Towards Designing Cost-Optimal Policies to Utilize IaaS Clouds with Online Learning

Xiaohu Wu, Patrick Loiseau, and Esa Hyytiä

Abstract—Many businesses possess a small infrastructure that they can use for their computing tasks, but also often buy extra computing resources from clouds. Cloud vendors such as Amazon EC2 offer two types of purchase options: on-demand and spot instances. As tenants have limited budgets to satisfy their computing needs, it is crucial for them to determine how to purchase different options and utilize them (in addition to possible self-owned instances) in a cost-effective manner while respecting their response-time targets. In this paper, we propose a framework to design policies to allocate self-owned, on-demand and spot instances to arriving jobs. In particular, we propose a near-optimal policy to determine the number of self-owned instances and an optimal policy to determine the number of on-demand instances to buy and the number of spot instances to bid for at each time unit. Our policies rely on a small number of parameters and we use an online learning technique to infer their optimal values. Through numerical simulations, we show the effectiveness of our proposed policies, in particular that they achieve a cost reduction of up to 64.51% when spot and on-demand instances are considered and of up to 43.74% when self-owned instances are considered, compared to previously proposed or intuitive policies.

Index Terms—On-demand instances, spot instances, cost efficiency, online learning.

1 Introduction

Infrastructure as a Service (IaaS) holds exciting potential of elastically scaling users' computation capacity up and down to match their time-varying demand. This eliminates the users' need of purchasing servers to satisfy their peak demand, without causing an unacceptable latency. The global cloud IaaS market grew to $34.6 billion in 2017, and is projected to increase to $71.6 billion in 2020 [1]. Main IaaS service providers include Amazon, Microsoft, Google, etc. Amazon is the most popular one and represents 51.8% of the global market share in 2017; here, two typical purchase options are on-demand and spot instances (i.e., virtual machines). Recently, the issue of cost-effectively utilizing spot and on-demand instances has received significant attention [2].

On-demand instances are always available at a fixed price and tenants[1] pay only for the period in which instances are consumed at an hourly rate. Users can also bid a price for spot instances and successfully get them only if their bid is above the spot price. However, spot instances will get lost once the spot price becomes higher than their bid. Here, spot prices usually vary unpredictably over time and users will be charged the spot prices for their use. Compared to on-demand instances, spot instances can reduce the cost by up to 50-90% [3]. Users that purchase cloud instances may also have their own instances, referred to as self-owned instances, which can be used to process jobs but are insufficient at times (hence the need to purchase extra IaaS instances). They may also not have any self-owned instances (e.g., in the case of startups) and therefore need to buy from the cloud all necessary computing resources.

Tenants' jobs arrive over time. We focus on processing a type of embarrassingly parallel workloads/jobs [5], [6]. Each job can be separated into a large number of small tasks. These tasks are independent and can be executed on multiple machines simultaneously. Completing a job means completing all its tasks and the maximum completion time of all tasks is the job's completion time. This type of jobs accounts for a significant proportion in cloud market; examples include 3D video rendering, BLAST searches, data cleaning and pre-processing. Such job is also called malleable job [9], [10], [11] and it has a parallelism bound specifying the maximum number of instances that it can utilize simultaneously. Each job also has constraint on timing, i.e., a deadline by which to complete all tasks of a job. Subject to the parallelism constraint, an arriving job will be allocated instances of different types (self-owned, on-demand and spot) and the allocation can be updated at most once every hour (since billing is done per hour). Our problem is then to find an allocation that minimizes cost while satisfying the deadline constraint.

Challenges. In this paper, we make a natural assumption that the costs of utilizing self-owned, spot and on-demand instances are increasing. To be cost-optimal, an allocation policy should sequentially maximize the utilization of self-owned and then spot instances. This is, however, a difficult task. For instance, *a naive policy* would be, whenever a job arrives, to assign as many remaining self-owned instances as possible to it. However, this policy turns out not to be good wrt cost. Indeed, it ignores the difference of jobs and treats all jobs equally when assigning instances, whereas we find that a good policy wrt cost needs instead to determine the allocations of self-owned instances to jobs according to their capabilities of utilizing spot instances. In particular, subject to the parallelism constraint of a job j, the availability of spot instances varies in the period between its arrival

- *Xiaohu Wu is with Fondazione Bruno Kessler, Trento, Italy. E-mail: xiaohuwu@fbk.eu*
- *Patrick Loiseau is with Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, France and MPI-SWS, Germany. E-mail: patrick.loiseau@inria.fr*
- *Esa Hyytiä is with University of Iceland, Reykjavík, Iceland. E-mail: esa@hi.is*

Manuscript received April 19, 2005; revised August 26, 2015.

1. In this paper, we use "users" and "tenants" interchangeably.

and deadline and determines the maximum workload of j that could be processed by spot instances. If the workload of j is large, j has to utilize some stable self-owned or on-demand instances in order to finish itself by the deadline; such job is said to have poor capability of utilizing spot instances alone to finish itself by its deadline (also called poor jobs). If the workload of j is small, finishing j by its deadline only needs to utilize spot instance alone, with no need of self-owned or on-demand instances; such job is said to have strong such capability (called rich job).

When self-owned instances are inadequate, actively assign self-owned instances to poor jobs and assign nothing to rich jobs; otherwise, such poor jobs will have to consume costly on-demand instances, and it also causes a waste of other rich jobs' capabilities to utilize spot instances. When self-owned instances are adequate, assign a proper amount of self-owned instances to every job with either poor or strong capability such that after the allocations all jobs are expected to be completed by utilizing spot instances alone, eliminating the need of consuming costly on-demand instances. After allocating self-owned instances, the remaining question is to identify a job's capacity to utilize spot instances for processing its workload, and propose an expected optimal policy to achieve the capacities of jobs, further escaping unnecessary consumption of on-demand instances.

Our Contributions. In this paper, we propose a framework to design policies to allocate various instances. Based on the two principles that (i) self-owned instances should be allocated to maximize their utilization while maximizing the opportunity of all jobs utilizing spot instances and (ii) on-demand instances should be allocated to maximize the opportunity to utilize spot instances, we propose parametric policies for the allocation of self-owned, on-demand and spot instances that achieve near-minimal costs. To cope with the cloud market dynamic and the uncertainty of job's characteristics, we use the online learning technique in [8] to infer the optimal parameters. More specifically:

- We propose a cost-effective policy for allocating self-owned instances that is smarter than the naive allocation mentioned above and hits a good trade-off between the utilization of self-owned instances and the opportunity of utilizing spot instances. We show in our numerical experiments that this policy improves the cost by up to 43.74% compared to the naive policy.
- We propose a cost-optimal policy for the utilization of on-demand and spot instances, based on a formulation of the original problem as an integer program to maximize the utilization of spot instances. This policy can be used both when the tenant has self-owned resources and when he does not. Our simulation results show that it improves the cost of previous policies in [8] by up to 64.51%.

We note that the paper [8] also appears in [7] as a U.S. Patent.

The rest of this paper is organized as follows. We introduce the related works in Section 2 and describe the problem formally in Section 3. In Section 4, we propose scheduling policies for self-owned, on-demand and spot instances. In Section 5, simulations are done to show the effectiveness of the solutions of this paper. Finally, we conclude this paper in Section 6. The proofs of some propositions are omitted and can be found in the supplementary material, available online with this paper and in the item "Media" at IEEE Xplore Digital Library. We note that a part of results of this paper also appeared at the 2017 IEEE International Conference on Cloud and Autonomic Computing [25].

2 Related Work

In this paper, we use the online learning technique to learn the most effective parametric policy for utilizing various instances. Jain et al. were the first to consider the application of this approach to the scenario of cloud computing[2] [8]. However, they do not consider the problem of how to optimally utilize the purchase options in IaaS clouds and self-owned instances are also not taken into account. This approach is interesting because it does not impose the restriction of a priori statistical knowledge of workload, compared to other techniques such as stochastic programming. However, it can achieve good performances only if the potentially optimal scheduling policies are identified among all possible policies. Similar to our paper and [8], cost-effectively executing deadline-constrained jobs in IaaS clouds is also studied in [13], [14]. In particular, Zafer et al. characterize the evolution of spot prices by a Markov model and propose an optimal bidding strategy for utilizing spot instances to complete a serial or parallel job by some deadline [13]. Yao et al. study the problem of utilizing reserved and on-demand instances to complete online batch jobs by their deadlines and formulate it as integer programming problems; then heuristic algorithms are proposed to give approximate solutions [14].

There have been substantial works on cost-effective resource provisioning in IaaS clouds [15], and we introduce some typical approaches. Built on the assumption of a priori statistical knowledge of the workload or spot prices, several techniques could be applied. For example, in [16], [17], the techniques of stochastic programming is applied to achieve the cost-optimal acquisition of reserved and on-demand instances; in [24], the optimal strategy for the users to bid for the spot instances are derived, given a predicted distribution over spot prices. However, a high computational complexity arises when implementing these techniques, though the statistical knowledge could be derived by the techniques such as dynamic programming [20]. Wang et al. use the competitive analysis technique to purchase reserved and on-demand instances without knowing the future workload [18], where the Bahncard problem is applied to propose a deterministic and a randomized algorithm. In [19], a genetic algorithm is proposed to quickly approximate the pareto-set of makespan and cost for a bag of tasks where on-demand and spot instances are considered. In [20], the technique of Lyapunov optimization is applied and it's said to be the first effort on jointly leveraging all three common IaaS cloud pricing options to comprehensively reduce the cost of users; however, a large delay will be caused when processing jobs; in order to achieve an $\mathcal{O}(\epsilon)$ close-to-optimal performance, the queue size has to be $\Theta(1/\epsilon)$ [21].

3 Problem Description and Model

In this section, we introduce the cloud pricing models, define the operational space of a user to utilize various instances, and characterize the objective of this paper.

3.1 Pricing Models in the Cloud

We first introduce the pricing models in the cloud. The price of an *on-demand instance* is charged on an hourly basis and it is fixed and denoted by p. Even if on-demand instances are consumed for part of an hour, the tenant will be charged the fee of the entire hour, as illustrated in Fig. 1.

2. The objective of this paper corresponds to a special case of [8] where the value of each job is larger than the cost of completing it.

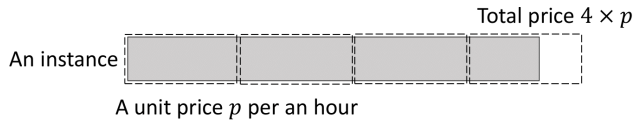

Fig. 1. On-demand price: users are charged on an hourly basis at a fixed price p.

