same. Lastly, we have performed the same test with the Reaper prototype service [40] already discussed in Section 2.

Table 6
Test-4, preemptible instances evaluation using different VM sizes. The labels marked with ([a]) indicate the terminated instances. Time is expressed in minutes. S, M, L stand for small, medium and large respectively.

Host	Instances			Preemptible Instances		
	ID	Time	Size	ID	Time	Size
	A1	234	L	AP1	172	M
	A2	122	M			
host-B				BP1	272	L
				BP2	212	M
				BP3 ([a])	380	S
host-C	C1	182	S			
	C2	120	M			
	C3	116	L			
host-D				DP1	232	L
				DP2	213	S
				DP3	324	M
				DP4	314	S

[a]Selected instances.

We have scheduled 130 Virtual Machines of the same size on our test infrastructure and we have recorded the timings for the scheduling function, thus calculating the means and standard deviation for each of the following scenarios:

- Using the original, unmodified OpenStack Filter scheduler with an empty infrastructure.
- Using the preemptible instances Filter Scheduler, the retry scheduler and the Reaper prototype service:
  - Requesting normal instances with an empty infrastructure.
  - Requesting preemptible instances with an empty infrastructure.
  - Requesting normal instances with a saturated infrastructure, thus implying the termination of a preemptible instance each time a request is performed.

We have then collected the scheduling calls timings and we have calculated the means and deviations for each scenario, as shown in Fig. 2. Numbers in these scenarios are quite low, since the infrastructure is a small testbed, but these numbers are expected to become larger as the infrastructure grows in size.

As it can be seen in the aforementioned Fig. 2, our solution introduces a delay in the scheduling calls, as we need to calculate additional host states (we hold two different states for each node) and we need to select a preemptible instance for termination (in case it is needed). In the case of the retry scheduler, this delay does not exists and numbers are similar to the original scheduler. However, when it is needed to trigger the termination of a preemptible instance, having a retry mechanism (thus executing the same scheduling call two times) introduces a significantly larger penalty when compared to our proposed solution. Also, the numbers for the Reaper service are higher, as it requires that the scheduler raises an exception that is then handled by the Reaper prototype. We consider that the latency that we are introducing is within an acceptable range, therefore not impacting significantly the scheduler performance.

During the execution of these tests we had to face several integration problems due to incompatible changes between versions,

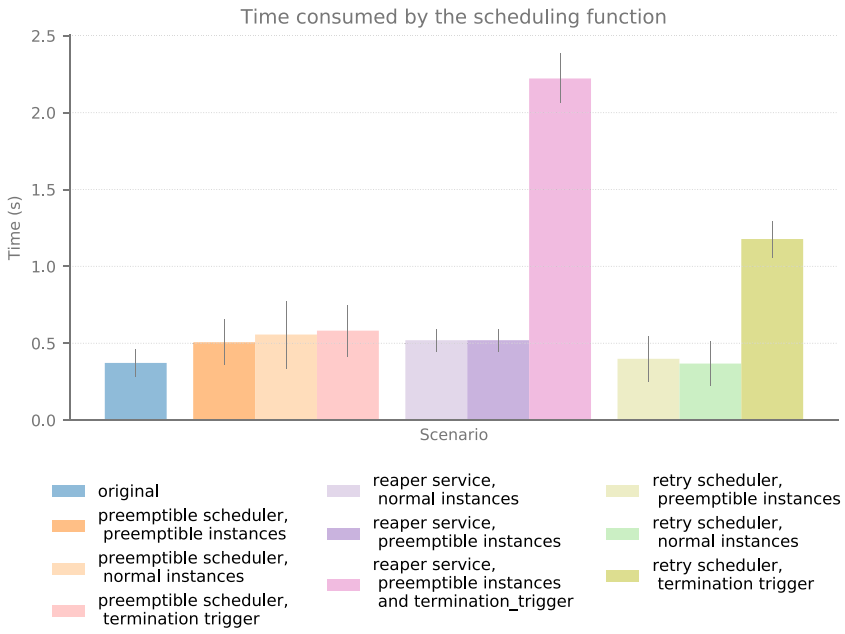


Fig. 2. Comparison of the time consumed by the different scheduling options in different scenarios. Error bars represent the standard deviation. *Original* refers to the unmodified OpenStack Filter Scheduler [37], *retry scheduler* refers to the *original* scheduler modified to perform an additional scheduling cycle, *reaper service* refers to the usage of the Reaper Prototype [40]. *Preemptible scheduler* refers to this work.

