On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud

Ishai Menache, ***Microsoft Research;*** **Ohad Shamir,** ***Weizmann Institute;***
Navendu Jain, ***Microsoft Research***

https://www.usenix.org/conference/icac14/technical-sessions/presentation/menache

This paper is included in the Proceedings of the 11th International Conference on Autonomic Computing (ICAC '14).

June 18–20, 2014 • Philadelphia, PA

ISBN 978-1-931971-11-9

On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud

Ishai Menache
Microsoft Research

Ohad Shamir
Weizmann Institute

Navendu Jain
Microsoft Research

Abstract

Cloud computing provides an attractive computing paradigm in which computational resources are rented on-demand to users with zero capital and maintenance costs. Cloud providers offer different pricing options to meet computing requirements of a wide variety of applications. An attractive option for batch computing is *spot-instances*, which allows users to place bids for spare computing instances and rent them at a (often) substantially lower price compared to the fixed *on-demand* price. However, this raises three main challenges for users: how many instances to rent at any time? what type (on-demand, spot, or both)? and what bid value to use for spot instances? In particular, renting on-demand risks high costs while renting spot instances risks job interruption and delayed completion when the spot market price exceeds the bid. This paper introduces an online learning algorithm for resource allocation to address this fundamental tradeoff between computation cost and performance. Our algorithm dynamically adapts resource allocation by learning from its performance on prior job executions while incorporating history of spot prices and workload characteristics. We provide theoretical bounds on its performance and prove that the average *regret* of our approach (compared to the best policy in hindsight) vanishes to zero with time. Evaluation on traces from a large datacenter cluster shows that our algorithm outperforms greedy allocation heuristics and quickly converges to a small set of best performing policies.

1 Introduction

This paper presents an online learning approach that allocates resources for executing batch jobs on cloud platforms by adaptively managing the tradeoff between the cost of renting compute instances and the user-centric utility of finishing jobs by their specified due dates. Cloud computing is revolutionizing computing as a service due to its cost-efficiency and flexibility. By allowing multiplexing of large resources pools among users, the cloud enables *agility*—the ability to dynamically scale-out and scale-in application instances across hosting servers. Major cloud computing providers include Amazon EC2, Microsoft's Windows Azure, Google AppEngine, and IBM's Smart Business cloud offerings.

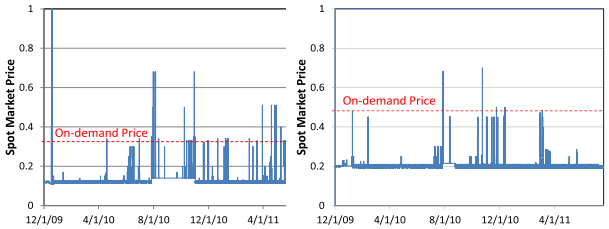

Figure 1: The variation in Amazon EC2 spot market prices for 'large' computing instances in the US East-coast region: Linux (left) and Windows (right). The fixed on-demand price for Linux and Windows instances is 0.34 and 0.48, respectively.

The common cloud pricing schemes are (i) *reserved*, (ii) *on-demand*, and (iii) *spot*. Reserved instances offer users to make a one-time payment for reserving instances over 1-3 years and then receive discounted hourly pricing on usage. On-demand instances allow users to pay for instances by the hour without any long-term commitment. Spot instances, offered by Amazon EC2, allow users to bid for spare instances and to run them as long as their bid price is above the spot market price. For *batch applications* with flexibility on when they can run (e.g., Monte Carlo simulations, software testing, image processing, web crawling), renting spot instances can significantly reduce the execution costs. Indeed, several enterprises claim to save 50%-66% in computing costs by using spot instances over on-demand instances, or their combination [3].

Reserved instances are most beneficial for hosting long running services (e.g., web applications), and may

also be used for batch jobs, especially if future load can be predicted [19]. The focus of this work, however, is on managing the choice between on-demand and spot instances, which are suitable for batch jobs that perform computation for a bounded period. Customers face a fundamental challenge of how to combine on-demand and spot instances to execute their jobs. On one hand, always renting on-demand incurs high costs. On the other hand, spot instances with a low bid price risks high delay before the job gets started (till the bid is accepted), or frequent interruption during its execution (when the spot market price exceeds the bid). Figure 1 shows the variation in Amazon EC2 spot prices for their US east coast region for Linux and Windows instances of type 'large'. We observe that spot market prices exhibit a significant fluctuation, and at times exceed even the on-demand price. For batch jobs requiring strict completion deadlines, this fluctuation can directly impact the result quality. For example, web search requires frequent crawling and update of search index as the freshness of this data affects the end-user experience, product purchases, and advertisement revenues [2].