Furthermore, tenants can bid a price for *spot instances* and *spot prices are updated at regular time slots* (e.g., every $L = 5$ minutes in Amazon) [24]. Spot instances are assigned to a job and continue running if the spot price is lower than the bid price. Spot prices usually change unpredictably over time [4]. Once the spot price exceeds the bid price of a job, its spot instances will get terminated immediately by the cloud, as illustrated in Fig. 2; here, the termination occurs at the very beginning of a time slot. The tenant will be charged the spot prices for the maximum integer hours of execution. A partial hour of execution is not charged in the case where its instances are terminated by the cloud; in contrast, if spot instances run until a job is completed and then are terminated by the tenant, for the partial hour of execution, the tenant will also be charged for the full hour.

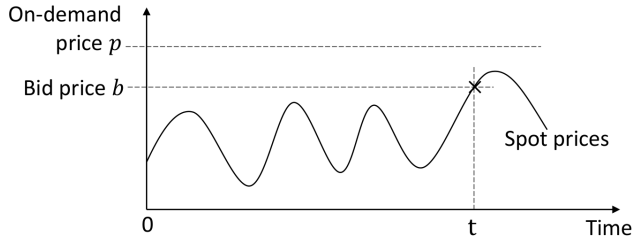

Fig. 2. Spot price: a user bids a price b for an instance at time 0 and can use it until time t.

Finally, a user might have its own instances, i.e., *self-owned instances*. The (averaged) hourly cost of utilizing self-owned instances is assumed to be p_1. It is the cheapest to use self-owned instances so that p_1 is without loss of generality assumed to be 0, which implies that a user always prefers to first utilize its own instances before purchasing instances from the cloud. An example of self-owned instances is academic private clouds, which are provided to researchers free of charge.

3.2 Jobs

Our paper focuses on processing a type of embarrassingly parallel computations, also called map-only jobs; see [6] for the classification of big-data applications on clouds. Each job can be separated into a large number of small tasks. These tasks are independent and can be executed on multiple machines simultaneously; completing a job means completing all its tasks, and the maximum completion time of all tasks is the job's completion time. Examples of such computations include parallel rendering in computer graphics, BLAST searches and CAP3 in bioinformatics, large scale facial recognition systems that compare thousands of faces, grid and random search for hyperparameter optimization in machine learning, data cleaning and pre-processing and so on. A task is the minimum running unit and should be processed continuously without preemption until its completion.

For example, in CAP3, the data are divided into many files and each task finishes reading a file; if each file has 458 reads, it may take about 7 seconds to process a task when two high CPU extra large instances are used with 8 workers per instance [5]. For each job j, its size/workload is defined as the time when finishing it on one instance, denoted by z_j, and can be estimated by the input data size or the number of small tasks. While executing a job, the remaining workload can also be estimated by the remaining input data to be processed. Such jobs can be formally modeled as malleable jobs in literature [9], [11]. Each job j has a parallelism bound δ_j that limits the maximum number of instances that it could utilize simultaneously. While executing a job, the number of instances assigned to a job could change over time.

The job arrival of a tenant is monitored every time slot of L minutes (i.e., at the time points when spot prices change) and time slots are indexed by $t = 1, 2, \cdots$. Each job j has four characteristics: (i) *an arrival slot* a_j: If job j arrives at a certain continuous time point in $[(t-1) \cdot L, t \cdot L)$, then set a_j to t; (ii) *a relative deadline* $d_j \in \mathcal{Z}^+$: it is a time constraint on completing a job, that is, every job must be completed at or before time slot $a_j + d_j - 1$; (iii) *a job size* z_j (measured in the instance time slots (CPU time) that need to be utilized): the workload to complete j; (iv) *a parallelism bound* δ_j: the upper bound on the number of instances that could be simultaneously utilized by j. The tenant plans to rent instances in IaaS clouds to process its jobs and aims to minimize the cost of completing a set of jobs $\mathcal{J}$ (that arrive over a time horizon T) by their deadlines.

3.3 Rules for Allocating Instances to Jobs

The pricing models define the rules of allocating instances to jobs and also the operational space of a user, i.e., (i) the moment that allocation is done and updated, and (ii) how different instances are utilized at every allocation.

3.3.1 On-demand and spot instances

We first consider the allocation of on-demand and spot instances alone, ignoring self-owned instances temporarily.

To meet deadlines, we assume that **(i)** *whenever a job j arrives at a_j, the allocation of spot and on-demand instances to it is done immediately*. The following rules apply to the case where j has flexibility to utilize spot instances. Given the fact that the tenant is charged on hourly boundaries, **(ii)** *the allocation of on-demand and spot instances to each job j is updated simultaneously every hour.* At the i-th allocation to j, the number of on-demand instances allocated to j is denoted by o_j^i and they can be utilized for the entire hour; we assume that **(iii)** *the tenant will bid a price b_j^i for a fixed number si_j^i of spot instances.* At the i-th allocation of j, b_j^i together with the spot prices determines whether j can successfully obtain spot instances and how long it can utilize them. Usually, spot instances are on average cheaper than on-demand instances, and we assume that **(iv)** *at every allocation the tenant will bid for the maximum number of spot instances under the parallelism constraint, i.e., $si_j^i = \delta_j - o_j^i$.* The crucial question is thus how to determine the proportion of on-demand and spot instances, i.e., o_j^i and si_j^i, that are acquired from the cloud.

Before the i-th allocation to j, we use z_j^i to denote the remaining workload of j to be processed, i.e., z_j minus the workload of j that has been processed, where $z_j^1 = z_j$. We define the current slackness of j as

$$s_j^i = \frac{(d_j - (i-1) \cdot Len) \cdot \delta_j}{z_j^i}, \tag{1}$$

where $Len = 60/L$ is the number of slots per hour. Let $s_j = s_j^1 = (d_j \cdot \delta_j)/z_j$. The slackness can be used to measure the time flexibility that j has to utilize spot instances; the process of allocating on-demand and spot instances to j is in fact divided into two phases by the value of s_j^i:

Definition 3.1. *When spot instances get lost at the very beginning of slot t' and are not utilized for the entire hour at the i-th allocation of j, we say that, at the next allocation,*

1) *j has flexibility to utilize spot instances, if $s_j^{i+1} \geq 1$;*
2) *j does not have such flexibility, otherwise.*

The intuition of Definition 3.1 is as follows. After the i-th allocation, when spot instances are interrupted, we could know the job slackness s_j^{i+1} at the $(i+1)$-th allocation update; here, z_j^{i+1} is known since we know z_j^i and the total workload processed by spot and on-demand instances at the i-th allocation. After the i-th allocation, when spot instances are interrupted, if $s_j^{i+1} < 1$, this means that j has to give up utilizing spot instances at the next allocation update; otherwise, since the availability of spot instances is uncertain, j cannot be completed by its deadline even if j totally utilizes stable on-demand instances from the $(i+1)$-th allocation update until its completion. In this case, once the spot instances of j is interrupted at a moment after the i-th allocation, j can immediately turn to totally utilize stable on-demand instances. In contrast, if $s_j^{i+1} \geq 1$, j is still able to utilize unstable spot instances at the next allocation update. We also illustrate Definition 3.1 by Fig. 3. As illustrated in Fig. 3, $z_j = z_j^1 = 132$ and, at the 1st allocation, $o_j^1 = si_j^1 = 2$; then $z_j^2 = 132 - 2 \cdot 12 - 2 \cdot 8 = 92$. At the 2nd update, o_j^2 and si_j^2 are still 2 and then $z_j^3 = 92 - 2 \cdot 12 - 2 \cdot 8 = 52$. Further, $s_j^3 = \frac{Len \cdot \delta_j}{z_j^3} < 1$ and there is no flexibility for j to utilize spot instances at the 3rd allocation. We use i_j to index the last allocation of j after which there is no such flexibility; in Fig. 3, $i_j = 2$. As a result, the decision on how to determine the (i_j+1)-th allocation of instances to j has to be done earlier, since there exists an on-demand instance that has to be utilized for $\frac{4}{3}$ hours to satisfy the deadline constraint.

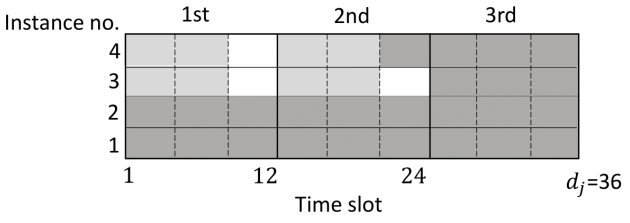

Fig. 3. Illustration of the process of allocating resource to j where $a_j = 1$, d_j equals 3 hours, $L = 5$ minutes, $z_j = 132$, and $\delta_j = 4$: the light gray (resp. heavy gray) area in every period of $[12 \cdot (i-1)+1, 12 \cdot i]$ illustrates the workload processed by spot (resp. on-demand) instances at the i-th allocation where $i = 1, 2, 3$.

As illustrated above, the instance allocation is divided into two phases. In the first phase,

- the instance allocation is updated every hour (i.e., every Len slots).

At every i-th allocation of j, the remaining workload to be processed by spot and on-demand instances is z_j^i; on-demand instances are charged on an hourly basis and the workload that could be processed by on-demand instances is $Len \cdot o_j^i$. At every i-th allocation, as time goes by, there are two possible states for spot instances:

(i) if $z_j^i - Len \cdot o_j^i$ workload of j has been processed by spot instances, they will be terminated by the user itself;
(ii) if the bid price is below the spot price, the user will lose its spot instances immediately; otherwise, they will be utilized for an hour.

The first state occurs because j will be finally completed after the on-demand instances acquired at this allocation are consumed. If the second state occurs, we need to check whether or not job j still has flexibility to utilize spot instances using Definition 3.1: if there is such flexibility, the next allocation update of j is still in the first phase; otherwise,

- the next allocation of j (i.e., the (i_j+1)-th allocation) needs to be done immediately after the spot instances of the i_j-th allocation get lost,

which is referred to as the second phase of instance allocation. In the second phase, only stable on-demand instances will be used to meet the deadline.

3.3.2 Self-owned instances

When self-owned instances are also taken into account, we assume that (**v**) *the allocation of self-owned instances to a job can be updated at most once at every allocation of j*. We denote by r_j^i the number of self-owned instances assigned to j at the i-th allocation; the parallelism constraint further translates to $o_j^i + si_j^i + r_j^i = \delta_j$. In this paper, o_j^i and si_j^i denotes the numbers of on-demand and spot instances acquired at the i-th allocation and will be used to track the cost of completing j. As we will see in Section 4.2, the acquired on-demand instances may not be fully utilized for an entire hour in the (i_j+1)-th execution, and, we use $o_j(t)$, $si_j(t)$ and $r_j(t)$ to denote the numbers of on-demand, spot and self-owned instances that are actually utilized by j at every slot $t \in [a_j, a_j + d_j - 1]$, where $r_j(t) = r_j^i$ for all $t \in [a_j + (i-1) \cdot Len, a_j + i \cdot Len - 1]$; then the parallelism constraint translates to $o_j(t) + si_j(t) + r_j(t) = \delta_j$.