lack of documentation or clear integration recipes that prevented us from performing further tests with other frameworks described Section 2. We consider that all of these issues are a consequence of the rapid development pace of cloud management frameworks and not a failure on the existing tools.

5. Exploitation and integration in existing infrastructures

The functionality introduced by the preemptible instances model that we have described in this work can be exploited not only within a cloud resource provider, but it can also be leveraged on more complex hybrid infrastructures.

5.1. High performance computing integration

One can find in the literature several exercises of integration of hybrid infrastructures, integrating cloud resources, commercial or private, with High Performance Computing (HPC) resources. Those efforts focus on outbursting resources from the cloud, when the HPC system does not provide enough resources to solve a particular problem [41].

On-demand provisioning using cloud resources when the batch system of the HPC is full is certainly a viable option to expand the capabilities of a HPC center for serial batch processing.

We focus however in the complementary approach, this is, using HPC resources to provide cloud resources capability, so as to complement existing distributed infrastructures. Obviously HPC systems are oriented to batch processing of highly coupled (parallel) jobs. The question here is optimizing resource utilization when the HPC batch system has empty slots.

If we backfill the empty slots of a HPC system with cloud jobs, and a new regular batch job arrives from the HPC users, the cloud jobs occupying the slots needed by the newly arrived batch job should be terminated immediately, so as to not disturb regular work. Therefore such cloud jobs should be submitted as Spot Instances.

Enabling HPC systems to process other jobs during periods in which the load of the HPC mainframe is low, appears as an attractive possibility from the point of view of resource optimization. However the practical implementation of such idea would need to be compatible with both, the HPC usage model, and the cloud usage model.

In HPC systems users login via ssh to a frontend. At the frontend the user has the tools to submit jobs. The scheduling of HPC jobs is done using a regular batch systems software (such as SLURM, SGE, etc...).

HPC systems are typically running MPI parallel jobs as well using specialized hardware interconnects such as Infiniband.

Let us imagine a situation in which the load of the HPC system is low. One can instruct the scheduler of the batch system to allow cloud jobs to HPC system occupying those slots not allocated by the regular batch allocation.

In order to be as less disrupting as possible the best option is that the cloud jobs arrive as preemptible instances as described through this paper. When a batch job arrives to the HPC system, this job should be immediately scheduled and executed. Therefore the scheduler should be able to perform the following steps:

- Allocate resources for the job that just arrived to the batch queue system
- Identify the cloud jobs that are occupying those resources, and stop them.
- Dispatch the batch job.

In the case of parallel jobs the scheduling decision may depend on many factors like the topology of the network requested, or the affinity of the processes at the core/CPU level. In any case parallel jobs using heavily the low latency interconnect should not share nodes with any other job.

5.2. High throughput computing integration

Existing High Throughput Computing Infrastructures, like the service offered by EGI,[13] could benefit from a cloud providers offering preemptible instances. It has been shown that cloud resources and IaaS offerings can be used to run HTC tasks [42] in a pull mode,

[13] https://www.egi.eu/services/high-throughput-compute/.

where cloud instances are started in a way that they are able to pull computing tasks from a central location (for example using a distributed batch system like HTCondor).

However, sites are reluctant to offer large amounts of resources to be used in this mode due to the lack of a fixed duration for cloud instances. In this context, federated cloud e-Infrastructures like the EGI Federated Cloud [43], could benefit from resource providers offering preemptible instances. Users could populate idle resources with preemptible instances pulling their HTC tasks, whereas interactive and normal IaaS users will not be impacted negatively, as they will get the requests satisfied. In this way, large amounts of cloud computing power could be offered to the European research community.