Unfortunately, most customers resort to simple heuristics to address these issues while renting computing instances; we exemplify this observation by analyzing several case studies, reported on the Amazon EC2 website [3]. Litmus [16] offers testing tools to marketing professionals for their web site designs and email campaigns. Its heuristic for resource allocation is to first launch spot instances and then on-demand instances if spot instances do not get allocated within 20 minutes. Their bid price is set to be above the on-demand price to improve the probability of their bid getting accepted. Similarly, BrowserMob [7], a startup that provides website load testing and monitoring services, attempts to launch spot instances first at a low bid price. If instances do not launch within 7 minutes, it switches to on-demand. Other companies manually assign delay sensitive jobs to on-demand instances, and delay-tolerant ones to spot instances. In general, these schemes do not provide any payoff guarantees or how far do they operate from the optimal cost vs. performance point. Further, as expected, these approaches are limited in terms of explored policies, which account for only a small portion of the state space. Note that a strawman of simply waiting for the spot instances at the lowest price and purchasing in bulk risks delayed job completion, insufficient resources (due to limit on spot instances and job parallelism constraints), or both. Therefore, given fluctuating and unpredictable spot prices (Fig. 1), users do not have an effective way of reinforcing the better performing policies.

In this paper, we propose an online learning approach for automated resource allocation for batch applications, which balances the fundamental tradeoff between cloud computing costs and job due dates. Intuitively, given a set of jobs and resource allocation policies, our algorithm continuously adjusts per-policy weights based on their performance on job executions, in order to reinforce best performing policies. In addition, the learning method takes into account prior history of spot prices and characteristics of input jobs to adapt policy weights. Finally, to prevent overfitting to only a small set of policies, our approach allows defining a broad range of parameterized policy combinations (based on discussion with users and cloud operators) such as (a) rent on-demand, spot instances, or both; (b) vary spot bid prices in a predefined range; and (c) choose bid value based on past spot market prices. Note that these policy combinations are illustrative, not comprehensive, in the sense that additional parameterized families of policies can be defined and integrated into our framework. Likewise, our learning approach can incorporate other resource allocation parameters being provided by cloud platforms e.g., Virtual Machine (VM) instance type, datacenter/region.

Our proposed algorithm is based on machine learning approaches (e.g., [8]), which aim to learn good performing policies given a set of candidate policies. While these schemes provide performance guarantees with respect to the optimal policy in hindsight, they are not applicable *as-is* to our problem. In particular, they require a payoff value per execution step to measure how well a policy is performing and to tune the learning process. However, in batch computing, the performance of a policy can only be calculated after the job has completed. Thus, these schemes do not explicitly address the issue of *delay* in getting feedback on how well a particular policy performed in executing jobs. Our online learning algorithm handles bounded delay and provides formal guarantees on its performance which scales with the amount of delay and the total number of jobs to be processed.

We evaluate our algorithms via simulations on a job trace from a datacenter cluster and Amazon EC2 spot market prices. We show that our approach outperforms greedy resource allocation heuristics in terms of total payoff – in particular, the average regret of our approach (compared to the best policy in hindsight) vanishes to zero with time. Further, it provides fast convergence while only using a small amount of training data. Finally, our algorithm enables interpreting the allocation strategy of the output policies, allowing users to apply them directly in practice.

2 Background and System Model

In this section we first provide a background on the online learning framework and then describe the problem setup and the parameterized set of policies for resource

allocation.

Regret-minimizing online learning. Our online learning framework is based on the substantial body of work on learning algorithms that make repeated decisions while aiming to minimize *regret*. The regret of an algorithm is defined as the difference between the cumulative performance of the sequence of its decisions and the cumulative performance of the best fixed decision in hindsight. We present only a brief overview of these algorithms due to space constraints.

In general, an online decision problem can be formulated as a repeated game between a learner (or decision maker) and the environment. The game proceeds in rounds. In each round j, the environment (possibly controlled by an adversary) assigns a reward $f_j(a)$ to each possible action a, which is not revealed beforehand to the learner. The learner then chooses one of the actions a_j, possibly in a randomized manner. The average payoff of an action a is the average of rewards $\frac{1}{J}\sum_{j=1}^{J} f_j(a)$ over the time horizon J, and the learner's average payoff is the average received reward $\frac{1}{J}\sum_{j=1}^{J} f_j(a_j)$ over the time horizon. The average regret of the learner is defined as $\max_a \frac{1}{J}\sum_{j=1}^{J} f_j(a) - \frac{1}{J}\sum_{i=1}^{J} f_j(a_j)$, namely the difference between the average payoff of the best action and the learner's sequence of actions. The goal of the learner is to minimize the average regret, and approach the average gain of the best action. Several learning algorithms have been proposed that approach zero average regret as the time horizon J approaches infinity, even against a fully adaptive adversary [8].