As shown later, allocating properly self-owned instances enables escaping unnecessary consumption of on-demand instances that are more expensive than the others, which can be achieved by simply allocating j the same number of self-owned instances at every time slot, i.e., $r_j(t) = r_j$. So, the allocation of self-owned instances is done only once upon arrival of j; after the allocation, the job can could be viewed as a new job with a parallelism bound $\delta_j - r_j$, a size $z_j - r_j \cdot d_j$, and the same arrival time and deadline as j , and it will be completed by utilizing spot and on-demand instances alone.

3.4 Scheduling Objective

We refer to the ratio of the total cost of utilizing a certain type of instances to the total workload processed by this type of instances as the average unit cost of this type of instances. As described in Section 3.1, we assume that

Assumption 1. *The average unit costs of self-owned instances is lower than the average unit cost of spot instances, which is lower than that of on-demand instances.*

Accordingly, to be cost-optimal, we should consider allocating various instances to each arriving job in the order of self-owned,

spot and on-demand instances. Further, in Principles 3.1 and 3.2, we give the objectives that should be achieved when considering allocating each type of instances to the arriving jobs.

Principle 3.1. *The scheduler should make self-owned instances (i) fully utilized, and (ii) utilized in a way so as to maximize the opportunity that all jobs have to utilize spot instances.*

Principle 3.2. *After self-owned instances are used, the scheduler should utilize on-demand instances in a way so as to maximize the opportunity that all jobs have to utilize spot instances.*

3.4.1 Decision variables

Our main objective of this paper is to propose scheduling policies that can realize Principles 3.1 and 3.2. To do so, we will first determine the allocation of self-owned instances and then the allocation of on-demand and spot instances for every arriving job j. For every arriving job j, it will be first allocated r_j self-owned instances in $[a_j, a_j + d_j - 1]$. Then, as described in Section 3.3, the allocation of spot and on-demand instances will be updated per hour in the first phase and we need to determine the number of spot instances to be bid for and the number of on-demand instances to be purchased (i.e., si_j^i and o_j^i); once there is no flexibility for j to utilize spot instances, we need to determine the allocation of on-demand instances alone in order to complete j by deadline. Hence, the main decision variables of this paper are r_j, si_j^i, and o_j^i where $o_j^i + si_j^i + r_j = \delta_j$.

In this paper, we apply the online learning approach and it does not require the exact statistical knowledge on jobs and spot prices. At every allocation update of j in the first phase, only the current characteristics of j (i.e., z_j^i, δ_j, a_j, and d_j) and the amount of available self-owned instances are definitely known for a user to determine o_j^i and si_j^i. The value of spot price is jointly determined by the arriving jobs of numerous users and the number of idle servers at a moment, usually varying over time unpredictably. In this paper, it is assumed that the change of spot prices over time is independent of the job's arrival of an individual user [13], [24]. In the i-th execution of j, when a user bids some price for si_j^i spot instances, without considering the case where the spot instances of j is terminated by a user itself, the period in which j can utilize spot instances is a random variable and we assume that the expected time for which j could utilize spot instances is $\beta \cdot Len$ where $\beta \in [0, 1]$. Finally, Table 1 summarizes the main notation of this paper.

TABLE 1
Main Notation

Symbol	Explanation
L	length of a time slot (e.g., 5 minutes)
Len	the number of time slots in an hour, i.e., $\frac{60}{L}$
$\mathcal{J}$	a set of jobs that arrive over time
j and a_j	a job of $\mathcal{J}$ and its arrival time
d_j	the relative deadline: j must be completed by a deadline $a_j + d_j - 1$
z_j	the job size of j, measured in CPU $\times$ time slots
δ_j	the parallelism bound, i.e., the maximum number of instances that can be simultaneously used by j
s_j	the slackness, i.e., $\frac{d_j}{z_j/\delta_j}$ where z_j/δ_j denotes the minimum execution time of j
T	the number of time slots, i.e., $\max_{j\in\mathcal{J}}\{a_j\}$
p and p_1	the price of respectively using an on-demand and self-owned instance for an hour
si_j^i, b_j^i, and o_j^i	the number of spot instances bid for, the bid price, and the number of on-demand instances acquired at the i-th allocation of j
$r_j(t)$, $si_j(t)$ and $o_j(t)$	the number of self-owned, spot and on-demand instances utilized by j at a slot t
p_j^i	the spot price charged in the i-th execution of j
z_j^i	the remaining workload of j to be processed at the i-th allocation of j
s_j^i	the slackness at the i-th allocation, i.e., $(d_j - (i-1)\cdot Len)\cdot \delta_j / z_j^i$
i_j	the last allocation of j at which j has flexibility to utilize spot instances
r_j	the number of self-owned instances allocated to a job j at every $t \in [a_j, a_j + d_j - 1]$
β	the availability of spot instances varies over time; at every allocation, the expected time for which a job could utilize spot instances is $\beta \cdot Len$ where $\beta \in [0, 1]$
R	the number of self-owned instances
$N(t)$	the number of currently idle self-owned instances at a slot t
$m_{t_1}(t_2)$	the maximum number of self-owned instances idle at every slot in $[t_1, t_2]$, i.e., $\min\{N(t_1), \cdots, N(t_2)\}$
b	the bid price
β_0	a parameter that control the allocation of self-owned instances via Equation (4)
$\{\beta, \beta_0, b\}$	a parameterized policy for allocating various instances to a job at every allocation, as stated in the Section 4.3
$\mathcal{P}$	a set of parameterized policies
π	the index of a policy in $\mathcal{P}$: $\pi = 1, 2, \cdots$

4 The Design of Near-Optimal Policies

In this section, we propose a theoretical framework to design (near-)optimal parametric policies, aiming at realizing Principles 3.1 and 3.2. Facing diverse users, the proposed policies should have good adaptability against the unknown statistics of the spot prices and each individual user's job characteristics; then, by applying the online learning technique, the best configuration parameter that corresponds to each user could be inferred to minimize its cost of processing jobs.

Upon arrival of a job j, the scheduler first considers the allocation of self-owned instances to it, aiming to realize the two goals in Principle 3.1. Next, as described in Section 3.3.1, the allocation of spot and on-demand instances is updated on an hourly basis.

4.1 Self-owned Instances

In this subsection, we study the allocation of self-owned instances.

4.1.1 Challenge

We first show the challenges in cost-effectively utilizing self-owned instances by an example. Initially, there is a fixed number R of self-owned instances. Upon arrival of a job j, it is allocated a fixed number of self-owned instances, and these instances will be reserved for this job in the period $[a_j, a_j + d_j - 1]$ and released by j after the slot $a_j + d_j - 1$. As time goes by, when time is at the beginning of any slot t, we have the information on the allocation of self-owned instances to the previous jobs (i.e., the number of self-owned instances allocated to each previous job and the period in which these instances are reserved for this job) and we can get the number of self-owned instances that are not reserved for the previous jobs in the period of each slot t', denoted by $N(t')$. Let $m_{t_1}(t_2) = \min\{N(t_1), \cdots, N(t_2)\}$, where $t_1 \le t_2$, and it represents the maximum number of self-owned instances idle/non-reserved at every slot in $[t_1, t_2]$. **An intuitive policy** would be,

whenever a job j arrives, to allocate as many self-owned instances to j to make self-owned instances fully utilized, i.e.,

$$r_j = \min\{m_{a_j}(a_j + d_j - 1), z_j/d_j\}. \quad (2)$$

However, this intuitive policy may not maximize the opportunity that all jobs have to utilize spot instances as illustrated in the following example.

There are two self-owned instance available, and two jobs whose have the same arrival time, relative deadline of 2 hours and parallelism bound of 4. Jobs 1 and 2 have a size of $4 \times Len$ and $6 \times Len$, respectively. It is expected that a job can utilize spot instances for $\beta = \frac{1}{2}$ hour ($\beta \cdot Len$ slots) at every allocation update. In Fig. 4, the area of diagonal stripes, the area of vertical stripes, and the dotted area denote the workload respectively processed by spot, self-owned and on-demand instances. Using the policy (2), the whole process of allocating instances is illustrated in Fig. 4 (left), where the user has to utilize two on-demand instances for 0.5 hour; however, it is not necessary to purchase more expensive on-demand instances if the allocation process is like Fig. 4 (right). In Fig. 4 (left), the cost of completing jobs 1 and 2 is $2 \cdot p$ while it is zero in Fig. 4; here, on-demand instances are charged on an hourly basis, and the fee of utilizing spot instances is zero when they are terminated by the cloud.

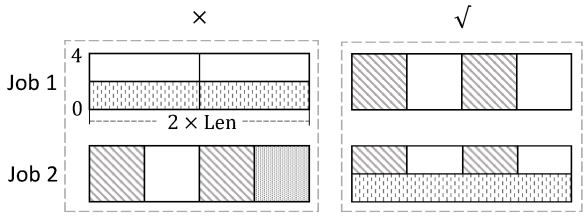

Fig. 4. The Challenge in Cost-Effectively Utilizing Self-owned Instances.

The above example reveals some challenges in designing cost-effective policies for allocating self-owned instances. For example, the policy should have the ability of (i) identifying the subset of jobs that can be expected to be completed by utilizing spot instances alone even if they are not allocated any self-owned instance, e.g., the job 1 in Fig. 4 (right), and (ii) properly allocating self-owned instances to the rest of jobs, when self-owned instances are inadequate. All in all, our aim is to realize Principle 3.1.

4.1.2 Policy Design

In the following, we propose a policy that has the abilities described above. In the subsequent analysis, the issue of rounding the allocations of a job to integers is ignored temporarily for simplicity; in reality, we could round the allocations up to integers, which does not affect the related conclusions much as shown by the analysis.