6. Conclusions

In this work we have proposed a preemptible instance scheduling design that does not modify substantially the existing scheduling algorithms, but rather enhances them. The modular rank and cost mechanisms allows the definition and implementation of any resource provider defined policy by means of additional pluggable rankers. Our proposal and implementation enables all kind of service providers – whose infrastructure is managed by open source middleware such as OpenStack – to offer a new access model based on preemptible instances, with a functionality similar to the one offered by the major commercial providers.

We have checked for the algorithm correctness when selecting the preemptible instances for termination. The results yield that the algorithm behaves as expected. Moreover we have compared the scheduling performance with regards equivalent default scheduler, obtaining similar results, thus ensuring that the scheduler performance is not significantly impacted.

Among the existing solutions for implementing preemptible instances there are some complementarities that are worth exploring. The Reaper service could benefit from the scheduling mechanism that is being presented here, so that instead of waiting for a scheduling failure to react, the preemptible instance termination could be performed directly by the scheduler if it detects that it is needed to free resources occupied by the preemptible instances. This way the scheduling time could be reduced significantly.

This implementation allows to apply more complex policies on top of the preemptible instances, like instance termination based on price fluctuations (that is, implementing a preemptible instance stock market), preemptible instance migration so as to consolidate them or proactive instance termination to maximize the provider's revenues by not delivering computing power at no cost to the users.

Acknowledgments

The authors acknowledge the financial support from the European Commission Horizon 2020 via INDIGO-DataCloud project (grant number 653549) and EGI-ENGAGE (grant number 654142) and the Ministry of Economy and Competitiveness for the support through the National Plan under contract number FPA2013-40715-P.

The authors want also to thank the IFCA Advanced Computing and e-Science Group.

References

[1] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, J. Good, On the use of cloud computing for scientific workflows, in: 2008 IEEE Fourth International Conference on eScience, IEEE, 2008, pp. 640–645, http://dx.doi.org/10.1109/eScience.2008.167.

[2] A. Iosup, S. Ostermann, M. Yigitbasi, Performance analysis of cloud computing services for many-tasks scientific computing, IEEE Trans. Parallel Distrib. Syst. 22 (2011) 931–945.

[3] J.-S. Vöckler, G. Juve, E. Deelman, M. Rynge, B. Berriman, Experiences using cloud computing for a scientific workflow application, Condor 300 (1–2) (2011) 15–24, http://dx.doi.org/10.1145/1996109.1996114.

[4] Á. López García, E. Fernández-del Castillo, P. Orviz Fernández, I. Campos Plasencia, J. Marco de Lucas, Resource provisioning in science clouds: Requirements and challenges, in: Software: Practice and Experience, 2017, http://dx.doi.org/10.1002/spe.2544, n/a–n/a.

[5] G. Juve, E. Deelman, Resource provisioning options for large-scale scientific workflows, in: 2008 IEEE Fourth International Conference on eScience, 2008, pp. 608–613, http://dx.doi.org/10.1109/eScience.2008.160.

[6] S. Singh, I. Chana, A survey on resource scheduling in cloud computing: Issues and challenges, J. Grid Comput. 14 (2) (2016) 217–264.

[7] D. Salomoni, I. Campos, L. Gaido, J.M. de Lucas, P. Solagna, J. Gomes, L. Matyska, P. Fuhrman, M. Hardt, G. Donvito, L. Dutka, M. Plociennik, R. Barbera, I. Blanquer, A. Ceccanti, E. Cetinic, M. David, C. Duma, A. López-García, G. Moltó, P. Orviz, Z. Sustr, M. Viljoen, F. Aguilar, L. Alves, M. Antonacci, L.A. Antonelli, S. Bagnasco, A.M.J.J. Bonvin, R. Bruno, Y. Chen, A. Costa, D. Davidovic, B. Ertl, M. Fargetta, S. Fiore, S. Gallozzi, Z. Kurkcuoglu, L. Lloret, J. Martins, A. Nuzzo, P. Nassisi, C. Palazzo, J. Pina, E. Sciacca, D. Spiga, M. Tangaro, M. Urbaniak, S. Vallero, B. Wegh, V. Zaccolo, F. Zambelli, T. Zok, Indigo-datacloud: A platform to facilitate seamless access to e-infrastructures, J. Grid Comput. 16 (3) (2018) 381–408, http://dx.doi.org/10.1007/s10723-018-9453-3.