Our problem of allocating between on-demand and spot instances can be cast as a problem of repeated decision making in which the resource allocation algorithm must decide in a repeated fashion over which policies to use for meeting job due dates while minimizing job execution costs. However, our problem also differs from standard online learning, in that the payoff of each policy is not revealed immediately after it is chosen, but only after some delay (due to the time it takes to process a job). This requires us to develop a modified online algorithm and analysis.

Problem Setup. Our problem setup focuses on a single enterprise whose batch jobs arrive over time. Jobs may arrive at any point in time, however job arrival is monitored every fixed time interval of L minutes e.g., $L = 5$. For simplicity, we assume that each hour is evenly divided into a fixed number of such time intervals (namely, $60/L$). We refer to this fixed time interval as a *time slot* (or *slot*); the time slots are indexed by $t = 1, 2, \ldots$.

Jobs. Each job j is characterized by five parameters: (i) Arrival slot A_j: If job j arrives at time $\in [L(t'-1), Lt')$, then $A_j = t'$. (ii) Due date $d_j \in \mathbb{N}$ (measured in hours): If the job is not completed after d_j time units since its arrival A_j, it becomes invalid and further execution yields zero value. (iii) Job size z_j (measured in CPU instance hours to be executed): Note that for many batch jobs such as parameter sweep applications and software testing, z_j is known in advance. Otherwise, a small bounded over-estimate of z_j suffices. (iv) Parallelism constraint c_j: The maximal degree of parallelism i.e., the upper bound on number of instances that can be simultaneously assigned to the job. (v) Value function: $V_j : \mathbb{N} \to \mathbb{R}_+$, which is a monotonically non-increasing function with $V_j(\tau) = 0\ \forall \tau > d_j$.

Thus, job j is described by the tuple $\{A_j, d_j, z_j, c_j, V_j\}$. The job j is said to be *active* at time slot τ if less than d_j hours have passed since its arrival A_j, and the total instance hours assigned so far are less than z_j.

Allocation updates. Each job j is allocated computing instances during its execution. Given the existing cloud pricing model of charging based on hourly boundaries, the instance allocation of each active job is updated every hour. The i-th allocation update for job j is formally defined as a triplet of the form (o_j^i, s_j^i, b_j^i). o_j^i denotes the number of assigned on-demand instances; s_j^i denotes the number of assigned spot instances and b_j^i denotes their bid values. The parallelism constraint translates to $o_j^i + s_j^i \leq c_j$. Note that a NOP decision i.e., allocating zero resources to a job, is handled by setting o_j^i and s_j^i to zero.

Spot instances. The spot instances assigned to a job operate until the spot market price exceeds the bid price. However, as Figure 1 shows, the spot prices may change unpredictably implying that spot instances can get terminated at any time. Formally, consider some job j; let us normalize the hour interval to the closed interval $[0,1]$. Let $y_j^i \in [0,1]$ be the point in time in which the spot price exceeded the i-th bid for job j; formally, $y_j^i = \inf_{y\in[0,1]}\{p_s(y) > b_j^i\}$, where $p_s(\cdot)$ is the spot price, and $y_j^i \equiv 1$ if the spot price does not exceed the bid. Then the cost of utilizing spot instances for job j for its i-th allocation is given by $s_j^i * \hat{p}_j^i$, where $\hat{p}_j^i = \int_0^{y_j^i} p_j(y)dy$, and the total amount of work carried out for this job by spot instances is $s_j^i * y_j^i$ (with the exception of the time slot in which the job is completed, for which the total amount of work is smaller). Note that under spot pricing, the instance is charged for the full hour even if the job finishes earlier. However, if the instance is terminated due to market price exceeding the bid, the user is not charged for the last partial hour of execution. Further, we assume that the cloud platform provides advance notification of the instance revocation in this scenario.[1] Finally, as in

[1] [23] studies dynamic checkpointing strategies for scenarios where customers might incur substantial overheads due to out-of-bid situation. For simplicity, we do not model such scenarios in this paper. However, we note that the techniques developed in [23] are complementary, and can be applied in conjunction to our online learning

Amazon EC2, our model allows spot instances to be persistent, in the sense that the user's bid will keep being submitted after each instance termination, until the job gets completed or the user cancels it .

On-Demand instances. The price for an on-demand instance is fixed and is denoted by p (per-unit per time-interval). As above, the instance hour is paid entirely, even if the job finishes before the end of the hourly interval.

Utility. The utility for a user is defined as the difference between the overall value obtained from executing all its jobs and the total costs paid for their execution. Formally, let T_j be the number of hours for which job j is executed (actual duration is rounded up to the next hour). Note that if the job did not complete by its lifetime d_j, we set $T_j = d_j + 1$ and allocation $a_j^{T_j} = (0,0,0)$.