Recall the meaning of β in Section 3.4. For every job j, we will go to find a function $g_j(x) \in [0, \frac{z_j}{d_j}]$ that satisfies the following properties where $\frac{z_j}{d_j} \leq \delta_j$:

Property 4.1. *$g_j(x)$ is non-increasing as x increases in $[0, 1)$.*

Property 4.2. *$g_j(\beta)$ is the minimum number such that when a job j is assigned r_j self-owned instances in $[a_j, a_j + d_j - 1]$ where $r_j \geq g_j(\beta)$, it could be expected that*

- *job j could be completed by its deadline by utilizing spot instances alone if $\delta_j - r_j$ spot instances are bid for at every allocation update of j, where no on-demand instances is acquired.*

The value of $g_j(\beta)$ is an indicator of the capability that j has such that it can be completed by utilizing spot instances alone. By Property 4.2, if $g_j(\beta) \leq 0$, it is expected that no self-owned or on-demand instances is needed in order to complete j and such jobs have strong capability to feed themselves with spot instances. Otherwise, $g_j(\beta)$ self-owned instances are needed, or j has to consume some amount of expensive on-demand instances in order to be completed by its deadline; for a job j, the larger the value of $g_j(\beta)$, the weaker its capability to feed itself with spot instances.

Let $\kappa_0 = \lceil \frac{d_j}{Len} \rceil - 1$, and we set

$$\overline{r}_j(x) = \begin{cases} r'_j(x) & \text{if } d_j - \kappa_0 \cdot Len > x \cdot Len, \\ r''_j(x) & \text{if } d_j - \kappa_0 \cdot Len \leq x \cdot Len, \end{cases}$$

where

$$r'_j(x) = \delta_j - \frac{d_j \cdot \delta_j - z_j}{d_j - (\kappa_0 + 1) \cdot Len \cdot x},$$

and

$$r''_j(x) = \begin{cases} 0 & \text{if } \kappa_0 = 0, \\ \delta_j - \frac{d_j \cdot \delta_j - z_j}{(1-x) \cdot \kappa_0 \cdot Len} & \text{if } \kappa_0 \geq 1. \end{cases}$$

We further set

$$g_j(x) = \max\left\{\overline{r}_j(x), 0\right\}. \quad (3)$$

When $x = 0$, $g_j(x) = \max\{r'_j(x), 0\} = \frac{z_j}{d_j}$. When $x \to 1$, $g_j(x) = \max\{r''_j(x), 0\}$ and we have that (i) if $\kappa_0 = 0$, $g_j(x) = 0$, (ii) if $\kappa_0 \geq 1$ and $d_j \cdot \delta_j = z_j$, $g_j(x) = \frac{z_j}{d_j}$, and (iii) if $\kappa_0 \geq 1$ and $d_j \cdot \delta_j > z_j$, $g_j(x) = 0$ since $r''_j(x) \to -\infty$. Now, we proceed to show that the particular $g_j(x)$ in (3) satisfies Properties 4.2 and 4.1.

Proposition 4.1. *The function $g_j(x)$ in (3) satisfies Property 4.2.*

Proposition 4.2. *The function $g_j(x)$ in (3) satisfies Property 4.1.*

In this paper, we consider a set of jobs $\mathcal{T}$ that arrive over time and can have diverse characteristics. When x ranges in $[0, 1)$, we illustrate the function $g_j(x)$ in Fig. 5 where $z_j = 240$, $L = 5$, $\delta_j = 20$, and $Len = 12$. The job's minimum execution time is $\frac{z_j}{\delta_j} = Len$ where j is assigned δ_j instances throughout its execution. The job's deadline reflects its ability to utilize spot instances and in Fig. 5 the solid curves from left to right represent $g_j(x)$ where d_j is respectively 5, 3, 2.1, 1.47, 1.25, 1.11, and 1.02 times Len: under the same x, the larger the deadline, the smaller the value of $g_j(x)$. Given z_j, δ_j and d_j, we can see in Fig. 5 that the function $g_j(x)$ is non-increasing as x ranges in $[0, 1)$.

Proposed Policy. Based on Propositions 4.1 and 4.2, we propose the following policy for allocating self-owned instances. Upon arrival of every job j, it is allocated $r_j(\beta_0)$ self-owned instances where

$$r_j(\beta_0) = \min\left\{g_j(\beta_0), m_t(a_j + d_j - 1)\right\}, \quad (4)$$

where $\beta_0 \in [0, 1)$ is a parameter to be learned.

This policy achieves more cost-effective resource allocation as illustrated in Fig. 4 (right) where β_0 is set to $\beta = \frac{1}{2}$. Furthermore, this policy is also adaptive. For example, given another user who owns more instances (e.g., 5 instances), β_0 can be set to a value $< \beta$ (e.g., 0); then, both jobs are allocated more self-owned

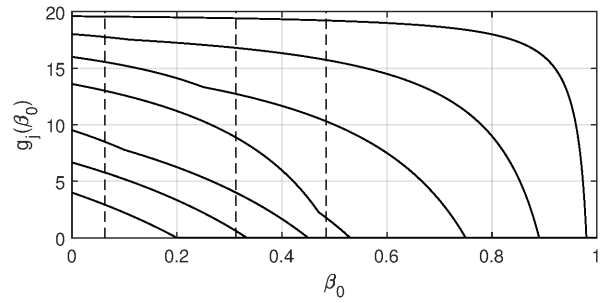

Fig. 5. As x ranges in $[0, 1)$, the function $g_j(x)$ for jobs respectively with different flexibility to utilize spot instances.

instances: $r_1 = 2$, and $r_2 = 3$. As a result, self-owned instances are fully utilized and there is no need purchasing spot or on-demand instances.

4.1.3 Explanation

Now, we further explain that the policy (4) has desired properties to realize Principle 3.1, which will also be validated by the simulations.

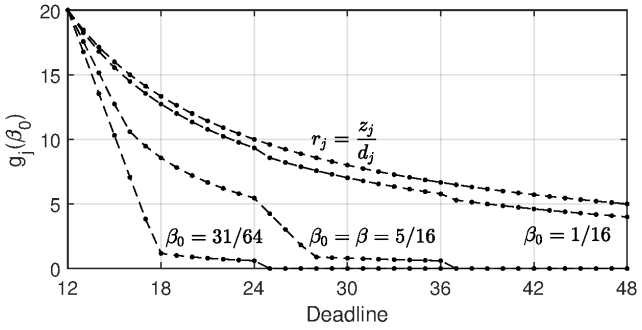

Fig. 6. As the (relative) deadline d_j increases from 12 to 48, the function $g_j(\beta_0)$ decreases respectively under $\beta_0 = \frac{31}{64}$, $\frac{5}{16}$, $\frac{1}{16}$, where $z_j = 240$, $\delta_j = 20$, and $Len = 12$.

The allocations of self-owned instances to all jobs are based on the same function (4) whose value depends on a single parameter β_0. Together with Properties 4.2 and 4.1, the power of the proposed policy can be achieved by setting β_0 to a value properly small in $[0, 1)$. Now, we explain this.

High Utilization. As illustrated in Fig. 5, the function $g_j(x)$ is non-increasing; no matter how many self-owned instances a user possesses, a high utilization of them is achieved after

- we set β_0 to a small enough value in $[0, 1)$.

This is because every arriving job will be assigned a large number of self-owned instances when β_0 is small, as illustrated in Fig. 6.

Fair Allocation. Fair allocation means that the allocations of self-owned instances among jobs need to be balanced according to their capabilities of utilizing spot instances. Fair allocation avoids ignoring the difference among jobs and treating them equally where a policy like (2) is used; together with Property 4.2, the latter can lead to that "rich" jobs (i.e., jobs with strong capabilities where $g_j(\beta)$ is small) are consuming unnecessary self-owned instances, i.e.,

- $r_j > g_j(\beta)$, where r_j denotes the number of self-owned instances allocated to a job; the job's remaining $z_j - r_j \cdot d_j$ workload is expected to be processed by spot instances alone;

whereas the others (with large $g_j(\beta)$) are allocated poorly and still starving for more self-owned instances, i.e.,

- $r_j < g_j(\beta)$; here, on-demand instances are expected to be consumed.

Indiscriminate allocations of instances to jobs do harm to the process of achieving the capacity that jobs have for utilizing spot instances, causing unnecessary consumption of more on-demand instances and a higher cost of completing all jobs. In particular, for every rich job, only $g_j(\beta)$ self-owned instances are needed to complete its remaining workload without on-demand instances; the saved $r_j - g_j(\beta)$ self-owned instances can be used for those poorly allocated jobs so as to reduce their consumption of on-demand instances, which improves the cost-efficiency of instance utilization.

Now, we explain that the proposed policy achieves fair allocation by properly setting the value of β_0. The cost-optimal β_0, denoted by β_0^*, depends on the statistics of jobs and the amount of self-owned instances available; the online learning technique will be used subsequently in Section 4.4 to infer β_0^*. When $\beta_0^* = 0$, self-owned instances themselves are enough to complete all jobs by their deadlines where $g_j(\beta_0) = \frac{z_j}{d_j}$.

When there are adequate self-owned instances such that $\beta_0^* \in (0, \beta]$, every arriving job j will be allocated $\geq g_j(\beta)$ self-owned instances whenever the amount of idle self-owned is large (i.e., $m_{a_j}(a_j + d_j - 1) \geq g_j(\beta_0)$), according to the policy (4); this is illustrated in Fig. 6 where $\beta = \frac{5}{16}$ and $\beta_0^* = \frac{1}{16}$. Afterwards, the job j is expected to be completed by utilizing spot instances alone. No job will be allocated $< g_j(\beta)$ self-owned instances whenever possible, and fair allocation is achieved. Furthermore, the arriving jobs are allocated on a first come first served basis and we note that β_0 should be properly small but cannot be set to a value too small. If β_0 is too small, jobs that arrive earlier might consume too many self-owned instances and then the jobs that arrive late have less opportunity to get $\geq g_j(\beta)$ self-owned instances subject to the availability of these instances (i.e., the value of $m_{a_j}(a_j + d_j - 1)$).

When self-owned instances are deficient such that $\beta_0^* \in (\beta, 1)$, every arriving job will be allocated $< g_j(\beta)$ self-owned instances; this is illustrated in Fig. 6 where $\beta = \frac{5}{16}$ and $\beta_0^* = \frac{31}{64}$. Afterwards, the job j is expected to have to utilizing some amount of on-demand instances to meet its deadline. No job will be allocated $> g_j(\beta)$ self-owned instances, achieving fair allocation among jobs: if there exists such allocation, a waste of self-owned instances is caused since we can reduce this allocation to $g_j(\beta)$ and allocate these saved instances to other jobs to reduce the consumption of on-demand instances.

4.2 Spot and On-demand Instances

As described in Section 3.3.1, the instance allocation process is divided into two phases. Now, we analyze the expected optimal strategy to utilize spot instances.