[8] A. Lopez Garcia, L. Zangrando, M. Sgaravatto, V. Llorens, S. Vallero, V. Zaccolo, S. Bagnasco, S. Taneja, S.D. Pra, D. Salomoni, G. Donvito, A.L. Garcia, L. Zangrando, M. Sgaravatto, V. Llorens, S. Vallero, V. Zaccolo, S. Bagnasco, S. Taneja, S.D. Pra, D. Salomoni, G. Donvito, Improved cloud resource allocation: How INDIGO-Datacloud is overcoming the current limitations in cloud schedulers, J. Phys. Conf. Ser. 898 (9) (2017) 92010, http://dx.doi.org/10.1088/1742-6596/898/9/092010, arXiv:1707.06403.

[9] L. Ramakrishnan, P.T. Zbiegel, S. Campbell, R. Bradshaw, R.S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, A. Liu, Magellan: experiences from a science cloud, in: Proceedings of the 2nd International Workshop on Scientific Cloud Computing, ScienceCloud '11, San Jose, California, USA, ACM, New York, NY, USA, 2011, pp. 49–58, http://dx.doi.org/10.1145/1996109.1996119.

[10] P. Marshall, K. Keahey, T. Freeman, Improving utilization of infrastructure clouds, in: Proceedings - 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011, 2011, pp. 205–214, http://dx.doi.org/10.1109/CCGrid.2011.56.

[11] A.L. Garcia, Openstack preemptible instances extension, 2018, http://dx.doi.org/10.5281/zenodo.1544248.

[12] B. Jennings, R. Stadler, Resource management in clouds: Survey and research challenges, J. Netw. Syst. Manage. 23 (3) (2015) 567–619, http://dx.doi.org/10.1007/s10922-014-9307-7.

[13] M.D. De Assunção, C.H. Cardonha, M.A. Netto, R.L. Cunha, Impact of user patience on auto-scaling resource capacity for cloud services, Future Gener. Comput. Syst. 55 (2016) 41–50, http://dx.doi.org/10.1016/j.future.2015.09.001.

[14] A. Andrzejak, D. Kondo, S.Y.S. Yi, Decision model for cloud computing under SLA constraints, modeling, analysis, in: Simulation of Computer and Telecommunication Systems (MASCOTS), in: 2010 IEEE International Symposium on http://dx.doi.org/10.1109/MASCOTS.2010.34.

[15] S. Yi, D. Kondo, A. Andrzejak, Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud, in: Proceedings - 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD, 2010, pp. 236–243, http://dx.doi.org/10.1109/CLOUD.2010.35.

[16] S. Yi, A. Andrzejak, D. Kondo, Monetary cost-aware checkpointing and migration on amazon cloud spot instances, IEEE Trans. Serv. Comput. 5 (4) (2012) 512–524, http://dx.doi.org/10.1109/TSC.2011.44.

[17] S. Khatua, N. Mukherjee, Application-centric resource provisioning for amazon EC2 spot instances, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) LNCS, vol. 8097, 2013, pp. 267–278, http://dx.doi.org/10.1007/978-3-642-40047-6_29, arXiv:12111279v1.

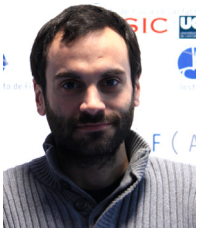
[18] D. Jung, S. Chin, K. Chung, H. Yu, J. Gil, An efficient checkpointing scheme using price history of spot instances in cloud computing environment, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) LNCS, vol. 6985, 2011, pp. 185–200, http://dx.doi.org/10.1007/978-3-642-24403-2_16.

[19] W. Voorsluys, S.K. Garg, R. Buyya, Provisioning spot market cloud resources to create cost-effective virtual clusters, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) LNCS, vol. 7016, 2011, pp. 395–408, http://dx.doi.org/10.1007/978-3-642-24650-0_34, arXiv:1110.5972, (PART 1).