The utility for job j is given by:

$$U_j(a_j^1,\ldots,a_j^{T_j}) = V_j(T_j) - \sum_{i=1}^{T_j} \{\hat{p}_j^i s_j^i + p \cdot o_j^i\} \qquad (1)$$

The overall user utility is then simply the sum of job utilities: $U(\mathbf{a}) = \sum_j U_j(a_j^1,\ldots,a_j^{T_j})$. The objective of our online learning algorithm is to maximize the total user utility.

For simplicity, we restrict attention to *deadline value functions*, which are value functions of the form $V_j(i) = v_j$, for all $i \in [1,\ldots,d_j]$ and $V_j(i) = 0$ otherwise, i.e., completing job j by its due date has a fixed positive value [12]. Note that our learning approach can be easily extended to handle general value functions.

Remark. We make an implicit assumption that a user immediately gets the amount of instances it requests if the "price is right" (i.e., if it pays the required price for on-demand instances, or if its bid is higher than market price for spot instances. In practice, however, a user might exhibit delays in getting all the required instances, especially if it requires a large amount of simultaneous instances. While we could seamlessly incorporate such delays into our model and solution framework, we ignore this aspect here in order to keep the exposition simple.

Resource Allocation Policies. Our algorithmic framework allows defining a broad range of policies for allocating resources to jobs and the objective of our online learning algorithm is to approach the performance of the best policy in hindsight. We describe the parameterized set of policies in this section, and present the learning algorithm to adapt these policies, in detail in Section 3.

For each active job, a policy takes as input the job specification and (possibly) history of spot prices, and outputs an allocation. Formally, a policy π is a mapping of the form $\pi : \mathscr{J} \times \mathbb{R}_+ \times \mathbb{R}_+ \times \mathbb{R}_+^n \to \mathscr{A}$, which for every active job j at time τ takes as input:

framework.

(i) the job specification of j: $\{A_j, d_j, z_j, c_j, V_j\}$

(ii) the remaining work for the job z_j^τ

(iii) the total execution cost C_j incurred for j up to time τ (namely, $C_j^\tau \triangleq \sum_{t'=A_j}^{\tau-1} s_j^{t'} \hat{p}_j^{t'} + p \cdot o_j^{t'}$, and

(iv) a history sequence $p_s(\cdot)$ of past spot prices.

In return, the policy outputs an allocation.

As expected, the set of possible policies define an explosively large state space. In particular, we must carefully handle all possible instance types (spot, on-demand, both, or NOP), different spot bid prices, and their exponential number of combinations in all possible job execution states. Of course, no approach can do an exhaustive search of the policy state space in an efficient manner. Therefore, our framework follows a best-effort approach to tackle this problem by exploring as many policies as possible in the *practical operating range* e.g., a spot bid price close to zero has very low probability of being accepted; similarly, bidding is futile when the spot market price is above the on-demand price. We address this issue in detail in Section 3.

An elegant way to generate this practical set of policies is to describe them by a small number of *control parameters* so that any particular choice of parameters defines a single policy. We consider two basic families of parameterized policies, which represent different ways to incorporate the tradeoff between on-demand instances and spot-instances: (1) *Deadline-Centric*. This family of policies is parameterized by a deadline threshold M. If the job's deadline is more than M time units away, the job attempts allocating only spot-instances. Otherwise (i.e., deadline is getting closer), it uses only on-demand instances. Further, it rejects jobs if they become non-profitable (i.e., cost incurred exceeds utility value) or if it cannot finish on time (since deadline value function V_j will become zero). (2) *Rate-Centric*. This family of policies is parameterized by a fixed rate σ of allocating on-demand instances per round. In each round, the policy attempts to assign c_j instances to job j as follows: it requests $\sigma * c_j$ instances on-demand (for simplicity, we ignore rounding issues) at price p. It also requests $(1-\sigma) * c_j$ spot instances, using a bid price strategy which will be described shortly. The policy monitors the amount of job processed so far, and if there is a risk of not completing the job by its due date, it switches to on-demand only. As above, it rejects jobs if they become non-profitable or if it cannot finish on time. A pseudo-code implementing this intuition is presented in Algorithm 1. The pseudo-code for the deadline-centric family is similar and thus omitted for brevity.