4.2.1 First phase

In the first phase, the allocation of j is updated per hour and there is flexibility for j to utilize spot instances. Now, we analyze the expected optimal policy in the first phase. One of the following two cases will happen: (i) the job j is completed in the first phase, and (ii) in the i_j-th execution of j, after spot instances

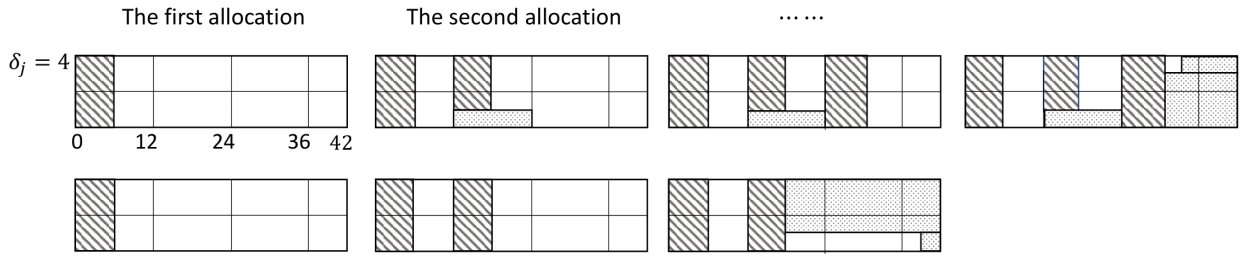

Fig. 7. Illustration of Proposition 4.6 in the case that $\nu(z_j, d_j) < (\kappa_0 - 1) \cdot \delta_j$ and $\kappa_2(z_j, d_j) < \kappa_3$.

are terminated by the cloud, there is no flexibility for j to utilize spot instances.

In this paper, the value of β is inferred by the online learning technique. If the previous allocation of self-owned instances r_j is $\geq g_j(\beta)$, it is expected that the first case will happen; then, by Property 4.2, we conclude that

Proposition 4.3. *An expected optimal strategy is to bid for $\delta_j - r_j$ spot instances at every allocation of j.*

Next, we analyze the optimal strategy when the second case happens. Job j is allocated r_j self-owned instances at every $t \in [a_j, a_j + d_j - 1]$; afterwards, it can be viewed as a new job with a workload $z_j - \delta_j \cdot d_j$ and a parallelism bound $\delta_j - r_j$, as described in Section 3.3.2. So, without loss of generality, we just analyze the optimal strategy in the case where a job j is completed by utilizing on-demand and spot instances alone.

Our decision variables are o_j^i and si_j^i where $o_j^i + si_j^i = \delta_j$. Let κ_1 denote the total number of allocation updates in the first phase where j has flexibility for spot instances; let $\kappa_0 = \lceil d_j / Len \rceil$ denoting the maximum possible number of allocation updates of j and we have

$$\kappa_1 \leq \kappa_0. \tag{5}$$

In the i-th execution of j where $i \in [1, \kappa_1]$, it is expected that the workloads processed by spot and on-demand instances are respectively $(\delta_j - o_j^i) \cdot Len \cdot \beta$ and $o_j^i \cdot Len$. By Definition 3.1, j has flexibility to utilize unstable spot instances at the κ_1-th allocation, i.e.,

$$s_j^{\kappa_1} = \frac{\delta_j \cdot (d_j - (\kappa_1 - 1) \cdot Len)}{z_j - \sum_{i=1}^{\kappa_1 - 1} \left(o_j^i \cdot Len + (\delta_j - o_j^i) \cdot Len \cdot \beta \right)} \geq 1,$$

and has no flexibility to utilize spot instances at the $(\kappa_1 + 1)$-th allocation, i.e.,

$$s_j^{\kappa_1 + 1} = \frac{\delta_j \cdot (d_j - \kappa_1 \cdot Len)}{z_j - \sum_{i=1}^{\kappa_1} \left(o_j^i \cdot Len + (\delta_j - o_j^i) \cdot Len \cdot \beta \right)} < 1.$$

They are respectively equivalent to the following relations:

$$\sum_{i=1}^{\kappa_1 - 1} (\delta_j - o_j^i) \cdot Len \cdot (1 - \beta) \leq d_j \cdot \delta_j - z_j, \tag{6}$$

$$\sum_{i=1}^{\kappa_1} (\delta_j - o_j^i) \cdot Len \cdot (1 - \beta) > d_j \cdot \delta_j - z_j. \tag{7}$$

For the condition that $s_j^{\kappa_1 + 1} < 1$, a special case is $\kappa_1 = \kappa_0$ where this condition holds trivially since $d_j - \kappa_1 \cdot Len \leq 0$; since $s_j^{\kappa_1} \geq 1$, the κ_1-th allocation of j is still in the first phase. In this subsection, our objective is to maximize the total workload processed by spot instances at the first κ_1 allocations, i.e.,

$$\text{maximize} \ \sum_{i=1}^{\kappa_1} (\delta_j - o_j^i) \cdot Len \cdot \beta, \tag{8}$$

subject to the constraints (5), (6), (7), and the constraint that o_j^i is an integer in $[0, \delta_j]$. Our decision variables are $o_j^1, \cdots, o_j^{\kappa_1}$.

Now, we give an optimal solution to (8).

Proposition 4.4. *An solution to (8) is optimal if it is of the following form: (i) $\sum_{i=1}^{\kappa_1 - 1} (\delta_j - o_j^i) = \min\{\nu(z_j, d_j), (\kappa_0 - 1) \cdot \delta_j\}$, and (ii) $o_j^{\kappa_1} = 0$, where*

$$\nu(z_j, d_j) = \left\lfloor \frac{d_j \cdot \delta_j - z_j}{Len \cdot (1 - \beta)} \right\rfloor.$$

Proposition 4.4 indicates the maximum number of spot instances that can be bid for in the first phase, i.e., the maximum value of $\sum_{i=1}^{\kappa_1} (\delta_j - o_j^i)$. As a corollary of Proposition 4.4, we conclude that

Proposition 4.5. *Given a job j, the expected maximum workload that can be processed by spot instances is*

$$(\min\{\nu(z_j, d_j), (\kappa_0 - 1) \cdot \delta_j\} + \delta_j) \cdot Len \cdot \beta.$$

Proposition 4.4 also implies an expected optimal strategy for spot instances.

Proposition 4.6. *Let $\kappa_2(z_j, d_j) = \lfloor \frac{\nu(z_j, d_j)}{\delta_j} \rfloor$ and $\kappa_3 = \frac{\nu(z_j, d_j)}{\delta_j}$. To maximize the total workload processed by spot instances, if $(\kappa_0 - 1) \cdot \delta_j \leq \nu(z_j, d_j)$, we can set $\kappa_1 = \kappa_0$ and* an expected optimal strategy *is to*

- *bid for δ_j spot instances at each allocation update of j.*

If $\nu(z_j, d_j) < (\kappa_0 - 1) \cdot \delta_j$, in the case that $\kappa_2(z_j, d_j) = \kappa_3$, we can set $\kappa_1 = \kappa_2(z_j, d_j) + 1$ and an expected optimal strategy *is to*

- *bid for δ_j spot instances at each of the first κ_1 allocations of j, i.e., $o_j^1 = \cdots = o_j^{\kappa_1} = \delta_j$;*

in the case that $\kappa_2(z_j, d_j) < \kappa_3$, we can set $\kappa_1 = \kappa_2(z_j, d_j) + 2$ and an expected optimal strategy *is to*

- *bid for δ_j spot instances at the 1st, $\cdots$, $(\kappa_1 - 2)$-th, κ_1-th allocations of j, i.e., $o_j^1 = \cdots = o_j^{\kappa_1 - 2} = o_j^{\kappa_1} = \delta_j$,*
- *bid for $\nu(z_j, d_j) - \kappa_2(z_j, d_j) \cdot \delta_j$ spot instances at the $(\kappa_1 - 1)$-th allocation of j, i.e., $o_j^{\kappa_1 - 1} = \nu(z_j, d_j) - \kappa_2(z_j, d_j) \cdot \delta_j$.*

We illustrate Proposition 4.6 in Fig. 7 where the area of diagonal stripes and the dotted area denote the workload processed respectively by spot and on-demand instances; in the blank area, no workload of j is processed. We assume that $\beta = \frac{1}{2}$ and $L = 5$ where $Len = 12$; job j has $d_j = 42$ (3.5 hours), $z_j = 122$ and $\delta_j = 4$. Here, we have $\nu(z_j, d_j) = 7$ and $\kappa_2(z_j, d_j) = 1$. From the left to the right, the first four subfigures illustrate the expected optimal allocation. At the first allocation of j, $\delta_j = 4$ spot instances are bid for and the expected execution time of spot instances is $\beta \cdot Len = 6$. At the second allocation of j, $(\nu(z_j, d_j) - \delta_j \cdot \kappa_2(z_j, d_j)) = 3$ spot instances are bid for and

one on-demand instance is purchased. So far, $\nu(z_j, d_j) = 7$ spot instances have been bid for. At the third allocation of j, δ_j spot instances are bid for and after the execution of spot instances, j has no flexibility to utilize spot instances and it turns to totally utilize on-demand instances as illustrated by the fourth subfigure. In contrast, we also use the last three subfigures to illustrate *an intuitive way* to utilize spot instances where δ_j instances are bid for at every allocation of j when there is flexibility for spot instances. After the execution of spot instances at the second allocation of j, it has no flexibility and turns to utilize on-demand instances since $s_j^3 < 1$. As illustrated in Fig. 7, the strategy in Proposition 4.6 can be explained as follows. Whenever possible, bid for the maximum number of spot instances (i.e., δ_j instances). An exception happens only at the second allocation of j where little flexibility is remaining, and we need to properly manage the instance allocation to ensure that there still exists flexibility to utilize spot instances at the third allocation of j: then, if δ_j spot instances are bid for at the second allocation, it is expected that there will be no flexibility for j to utilize spot instances at the third allocation; by bidding for less, it could be expected that the allocation will not get into the second phase and there will still be the last flexibility/opportunity at the third allocation of j.

Based on Proposition 4.6, we propose Algorithm 1 to dynamically determine the numbers of on-demand and spot instances allocated to j at every i-th allocation update when there is flexibility to utilize spot instances. At every allocation of j that occurs at slot t, the remaining workload of j to be processed could be viewed as a new job with the arrival time t, workload z'_j, parallelism bound δ_j, and relative deadline $a_j + d_j - t$; we always use Proposition 4.6 to determine the first allocation of this new job whose arrival time is t.