[20] W. Voorsluys, R. Buyya, Reliable provisioning of spot instances for compute-intensive applications, in: Proceedings - International Conference on Advanced Information Networking and Applications, AINA, 2012, pp. 542–549, http://dx.doi.org/10.1109/AINA.2012.106, arXiv:1110.5969.

[21] D. Jung, J. Lim, H. Yu, J. Gil, E. Lee, A workflow scheduling technique for task distribution in spot instance-based cloud, in: Ubiquitous Information Technologies and Applications, Springer, 2014, pp. 409–416.

[22] N. Jain, I. Menache, O. Shamir, On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud, in: 11th International Conference on Autonomic Computing (ICAC 14).
[23] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, See spot run: Using spot instances for mapreduce workflows, in: HotCloud 2010, 2012, pp. 1–7.
[24] H. Liu, Cutting mapreduce cost with spot market, in: USENIX HotCloud'11, 2011, p. 5.
[25] Y. Song, M. Zafer, K.-W. Lee, Optimal bidding in spot instance market, in: 2012 Proceedings IEEE, INFOCOM, 2012, pp. 190–198, http://dx.doi.org/10.1109/INFCOM.2012.6195567.
[26] K. Sowmya, R.P. Sundarraj, Strategic bidding for cloud resources under dynamic pricing schemes, in: Cloud and Services Computing (ISCOS), in: 2012 International Symposium on, 2012, pp. 25–30, http://dx.doi.org/10.1109/ISCOS.2012.28.
[27] S.-Y. Noh, S.C. Timm, H. Jang, Vcluster: A framework for auto scalable virtual cluster system in heterogeneous clouds, Clust. Comput. 17 (3) (2013) 741–749, http://dx.doi.org/10.1007/s10586-013-0292-5.
[28] A. Nadjaran Toosi, F. Khodadadi, R. Buyya, SipaaS: Spot instance pricing as a service framework and its implementation in openstack, in: Concurrency and Computation: Practice and Experience, 2015, pp. 3672–3690, http://dx.doi.org/10.1002/cpe.3749.
[29] A.N. Toosi, K. Vanmechelen, F. Khodadadi, R. Buyya, An auction mechanism for cloud spot markets, ACM Trans. Auton. Adapt. Syst.(TAAS) 11 (1) (2016) 2.
[30] M. Carvalho, D.A. Menascé, F. Brasileiro, Capacity planning for IaaS cloud providers offering multiple service classes, Future Gener. Comput. Syst. 77 (2017) 97–111, http://dx.doi.org/10.1016/j.future.2017.07.019.
[31] M. Carvalho, W. Cirne, F. Brasileiro, J. Wilkes, Long-term SLOs for reclaimed cloud computing resources, in: Proceedings of the ACM Symposium on Cloud Computing - SOCC'14, 2014, pp. 1–13, http://dx.doi.org/10.1145/2670979.2670999.
[32] M. Carvalho, D. Menasce, F. Brasileiro, Prediction-based admission control for IaaS clouds with multiple service classes, in: Proceedings - IEEE 7th International Conference on Cloud Computing Technology and Science, in: CloudCom 2015, 2016, pp. 82–90, http://dx.doi.org/10.1109/CloudCom.2015.16, (2).
[33] OpenStack Scientific Working Group, The Crossroads of Cloud and HPC : OpenStack for Scientific Research, CreateSpace Independent Publishing Platform, 2017.
[34] I. Foster, Y. Zhao, I. Raicu, S. Lu, Cloud computing and grid computing 360-degree compared, in: Grid Computing Environments Workshop, 2008, in: GCE'08, 2008, pp. 1–10, URL http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=4738445.
[35] R. Buyya, J. Broberg, A.M. Goscinski, B. Rochwerger, C. Vázquez, D. Breitgand, D. Hadas, M. Villari, P. Massonet, E. Levy, A. Galis, I.M. Llorente, R.S. Montero, Y. Wolfsthal, K. Nagin, L. Larsson, F. Galán, Cloud Computing: Prin-Ciples and Paradigms, vol. 87, John Wiley & Sons, 2010, http://dx.doi.org/10.1002/9780470940105.ch15.
[36] B. Sotomayor, R.S. Montero, I.M. Llorente, I. Foster, Virtual infrastructure management in private and hybrid clouds, IEEE Internet Comput. 13 (2009) 14–22, http://dx.doi.org/10.1109/MIC.2009.119.
[37] O. Litvinski, A. Gherbi, Experimental evaluation of openstack compute scheduler, Procedia Comput. Sci. 19 (Ant) (2013) 116–123, http://dx.doi.org/10.1016/j.procs.2013.06.020.
[38] Á. López García, E. Fernández-del Castillo, Efficient image deployment in cloud environments, J. Netw. Comput. Appl. 63 (2016) 140–149, http://dx.doi.org/10.1016/j.jnca.2015.10.015.
[39] D.E. Knuth, The Art of Computer Programming, vol. 2, Addison-Wesley, 1981.
[40] T. Tsioutsias, Reaper service prototype, 2018, URL https://gitlab.cern.ch/ttsiouts/ReaperServicePrototype/.
[41] M. Ben Belgacem, B. Chopard, A hybrid HPC/cloud distributed infrastructure: Coupling EC2 cloud resources with HPC clusters to run large tightly coupled multiscale applications, Future Gener. Comput. Syst. 42 (2015) 11–21, http://dx.doi.org/10.1016/j.future.2014.08.003.
[42] A. McNab, F. Stagni, M.U. Garcia, Running jobs in the vacuum, J. Phys. Conf. Ser. 513 (3) (2014) 32065, http://dx.doi.org/10.1088/1742-6596/513/3/032065.
[43] E. Fernández-del Castillo, D. Scardaci, Á. López García, The EGI federated cloud e-infrastructure, Procedia Comput. Sci. 68 (2015) 196–205, http://dx.doi.org/10.1016/j.procs.2015.09.235.