We next describe two different methods to set the bids for the spot instances. Each of the policies above can

use each of the methods described below: (i) *Fixed bid*. A fixed bid value b is used throughout. (ii) *Variable bid*. The bid price is chosen adaptively based on past spot market prices (which makes sense as long as the prices are not too fluctuating and unpredictable). The variable bid method is parameterized by a weight γ and a safety parameter ε to handle small price variations. At each round, the bid price for spot instances is set as the weighted average of past spot prices (where the effective horizon is determined by the weight γ) plus ε. For brevity, we shall often use the terms *fixed-bid policies* or *variable-bid policies*, to indicate that a policy (either deadline-centric or rate-centric) uses the fixed-bid method or the variable-bid method, respectively. Observe that variable bid policies represent one simple alternative for exploiting the knowledge about past spot prices. The design of more "sophisticated" policies that utilize price history, such as policies that incorporate potential seasonality variation, is left as an interesting direction for future work.

ALGORITHM 1: Ratio-centric Policy

Parameters (with Fixed-Bid method): On-demand rate $\sigma \in [0,1]$; bid $b \in \mathbb{R}_+$
Parameters (with Variable-Bid method): On-demand rate $\sigma \in [0,1]$; weight $\gamma \in [0,1]$; safety parameter $\varepsilon \in \mathbb{R}_+$
Input: Job parameters $\{d_j, z_j, c_j, v_j\}$
If $c_j * d_j < z_j$ or $p * \sigma * z_j > v_j$, drop job //Job too large or expensive to handle profitably
for Time slot t in which the job is active **do**
  If job is done, return
  Let m be the number of remaining time slots till job deadline (including the current one)
  Let r be the remaining job size
  Let q be the cost incurred so far in treating the job
  // Check if more on-demand instances needed to ensure timely job completion
  if $(\sigma + m - 1)\min\{r, c_j\} < r$ **then**
    // Check if running job just with on-demand is still worthwhile
    if $p * r + q < v_j$ **then**
      Request $\min\{r, c_j\}$ on-demand instances
    else
      Drop job
    end if
  else
    Request $\sigma * \min\{r, c_j\}$ on-demand instances
    Request $(1-\sigma) * \min\{r, c_j\}$ spot instances at price:
    • **Fixed-Bid method:** Bid Price b
    • **Variable-Bid method:** $\frac{1}{Z}\int_y p_s(y)\gamma^{\tau - y}dy + \varepsilon$, where $Z = \int_y \gamma^{\tau - y}dy$ is normalization constant
  end if
end for

Note that these policy sets include, as special cases, some simple heuristics that are used in practice [3]; for example, heuristics that place a fixed bid or choose a bid at random according to some distribution (both with the option of switching to on-demand instances at some point). These heuristics (and similar others) can be implemented by fixing the weights given to the different policies (e.g., to implement a policy which selects the bid uniformly at random, set equal weights for policies that use the fixed-bid method and zero weights for the policies that use the variable-bid method). The learning approach which we describe below is naturally more flexible and powerful, as it *adapts* the weights of the different policies based on performance. More generally, we emphasize that our framework can certainly include additional families of parameterized policies, while our focus on the above two families is for simplicity and proof of concept. In addition, our learning approach can incorporate other parameters for resource allocation that are provided by cloud platforms e.g., VM instance type, datacenter/region. At the same time, some of these parameters may be set a priori based on user constraints e.g., an 'extra-large' instance may be fixed to accommodate large working sets of an application in memory, and a datacenter may be fixed due to application data stored in that location.

3 The Online Learning Algorithm

In this section we first give an overview of the algorithm, and then describe how the algorithm is derived and provide theoretical guarantees on its performance.

Algorithm Overview. The learning algorithm pseudocode is presented as Algorithm 2. The algorithm works by maintaining a distribution over the set of allocation policies (described in Section 2). When a job arrives, it picks a policy at random according to that distribution, and uses that policy to handle the job. After the job finishes execution, the performance of each policy on that job is evaluated, and its probability weight is modified in accordance with its performance. The update is such that high-performing policies (as measured by $f_j(\pi)$) are assigned a relatively higher weight than low-performing policies. The multiplicative form of the update ensures strong theoretical guarantees (as shown later) and practical performance. The rate of modification is controlled by a step-size parameter η_j, which slowly decays throughout the algorithm's run. Our algorithm also uses a parameter d defined as an upper bound on the number of jobs that arrive during any single job's execution. Intuitively, d is a measure of the delay incurred between choosing which policy to treat a given job, till we can evaluate its performance on that job. Thus, d is closely related to job lifetimes d_j defined in Section 2. Note that while d_j is measured in time units (e.g., hours), d measures the number of new jobs arriv-

ing during a given job's execution. We again emphasize that this delay is what sets our setting apart from standard online learning, where the feedback on each policy's performance is immediate, and necessitates a modified algorithm and analysis. The running time of the algorithm scales linearly with the number of policies and thus our framework can deal with (polynomially) large sets of policies. It should be mentioned that there exist online learning techniques which can efficiently handle exponentially large policy sets by taking the set structure into account (e.g. [8], Chapter 5). Incorporating these techniques here remains an interesting direction for future work.