Algorithm 1: Proportion(j, β, b)

```
/* At the i-th allocation of j, its remaining
   workload is viewed as a new job with an
   arrival time t, and a relative deadline
   a_j + d_j - t                                   */
κ_0(t) ← ⌈(a_j + d_j - t)/Len⌉
/* the case (κ_0 - 1)·δ_j ≤ ν(z_j, d_j) in
   Proposition 4.6                                 */
if (κ_0(t) - 1)·δ_j ≤ ν(z_j, a_j + d_j - t) then
    si_j^i ← δ_j,  o_j^i ← 0;
else
    /* both cases κ_2(z_j, d_j) = κ_3 and
       κ_2(z_j, d_j) < κ_3 where κ_2(z_j, d_j) ≥ 1  */
    if κ_2(z'_j, a_j + d_j - t) ≥ 1 then
        si_j^i ← δ_j,  o_j^i ← 0;
    /* the case κ_2(z_j, d_j) < κ_3 where κ_2(z_j, d_j) = 0  */
    if κ_2(z'_j, a_j + d_j - t) = 0 ∧ ν(z_j, a_j + d_j - t) > 0
     then
        si_j^i ← ν(z_j, a_j + d_j - t),  o_j^i ← δ_j - si_j^i;
    /* the case κ_2(z_j, d_j) = κ_3 = 0                */
    if ν(z_j, a_j + d_j - t) = 0 then
        si_j^i ← δ_j,  o_j^i ← 0;
b_j^i ← b;
at the i-th allocation, bid a price b_j^i for si_j^i spot instances;
```

4.2.2 Second phase

As described in Section 3.3.1, once spot instances get lost at every allocation of j, the scheduler uses Definition 3.1 to check the flexibility to utilize spot instances. In the i_j-th execution, when spot instances get lost at the beginning of some slot t'_1, there is no such flexibility; then, the instance allocation enters the second phase where only on-demand instances are utilized. Now, we analyze their optimal utilization.

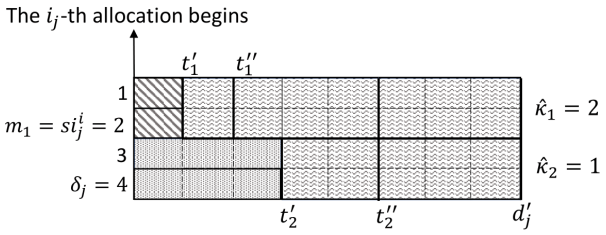

Fig. 8. The second phase of allocation where $i = i_j$: the area of waves denotes the available space in the second phase; the area of diagonal stripes and the dotted area respectively denote the workload processed in the i_j-th execution by spot instances and on-demand instances that are utilized for an hour.

As shown in Algorithm 2, at every allocation of j in the first phase (including the i_j-th allocation), the number of on-demand instances allocated to j is either 0 (see lines 3, 6, 10) or > 0 (see line 8). Let $t'_2 = a_j + i_j \cdot Len$, $d'_j = a_j + d_j - 1$, and we define two parameters that represent the maximum multiple of an hour (containing Len slots) respectively in time intervals $[t'_1, d'_j]$ and $[t'_2, d'_j]$:

$$\hat{\kappa}_1 = \left\lfloor \frac{d'_j - (t'_1 - 1)}{Len} \right\rfloor, \text{ and } \hat{\kappa}_2 = \left\lfloor \frac{d'_j - t'_2 + 1}{Len} \right\rfloor;$$

Let $t''_i = d'_j - \hat{\kappa}_i \cdot Len + 1$ ($i \in \{1, 2\}$), and after deducting $\hat{\kappa}_1$ and $\hat{\kappa}_2$ hours respectively from the two intervals, the numbers of remaining slots in $[t'_1, t''_1 - 1]$ and $[t'_2, t''_2 - 1]$ are denoted by ϕ_1 and ϕ_2:

$$\phi_1 = t''_1 - t'_1, \text{ and } \phi_2 = t''_2 - t'_2,$$

where $0 \le \phi_1, \phi_2 < Len$. The related notation is also illustrated in Fig. 8. Let

$$m_0 = si_j^i \cdot \hat{\kappa}_1 + o_j^i \cdot \hat{\kappa}_2, \ m_1 = si_j^i, \text{ and } m_2 = o_j^i,$$

where $i = i_j$. In Fig. 8, the available space in the second phase is the area of waves and m_0 represents the maximum integer of instance hour that can be utilized by j.

Since every on-demand instance is charged on an hourly basis, a cost-optimal strategy in the second phase is to minimize the integer instance hours (i.e., the number of on-demand instances $\times$ the time for which they are utilized). The following conclusion possibly is intuitive although a formal proof is also provided: whenever an instance is purchased for an hour, it should be utilized as long as possible with the space constraint.

Proposition 4.7. *Let $y = y_0 + y_1 + y_2$ be the minimum such that $y_0 \cdot Len + y_1 \cdot \phi_1 + y_2 \cdot \phi_2 \ge z_j^{i_j+1}$ subject to y_0, y_1, y_2 are non-negative integers and $y_0 \in [0, m_0]$, $y_1 \in [0, m_1]$, $y_2 \in [0, m_2]$. In the second phase, a cost-optimal strategy is to purchase on-demand instances for y instance hours*[3].

3. The specific value of y is given while proving this proposition, which can be found in the supplementary.

4.3 Scheduling Framework

As described above, a general policy is defined by a tuple $\{\beta_0, \beta, b\}$ and determines the amounts of self-owned, spot, and on-demand instances allocated to a job, and the bid price. The instance allocation process has been described in Section 3.3. Based on this, at every slot t, if a job j just arrives or it has arrived before but not been completed yet, we propose a framework, presented in Algorithm 2, to determine the action of allocating instances to j after checking the state of j. Actions are needed in the following three states: (i) t is the arrival time of j, determining the allocation of self-owned instances, (ii), t equals $a_j + (i-1) \cdot Len$ where the i-th allocation update of spot and on-demand instances needs to be done, (iii) the spot instances of j get lost at t where we need to check whether j still has flexibility for spot instances. In Algorithm 2, z'_j denotes the remaining workload of j to be processed after deducting its current allocations from z_j; upon arrival of j, $z'_j = z_j$.

4.4 The Application of Online Learning

Upon arrival of a job j, the allocation process in Algorithm 2 is determined by parameters β_0, β, b. To learn the most cost-effective parameters, we apply the online learning algorithm (TOLA) in [8]. We present here its main idea; a formal description can be found in the supplementary.

There are a set of jobs $\mathcal{J}$ that arrive over time, indexed by $j = 1, 2, \cdots$, and a set of n parametric policies $\mathcal{P}$ each specified by $\{\beta_0, \beta, b\}$ and indexed by $\pi = 1, 2, \cdots$. Let $d = \max_{j \in \mathcal{J}}\{d_j\}$, i.e., the maximum relative deadline of all jobs, and $\mathcal{J}_t \subseteq \mathcal{J}$ denote all jobs j arriving at slot t, i.e., $a_j = t$. There is a weight distribution w over n policies whose initial value is $\{1/n, \cdots, 1/n\}$. Time goes from slot 1 towards later slots. At every slot t, TOLA randomly picks for a job $j \in \mathcal{J}_t$ a policy π_j from $\mathcal{P}$ according to the current w and bases the allocation of instances to j on that policy. In the meantime, the distribution w will also be updated at every $t > d$. At such t, we have the knowledge of spot prices in $[t-d, t-1]$ and can derive the cost of completing a job $j' \in \mathcal{J}_{t-d}$ under every policy $\pi \in \mathcal{P}$; the distribution is updated such that the lower-cost (higher-cost) polices of this job are re-assigned the enlarged (resp. reduced) weights. Thus, as time goes by and more and more jobs are processed, the most cost-effective policies of $\mathcal{P}$ will be identified gradually, i.e., the ones with the highest weights. When t is large, TOLA will choose the most cost-effective policy for every arriving job and the actual cost of completing all jobs is close to the cost of completing all jobs under a specific policy $\pi^* \in \mathcal{P}$ that generates the lowest total cost.

4.5 Extension to Microsoft Azure Cloud

Above, we are essentially studying the following question. On-demand instances are always available and charged a fixed unit price. The availability of spot instances is uncertain over time; intuitively, it is the probability that a user successfully gets spot instances and we denote by β its average value. There is a fixed number of self-owned instances. The costs of utilizing self-owned, spot and on-demand instances are increasing. Our question is about the cost-effective strategy to utilize these instances. Our intuition is to maximize the utilization of self-owned instances; when they are not adequate for completing a job, we aim to minimize the utilization of costly on-demand while maximizing the utilization of spot instances. The availability β and the job characteristics (deadline, workload, parallelism bound) determine the unique capability of each job j to utilize spot instances, i.e., the maximum workload that could be processed by spot instances. We also give the minimum amount r_j^{min} of self-owned instances that each job j needs to complete itself without utilizing any costly on-demand instances. Based on this, related policies are proposed to allocate instances to jobs.

So far, this question has been addressed in the context of Amazon EC2 pricing. On-demand instances are charged on an hourly basis. The price of spot instances (i.e., spot price) fluctuates over time and is updated every 5 minutes. Every user bids a price for spot instances and only if its bid price is not below the current spot price, it could utilize the spot instances for at least 5 minutes. Spot users are charged according to the spot prices. Under such context, the availability of spot instances depends on the bid price of users and the spot prices.

Beyond Amazon EC2, Microsoft Azure began to offer low-priority VMs (virtual machines) since May 2017, as well as high-priority VMs [26]; it is the second largest IaaS service provider and accounts for 13.3% of the global market share in 2017 [1]. High- and low-priority VMs respectively correspond to on-

Algorithm 2: Dynalloc($a_j, d_j, z'_j, \delta_j, \beta_0, \beta, b, N, t$)

Input : the job's current characteristics $\{a_j, d_j, z'_j, \delta_j\}$ where z'_j is still > 0, and a parameterized policy $\{\beta_0, \beta, b\}$

```
/* allocate instances at the very beginning of
   slot t                                        */
```
if $a_j = t$ **then**
  `// upon arrival of j, allocate self-owned instances to it`
  set the value of r_j using Equation (4);
  for $\bar{t} \leftarrow a_j$ **to** $a_j + d_j - 1$ **do**
    $r_j(\bar{t}) \leftarrow r_j$;
$i \leftarrow \left\lfloor \frac{t-a_j}{Len} \right\rfloor + 1$ `// used to number the allocation update`
if $\frac{t-a_j}{Len} = i - 1$ **then**
  `// at the i-th allocation of j where it has flexibility for spot instances`
  if $r_j \geq g_j(\beta)$ **then**
    `// it is expected that j will be completed by utilizing spot instances alone after allocating self-owned instances`
    apply the strategy in Proposition 4.3 here;
  else
    call Algorithm 1;
if *the spot instances of j get lost at the beginning of slot t* **then**
  if $\frac{(\delta_j - r_j) \cdot (d_j - Len \cdot i)}{z'_j} < 1$ **then**
    `// j has no flexibility to utilize spot instances at the next allocation update by Definition 3.1`
    apply the strategy in Proposition 4.7 here;
  `// otherwise, j still has the flexibility at the next allocation update where` $z'_j = z_j^{i+1}$

demand and spot instances only with some difference in pricing. In Microsoft Azure, high-priority VMs are always available and charged a fixed price; also, low-priority VMs have a lower price but their availability varies over time. However, its pricing model simplifies the Amazon EC2 pricing in that (i) high-priority VMs are charged per second instead of on an hourly basis, and (ii) the price of low-priority VMs is fixed but their availability is a system-level random variable, without depending on the bid price of users. So, we can roughly say that users will be billed for the exact period when VMs are utilized, without rounding up partial instance hour to full hour.