Dr. Alvaro Lopez Garcia is a research associate at CSIC. He holds a Ph.D. in Science, Technology and Computing from the University of Cantabria (UC). He was a visiting researcher at the IN2P3/CNRS Computing Center in Lyon, France and a research associate at the Italian National Institute for Nuclear Physics (INFN). He is also an assistant professor at the UC, teaching several Computer's Architecture subjects, as well as professor at the official Master degree in Data Science of the Universidad Internacional Mendendez Pelayo (UIMP). He has taken part in several national and European projects such as EGEE-II/III, Int.Eu.Grid, EUFORIA, EGI-InSPIRE, EGI-Engage (task leader for the Federated Cloud JRA) and INDIGO-DataCloud (task leader for the Cloud Computing Virtualization JRA). He is currently co-coordinating the DEEP-Hybrid-DataCloud H2020 project and participating in the EOSC-Hub and AARC-II projects. In the last years he has been working on the adoption of the cloud by scientific datacenters. He is an individual member of the OpenStack foundation, being an Active Technical Contributor for several development cycles.

Dr. Enol Fernández del Castillo is a CSIC researcher, based at the IFCA in Santander (Spain), where he joined in 2009. Since 2003, he has been involved in EU funded grid-computing projects. He holds a Ph.D. in Computer Engineering from Universidad Autonoma de Barcelona (UAB) where he developed a grid scheduler for running interactive and parallel jobs. He is currently part of the software provisioning team for the EGI.eu infrastructure and a member of EGI.eu Federated Cloud. He is involved in the IBERCLOUD initiative for setting up a cloud for scientific users of Spanish and Portuguese NGIs (IBERGRID).

Dr. Isabel Campos Plasencia is a researcher at CSIC since 2008. She is the Director of the Spanish NGI, representative of Spain in the EGI Council. She holds a Ph.D. in Theoretical Physics, and has held positions as research associate at DESY, Brookhaven National Laboratory, and at the Leibniz Computer Center in Munich. She has a wide experience in international collaborations oriented to distributed computing technology. She has over 45 publications in peer-reviewed journals and has participated or presented over 72 communications to international conferences.