We assume, without loss of generality, that the payoff for each job is bounded in the range $[0,1]$. If this does not hold, then one can simply feed the algorithm with normalized values of the payoffs $f_i(j)$. In practice, it is enough for the payoffs to be on the order of ± 1 on average for the algorithm to work well, as shown in our experiments in Section 4.

ALGORITHM 2: Online Learning Algorithm

Input: Set of n policies π parameterized by $\{1,\ldots,n\}$, upper bound d on jobs' lifetime
Initialize $\mathbf{w}_1 = (1/n, 1/n, \ldots, 1/n)$
for $j = 1,\ldots,J$ **do**
  Receive job j
  Pick policy π with probability $w_{j,\pi}$, and apply to job j
  if $j \leq d$ **then**
    $\mathbf{w}_{j+1} := \mathbf{w}_j$
  else
    $\eta_j := \sqrt{2\log(n)/d(j-d)}$
    for $\pi = 1,\ldots,n$ **do**
      Compute $f_j(\pi)$ to be the utility for job $j-d$, assuming we used policy π
      $w_{j+1,\pi} := w_{j,\pi} \exp\left(\eta_j f_j(\pi)\right)$
    end for
    for $\pi = 1,\ldots,n$ **do**
      $w_{j+1,\pi} := w_{j+1,\pi} / \sum_{r=1}^{n} w_{j+1,r}$
    end for
  end if
end for

Derivation of the Algorithm. Next we provide a formal derivation of the algorithm as well as theoretical guarantees. The setting of our learning framework can be abstracted as follows: we divide time into rounds such that round j starts when job j arrives. At each such round, we make some choice on how to deal with the arriving job. The choice is made by picking a policy π_j from a fixed set of n policies, which will be parameterized by $\{1,\ldots,n\}$. However, initially, we do not know the utility of our policy choice as future spot prices are unknown. We can eventually compute this utility in retrospect, but only after $\leq d$ rounds have elapsed and the relevant spot prices are revealed.

Let $f_j(\pi_{j-d})$ denote the utility function of the policy choice π_{j-d} made in round $j-d$. Note that according to our model, this function can be evaluated given the spot prices till round j. Thus, $\sum_{j=1+d}^{J+d} f_j(\pi_{j-d})$ is our total payoff from all the jobs we handled. We measure the algorithm's performance in terms of *average regret* with respect to any fixed choice in hindsight, i.e.,

$$\max_{\pi} \frac{1}{J} \sum_{j=1+d}^{J+d} f_j(\pi) - \frac{1}{J} \sum_{j=1+d}^{J+d} f_j(\pi_{j-d}).$$

Generally speaking, online learning algorithms attempt to minimize this regret, and ensure that as J increases the average regret converges to 0, hence the algorithm's performance converges to that of the single best policy in hindsight. A crucial advantage of online learning is that this can be attained without *any* statistical assumptions on the job characteristics or the price fluctuations.

When $d = 0$, this problem reduces to the standard setting of online learning, where we immediately obtain feedback on the chosen policy's performance. However, as discussed in Section 1, this setting does not apply here because the function f_j does not depend on the learner's current policy choice π_j, but rather on its choice at an earlier round, π_{j-d}. Hence, there is a delay between the algorithm's decision and feedback on the decision's outcome.

Our algorithm is based on the following randomized approach. The learner first picks an n-dimensional distribution vector $\mathbf{w}_1 = (1/n,\ldots,1/n)$, whose entries are indexed by the policies π. At every round j, the learner chooses a policy $\pi_j \in \{1,\ldots,n\}$ with probability w_{j,π_j}. If $j \leq d$, the learner lets $\mathbf{w}_{j+1} = \mathbf{w}_j$. Otherwise it updates the distribution according to

$$w_{j+1,\pi} = \frac{w_{j,\pi} \exp\left(\eta_j f_j(\pi)\right)}{\sum_{\pi=1}^{n} w_{j,i} \exp\left(\eta_j f_j(i)\right)},$$

where η_j is a step-size parameter. Again, this form of update puts more weight to higher-performing policies, as measured by $f_j(\pi)$.

Theoretical Guarantees. The following result quantifies the regret of the algorithm, as well as the (theoretically optimal) choice of the step-size parameter η_j. This theorem shows that the average regret of the algorithm scales with the jobs' lifetime bound d, and decays to zero with the number of jobs J. Specifically, as J increases, the performance of our algorithm converges to that of the best-performing policy in hindsight. This behavior is to be expected from a learning algorithm, and crucially, occurs without any statistical assumptions on the jobs characteristics or the price fluctuations. The performance also depends - but very weakly - on the size

n of our set of policies. From a machine learning perspective, the result shows that the multiplicative-update mechanism that we build upon can indeed be adapted to a delayed feedback setting, by adapting the step-size to the delay bound, thus retaining its simplicity and scalability.