Now, we explain how to apply the framework of this paper to the Microsoft Azure scenario. Upon arrival of a job j, we can still use the policy (4) proposed in Section 3.3.2 for the allocation of self-owned instances: in the case that they are not adequate, the number of allocated instances is no larger than r_j^{min}; in the opposite case, the number is no smaller than r_j^{min}. After allocating self-owned instances, the job j can be viewed as a new job with a reduced parallelism bound $\delta_j - r_j$. Unlike the case in Amazon EC2, it is not necessary to update the allocation of on-demand instances and bid a price for spot instances every hour. From its arrival on, job j continuously attempts to utilize $\delta_j - r_j$ low-priority VMs; once there is no flexibility for j to utilize such VMs at some moment, j turns to utilize $\delta_j - r_j$ high-priority VMs until it is completed. In the allocation process above, only one parameter β_0 is needed to control the allocation of self-owned instances; in contrast, there are three parameters $\{\beta_0, \beta, b\}$ in the case of Amazon EC2. So, when the approach of online learning is applied here, only β_0 is needed to be learned.

5 EVALUATION

The main aim of our evaluations is to show the effectiveness of the proposed policies of this paper.

5.1 Simulation Setups

The on-demand price is $p = 0.25$ per hour. We set L to 5 (minutes) and all jobs have a parallelism bound of 20. Following [22], [23], we generate the jobs as follows. The job's arrival is generated according to a poisson distribution with a mean of 2. The size z_j of every job j is set to $12 \times 20 \times x$ where x follows a bounded Pareto distribution with a shape parameter $\epsilon = \frac{1}{1.01}$, a scale parameter $\sigma = \frac{1}{6.06}$ and a location parameter $\mu = \frac{1}{6}$; the maximum and minimum value of x is set to 1 and 10. The job's relative deadline is generated as $x \cdot z_j/\delta_j$, where x is uniformly distributed over $[1, x_0]$. x represents the slackness of a job; it affects the jobs' capability to utilize spot instances as shown by Proposition 4.5, and is a main factor that determines the performance. In this paper, we consider three types of jobs respectively with a small, medium, and large slackness: the 1st, 2nd, 3rd types of jobs respectively with $x_0 = 3, 7, 13$. Spot prices are updated every time slot and their values can follow an exponential distribution where its mean is set to 0.11 [24].

Proposed Policies. The policies of this paper are parameterized: β and b are used for determine the allocation of spot and on-demand instances (see lines 5-13 of Algorithm 2), and β_0 is for self-owned instances (see lines 1-4 of Algorithm 2). The parameter β_0 is chosen in $\mathcal{C}_1 = \{\frac{i}{10} \mid 0 \le i \le 6\}$. As illustrated in Fig. 5, for jobs with $x_0 > 1.25$, the amount of self-owned instances allocated to jobs can be effectively controlled by selecting a value ≤ 0.6; for the others with little flexibility to utilize spot instances, they will be a large number of self-owned instances whenever possible to reduce the consumption of on-demand instances. The parameter β is chosen from $\mathcal{C}_2 = \{\frac{i}{10} \mid 0 \le i \le 9\} \cup \{0.9999\}$. The bid price b is chosen in $\mathcal{B} = \{b_i = 0.13 + 0.03 \cdot (i-1) \mid 1 \le i \le 6\}$. When only spot and on-demand instances are considered, let $\boldsymbol{\mathcal{P}} = \{(\beta, b) \mid \beta \in \mathcal{C}_2, b \in \mathcal{B}\}$, representing all policies of this paper to be evaluated; when self-owned instances are also taken into account, let $\boldsymbol{\mathcal{P}} = \{(\beta, b, \beta_0) \mid \beta_0 \in \mathcal{C}_1, \beta \in \mathcal{C}_2, b \in \mathcal{B}\}$.

Compared Policies. The policies of this paper are compared with (i) the naive policy (2) for self-owned instances and (ii) the policy proposed in [8] only for spot and on-demand instances (see Algorithm 1 in [8]). The latter randomly selects a parameter $\theta \in \Theta = \{\frac{i}{10} \mid 0 \le i \le 10\}$ for every job j: (i) the user will bid a price b for $\theta \cdot \delta_j$ spot instances and acquire $(1-\theta) \cdot \delta_j$ on-demand instances at every allocation update of j; (ii) it monitors at every slot t whether there is a risk of not completing the job by its deadline if only $(1-\theta) \cdot \delta_j$ on-demand instances are utilized in the remaining slots; (iii) if such risk exists, there is no flexibility for utilizing spot instances and it turns to utilize $\min\left\{\delta_j, \left\lceil z_j^{i_j+1}/Len \right\rceil\right\}$ on-demand instances alone until j is completed[4]. Let $\boldsymbol{\mathcal{P}'} = \{(\theta, b) \mid \theta \in \Theta, b \in \mathcal{B}\}$, representing all the policies of [8].

Performance Metric. Let π denote a policy in $\mathcal{P}$ or $\mathcal{P}'$. Given a set of jobs $\mathcal{J}$ that arrive over time, our aim is to minimize the cost of completing all jobs in $\mathcal{J}$; and a main performance metric is *the average unit cost of processing jobs* when the x_2-th type of jobs are processed with x_1 self-owned instances available, i.e.,

- the ratio of the total cost of utilizing various instances to the processed workload of jobs, denoted by α_{x_1,x_2}, where $\alpha_{x_1,x_2} = \sum_{j\in\mathcal{J}} c_j(\pi) / \sum_{j\in\mathcal{J}} z_j$.

When a policy in $\mathcal{P}$ or $\mathcal{P}'$ is applied to process all jobs, we denote by $\alpha_{x_1,x_2}(\pi)$ the corresponding average unit cost of processing jobs. Against the unknown statistics of spot prices and job's characteristics, there are some policies in $\mathcal{P}$ or $\mathcal{P}'$ that are the most cost-effective. We use α_{x_1,x_2} (resp. α'_{x_1,x_2}) to denote the minimum of the average unit costs of our policies (resp. the policies in [8] and defined by (2)), where $x_2 = 1, 2$, e.g., $\alpha_{x_1,x_2} = \min_{\pi\in\mathcal{P}}\{\alpha_{x_1,x_2}(\pi)\}$.

The performance of the intuitive policy (2) (for self-owned instances) and the existing policy in [8] (for spot and on-demand instances) are used as *the baseline* to measure the performance of the proposed policies; so, one performance indicator can be as follows:

$$\rho_{x_1,x_2} = 1 - \frac{\alpha_{x_1,x_2}}{\alpha'_{x_1,x_2}};$$

it represents the performance improvement of the proposed policies $\mathcal{P}$ over the baseline, that is, the ratio in cost reduction. Moreover, in this paper, the online learning algorithm TOLA is run to actually select a policy for each arriving job. The selection is random according to a distribution that will be updated according to the cost of completing that job; after numerous jobs are processed, the policies that generate the lowest cost will be associated with the highest probability. In this case, we use $\overline{\alpha}_{x_1,x_2}(\mathcal{P})$ or $\overline{\alpha}_{x_1,x_2}(\mathcal{P}')$ to denote the average unit cost of processing jobs when $\mathcal{P}$ or $\mathcal{P}'$ is applied to TOLA. When online learning is applied, the performance indicator can be as follows:

4. In [8], the workload of j is measured in instance hours.

$$\overline{\rho}_{x_1,x_2} = 1 - \frac{\overline{\alpha}_{x_1,x_2}(\mathcal{P})}{\overline{\alpha}_{x_1,x_2}(\mathcal{P}')};$$

it represents the ratio in cost reduction when online learning is applied.

5.2 Results

In the following, we give the results of simulations that are taken over about 60000 jobs, mainly listed in Tables 2, 4, 7, and 8. In our simulations, all fractional solutions will be rounded up to the nearest integers.

Experiment 1. We aim to evaluate the effectiveness of the proposed policies $\mathcal{P}$ for spot and on-demand instances alone by means of comparisons with the policies $\mathcal{P}'$ in [8], where $x_1 = 0$. The simulation results are listed in Table 2 and show a noticeable cost reduction by up to 64.51%.

TABLE 2
Performance Improvements for Spot and On-Demand Instances

$\rho_{0,1}$	$\rho_{0,2}$	$\rho_{0,3}$
58.87%	60.84%	64.51%

There are a total of 66 policies in $\mathcal{P}$. In our simulations, every 11 policies are grouped together and they use the same bid price. We have in the same group of policies that the cost-optimal value of β (denoted by β^*) is the same even under different types of jobs; the particular results are illustrated in Table 3. So, in the rest of our simulations, the effective range of β will be defined in $\{0.5, 0.6, 0.7, 0.8, 0.9, 0.999999\}$, to which we reset the value of $\mathcal{C}_2$.

TABLE 3
The Optimal β under a Bid Price b

b	0.13	0.16	0.19	0.22	0.25	0.28
β	0.7	0.8	0.9	0.9	0.999999	0.999999

Experiment 2. We aim to evaluate the proposed policy for self-owned instances, compared with the naive policy in (2); here, the allocation of spot and on-demand instances will use the same policy $\mathcal{P}$ proposed in this paper. The simulation results are listed in Table 4, showing a noticeable cost reduction by up to 43.74%.