Theorem 1 *Suppose (without loss of generality) that f_j for all $j = 1,\ldots,J$ is bounded in $[0,1]$. For the algorithm described above, suppose we pick $\eta_j = \sqrt{1\log(n)/2d(j-d)}$. Then for any $\delta \in (0,1)$, it holds with probability at least $1-\delta$ over the algorithm's randomness that*

$$\max_{\pi} \frac{1}{J}\sum_{j=1}^{J} f_j(\pi) - \frac{1}{J}\sum_{j=1}^{J} f_j(\pi_{j-d}) \;\leq\; 9\sqrt{\frac{2d\log(n/\delta)}{J}}.$$

The proof of the theorem is omitted here due to space constraints, and can be found in [18].

4 Evaluation

In this section we evaluate the performance of our learning algorithm via simulations on synthetic job data as well as a real dataset from a large batch computing cluster. The benefits of using synthetic datasets is that it allows the flexibility to evaluate our approach under a wide range of workloads. Before continuing, we would like to emphasize that the contribution of our paper is beyond the design of particular sets of policies - there are many other policies which can potentially be designed for our task. What we provide is a meta-algorithm which can work on any possible policy set, and in our experiments we intend to exemplify this on plausible policy sets which can be easily understood and interpreted.

Throughout this section, the parameters of the different policies are set such that the entire range of plausible policies is covered (with limitation of discretization). For example, the spot-price time series in Section 4.2 ranges between 0.12 and 0.68 (see Fig. 6(a)). Accordingly, we allow the fixed bids b to range between 0.15 and 0.7 with 5 cents resolution. Higher than 0.7 bids perform exactly as the 0.7 bid, hence can be excluded; bids of 0.1 or lower will always be rejected, hence can be excluded as well.

4.1 Simulations on Synthetic Data

Setup: For all the experiments on synthetic data, we use the following setup. Job arrivals are generated according to a Poisson distribution with mean 10 minutes; job size z_j (in instance-hours) is chosen uniformly and independently at random up to a maximum size of 100, and the parallelism constraint c_j was fixed at 20 instance-hours. Job values scale with the job size and the instance prices. More precisely, we generate the value as $x*p*z_j$, where x is a uniform random variable in $[0.5,2]$, and p is the on-demand price. Similarly, job deadlines also scale with size and are chosen to be $x*z_j/c_j$, where x is uniformly random on $[1,2]$. As discussed in Section 3, the on-demand and spot prices are normalized (divided by 10) to ensure that the average payoff per job is on the order of ± 1. The on-demand price is 0.25 per hour, while spot prices are updated every 5 minutes (the way we generate spot prices varies across experiments).

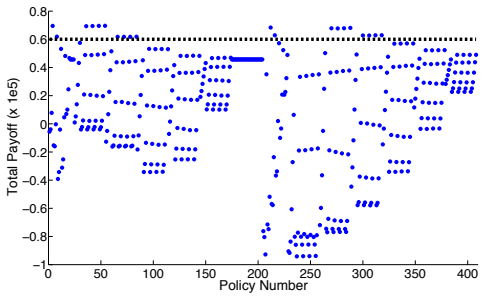

Figure 2: Total payoff for processing 20k jobs across each of the 408 resource allocation policies (while algorithm's payoff is shown as a dashed black line). The first 204 policies are rate-centric, and the last 204 policies are deadline-centric.

Resource allocation policies. We generate a parameterized set of policies. Specifically, we use 204 deadline-centric policies, and a same number of rate-centric policies. These policy set uses six values for M ($M \in \{0,\ldots,5\}$) and σ ($\sigma \in \{0,0.2,0.4,0.6,0.8,1\}$), respectively.

For either policy set, we have policies that use the fixed-bid method ($b \in \{0.1,0.15,0.2,0.25\}$), and policies that use the variable-bid method (weight $\gamma \in \{0,0.2,0.4,0.6,0.8\}$, and safety parameter $\varepsilon \in \{0,0.02,0.04,0.06,0.08,0.1\}$).

Simulation results: *Experiment 1*. In the first experiment, we compare the total payoff across 10k jobs of all the 408 policies to our algorithm. Spot prices are chosen independently and randomly as $0.15+0.05x$, where x is a standard Gaussian random variable (negative values were clipped to 0). The results presented below pertain to a single run of the algorithm, as they were virtually identical across independent runs. Figure 2 shows the total payoff for the 408 policies for this dataset. The first 204 policies are rate-centric policies, while the remaining 204 are deadline-centric policies. The performance of our algorithm is marked using dashed line. As can be seen, our algorithm performs close to the best policies in hindsight. Further, it is interesting to note that we have both deadline-centric and rate-centric policies among the best policies, indicating that one needs to consider both sets as candidate policies.