TABLE 4
Performance Improvement for Self-Owned Instances

	ρ_{200,x_2}	ρ_{400,x_2}	ρ_{600,x_2}	ρ_{800,x_2}
$x_2 = 1$	15.73%	21.41%	27.07%	22.83%
$x_2 = 2$	27.25%	39.59%	34.04%	17.85%
$x_2 = 3$	33.05%	34.41%	43.74%	31.88%

The utilizations of self-owned instances under different policies are illustrated in Fig. 9, where the dotted lines from top to down respectively represents the case where $x_1 = 200, 400, 600$ and 800; the particular results are given by the stars on the same dotted line. The allocation of self-owned instances are determined by the policy (4) or (2). Given a set of jobs, the utilization of self-owned instances under the policy (4) only depends on the parameter β_0 since their allocation is before and independent of the allocation of spot and on-demand instances. The intuitive policy (2) is a special form of the policy (4) when $\beta_0 = 0$. In the case that $x_2 = 2$, when $x_1 = 200, 400, 600, 800$, the minimum average unit cost is generated when $\beta = 0.3, 0.2, 0.2, 0.1$ respectively; the corresponding utilizations are given in Table 5; the utilization of the intuitive policy (2) is illustrated in Table 6. We can see that, given a case of x_1 and x_2, the proposed policy achieves a lower utilization than the intuitive policy; even so, it still achieves a lower average unit cost as shown in Table 4 where $x_2 = 2$. This is because the proposed policy could effectively reduce the unnecessary consumption of on-demand instances as explained in Section 4.1.3.

TABLE 5
The Instance Utilization of the Proposed Policy under Cost-Optimal β_0

(β_0, x_1)	(0.3, 200)	(0.2, 400)	(0.2, 600)	(0.1, 800)
Utilization	89.89%	92.41%	72.70%	96.39%

TABLE 6
The Instance Utilization of the Intuitive Policy

x_1	200	400	600	800
Utilization	99.73%	99.57%	99.31%	98.89%

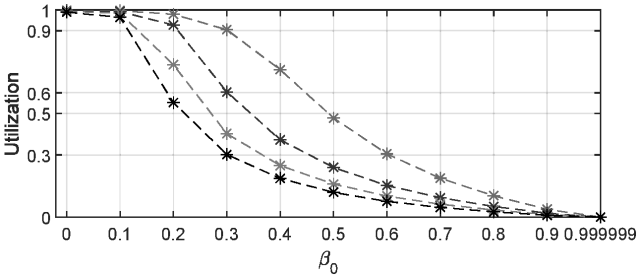

Fig. 9. The utilization of self-owned instances under different values of β_0.

Experiment 3. Assume that there are some amount of self-owned instances, and we show the performance improvement of the proposed policies $\mathcal{P}$, compared with the policies that use $\mathcal{P}'$ for spot and on-demand instances and (2) for self-owned instances. The simulation is done under the 2nd type of jobs that have a medium slackness, and the results are listed in Table 7, showing the improvement of performance by up to 75.68%.

TABLE 7
Performance Improvement for Three Types of Instances

$\rho_{200,2}$	$\rho_{400,2}$	$\rho_{600,2}$	$\rho_{800,2}$
71.30%	75.68%	72.83%	66.65%

Experiment 4. Now, we show the performance of the proposed policies when online learning is applied. The simulation setting is the same as Experiment 3. The related results are illustrated in Table 8, showing a cost reduction by up to 66.71%.

TABLE 8
Performance Improvement under Online Learning

$\overline{\rho}_{0,2}$	$\overline{\rho}_{200,2}$	$\overline{\rho}_{400,2}$	$\overline{\rho}_{600,2}$	$\overline{\rho}_{800,2}$
60.89%	63.28%	66.71%	63.60%	51.11%

6 Concluding Remark

Utilizing IaaS clouds cost-effectively is an important concern for all users. In this paper, we consider the problem of how to

utilize different purchase options including spot and on-demand instances, in addition to possibly existing self-owned instances, to minimize the cost of processing all incoming jobs while respecting their response-time targets. Driven by the goal of maximizing the utilization of self-owned instances while optimizing the possibility of utilizing spot instances, we answer two underlying questions in the instance allocation process: to be cost-effective, what properties should be kept in the policy for allocating self-owned instances and what policy can maximize the utilization of spot instances, escaping unnecessary consumption of costly on-demand instances.

As a result, we propose parametric policies for the allocation of these three types of instances that achieve small costs. The proposed policies are adaptive and, facing the dynamic of cloud market, these policies use online learning to infer the optimal values of their parameters. Through numerical simulations, we show the effectiveness of our proposed policies, in particular that they achieve a cost reduction of up to 64.51% when spot and on-demand instances are considered and of up to 43.74% when self-owned instances are considered. In future, we will extend the framework of this paper to process precedence-constrained jobs.

Note that, in our paper, we have not considered the possibility that if a job allocated to a spot instance finishes before the end of the hour, the spot instance could be re-allocated to another job for the rest of the hour rather than being terminated by the tenant. That could possibly reduce the cost further although it would significantly complicate the allocation.

ACKNOWLEDGMENTS

The work of Patrick Loiseau was supported by the French National Research Agency (ANR) through the Investissements davenir program (ANR-15-IDEX- 02), and by the Alexander von Humboldt Foundation. Part of Xiaohu Wu's work was done when he was with Eurecom, Sophia-Antipolis, France; in addition, his work was also supported by the European Union's Horizon 2020 research and innovation programme in the ROMA project (grant no. 754514). The work of Esa Hyytiä was supported by the Academy of Finland in the FQ4BD project (grant no. 296206).

REFERENCES

[1] "Gartner Says Worldwide IaaS Public Cloud Services Market Grew 29.5 Percent in 2017." https://www.gartner.com/en/newsroom/ press-releases/2018-08-01-gartner-says-worldwide-iaas-public-cloud-services-market-grew-30-percent-in-2017 (accessed on February 26, 2019).

[2] Dinesh Kumar, Gaurav Baranwal, Zahid Raza, Deo Prakash Vidyarthi. "A Survey on Spot Pricing in Cloud Computing." Journal of Network and Systems Management 26, no. 4 (2018): 809-856.

[3] "Amazon EC2 pricing." https://aws.amazon.com/ec2/pricing/ (accessed on February 26, 2019).

[4] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, Dan Tsafrir. "Deconstructing Amazon EC2 Spot Instance Pricing." *ACM Transactions on Economics and Computation*, 2013.

[5] Thilina Gunarathne, Tak-Lon Wu, Judy Qiu, and Geoffrey Fox. "Cloud computing paradigms for pleasingly parallel biomedical applications." *In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*, pp. 460-469. ACM, 2010.

[6] Geoffrey C. Fox. "Data intensive applications on clouds." In Proceedings of the second international workshop on Data intensive computing in the clouds, pp. 1-2. ACM, 2011.

[7] Navendu Jain, Ishai Menache, Ohad Shamir. "Allocation of Computational Resources with Policy Selection." *U.S. Patent* 9,652,288, issued May 16, 2017.

[8] Ishai Menache, Ohad Shamir, Navendu 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)*. USENIX Association, 2014.

[9] Navendu Jain, Ishai Menache, Joseph Naor, Jonathan Yaniv. "Near-Optimal Scheduling Mechanisms for Deadline-Sensitive Jobs in Large Computing Clusters." *ACM Transactions on Parallel Computing*, 2015.

[10] Xiaohu Wu, Patrick Loiseau. "Algorithms for scheduling deadline-sensitive malleable tasks." *In Proceedings of 2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton'15)*. IEEE, 2015.

[11] Viswanath Nagarajan, Joel Wolf, Andrey Balmin, Kirsten Hildrum. "Flowflex: Malleable scheduling for flows of mapreduce jobs." In Proceedings of the ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (MiddleWare'13), pp. 103-122. Springer, 2013.

[12] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, Rodrigo Fonseca. "Jockey: Guaranteed Job Latency in Data Parallel Clusters." *In Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, 2012.

[13] Murtaza Zafer, Yang Song, Kang-Won Lee. "Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs." *In Proceedings of the IEEE 8th International Conference on Cloud Computing (CLOUD'12)*. IEEE, 2012.

[14] Min Yao, Peng Zhang, Yin Li, Jie Hu, Chuang Lin, Xiang Yang Li. "Cutting Your Cloud Computing Cost for Deadline-Constrained Batch Jobs." *In Proceedings of the IEEE International Conference on Web Services (ICWS'14)*. IEEE, 2014.

[15] Sunilkumar S. Manvi and Gopal Krishna Shyam. "Resource Management for Infrastructure as a Service (IaaS) in Cloud Computing: A Survey." *Journal of Network and Computer Applications (Elsevier)*, 2014.

[16] Yu-Ju Hong, Jiachen Xue, Mithuna Thottethodi. "Dynamic Server Provisioning to Minimize Cost in an IaaS Cloud." *In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'11)*. ACM, 2011.

[17] Sivadon Chaisiri, Bu-Sung Lee, Dusit Niyato. "Optimization of Resource Provisioning Cost in Cloud Computing." *IEEE Transactions on Services Computing*, 2012.

[18] Wei Wang, Baochun Li, Ben Liang. "Optimal Online Multi-Instance Acquisition in IaaS Clouds." *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[19] Alexandra Vintila, Ana-Maria Oprescu, Thilo Kielmann. "Fast (Re-) Configuration of Mixed On-demand and Spot Instance Pools for High-Throughput Computing." In *ACM Workshop on Optimization Techniques for Resources Management in Clouds*, 2013.

[20] Shengkai Shi, Chuan Wu, Zongpeng Li. "Cost-Minimizing Online VM Purchasing for Application Service Providers with Arbitrary Demands." *In Proceedings of the IEEE 8th International Conference on Cloud Computing (CLOUD'15)*. IEEE, 2015.

[21] Longbo Huang, Xin Liu, Xiaohong Hao. "The Power of Online Learning in Stochastic Network Optimization." *In Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics'14)*. ACM, 2014.

[22] Junliang Chen, Chen Wang, Bing Bing Zhou, Lei Sun, Young Choon Lee, and Albert Y. Zomaya. "Tradeoffs Between Profit and Customer Satisfaction for Service Provisioning in the Cloud." *In Proceedings of the 20th ACM Symposium on High performance Distributed Computing (HPDC'11)*. ACM, 2011.

[23] Liang Zheng, Carlee Joe-Wong, Christopher G. Brinton, Chee Wei Tan, Sangtae Ha, Mung Chiang. "On the Viability of a Cloud Virtual Service Provider." *In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'16)*. ACM, 2016.

[24] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, Xinyu Wang. "How to Bid the Cloud." *In the Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*. ACM, 2015.

[25] Xiaohu Wu, Patrick Loiseau, and Esa Hyytiä. "Towards designing cost-optimal policies to utilize IaaS clouds with online learning." *In Proceedings of 2017 International Conference on Cloud and Autonomic Computing (ICCAC'17)*, pp. 160-171. IEEE, 2017.

[26] "Low-priority VMs in Batch." https://azure.microsoft.com/en-us/pricing/details/batch/ (accessed on February 28, 2019).