We perform three additional experiment with similar

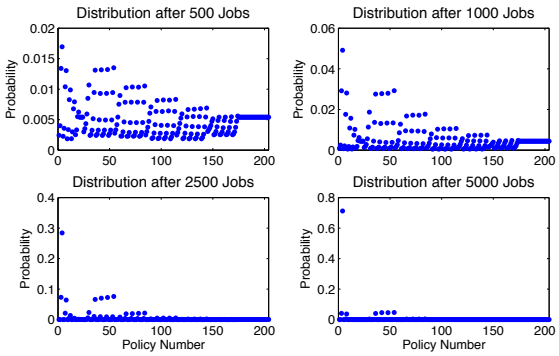

Figure 3: Evaluation under stationary spot-price distribution (mean spot price of 0.1): Probability assigned per policy after executing 500, 1000, 2500 and 5000 jobs.

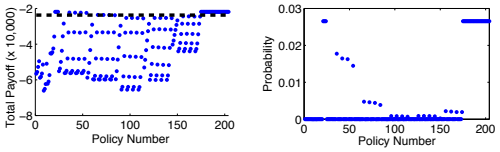

Figure 4: Evaluation under non-stationary distribution (mean spot price of 0.2): (a) Total payoff for executing 10k jobs across each of the 204 policies (while algorithm's payoff is shown as a dashed black line) and (b) the final probability assigned per policy by our learning algorithm.

setup to the above, in order to obtain insights on the properties and inner-working of the algorithm. To be able to dive deeper into the analysis, we use only the 204 rate-centric policies. The only element that we modify across experiments is the statistical properties of the spot-prices sequence.

Experiment 2. Spot prices are generated as above, except that we use 0.1 as their mean (opposed to 0.2 above). After executing 1000 jobs, our algorithm performs close to that of the best policy as it assigns probability close to 1 for that policy, while outperforming 199 out of total 204 policies. Further, its average regret is only 1.3 as opposed to 7.5 on average across all policies. Note that the upper bound on the delay in this experiment is $d = 66$, i.e., up to 66 jobs are being processed while a single job finishes execution. This shows that our approach can handle significant delay in getting feedback, while still performing close to the best policy.

In this experiment, the best policy in hindsight uses a fixed-bid of 0.25. This can be explained by considering the parameters of our simulation: since the on-demand price is 0.25 and the spot price is always relatively lower, a bid of 0.25 always yields allocation of spot instances for the entire hour. This result also highlights the easy interpretation of the resource allocation strategy of the best policy. Figure 3 shows the probability assignment for each policy over time by our algorithm after executing 500, 1000, 2500 and 5000 jobs. We observe that as the number of processed jobs increase, our algorithm provides performance close to the best policy in hindsight.

Experiment 3. In the next experiment, the spot prices is set as above for the first 10% of the jobs, and then the mean is increased to 0.2 (rather than 0.1) during the execution of the last 90% jobs. This setup corresponds to a non-stationary distribution: a learning algorithm which simply attempts to find the best policy at the beginning and stick to it, will be severely penalized when the dynamics of spot prices change. Figure 4 shows the evaluation results. We observe that our online algorithm is able to adapt to changing dynamics and converges to a probability weight distribution different from the previous setting; Overall, our algorithm attains an average regret of only 0.5, as opposed to 4.8 on average across 204 baseline policies.

Note that in this setting, the best policies are those which rely purely on on-demand instances instead of spot instances. This is expected because the spot prices tend to be only slightly lower than the on-demand price, and their dynamic volatility make them unattractive in comparison. This result demonstrates that there are indeed scenarios where the dilemma between choosing on-demand vs. spot instances is important and can significantly impact performance, and that no single instance type is always suitable.

Experiment 4. This time we set the spot price to alternate between 0.3 for one hour and then zero in the next. This variation is favorable for variable-bid policies with small γ, which use a small history of spot prices to determine their next bid. Such policies quickly adapt when the spot price drops. In contrast, fixed-bid policies and variable-bid policies with large γ suffer, as their bid price is not sufficiently adaptive. Figure 5 shows the results. We find that the group of highest-payoff policies are those for which $\gamma = 0$ i.e., they use the last spot price to choose a bid for the current round, and thus quickly adapt to changing spot prices. Further, our algorithm quickly detects and adapts to the best policies in this setting. The average regret obtained by our algorithm is 0.8 compared to 4.5 on average for our baseline policies. Moreover, the algorithm's overall performance is better than 192 out of 204 policies.

4.2 Evaluation on Real Datasets

Setup: *Workload data*. We use job traces from a large batch computing cluster for two days consisting of about 600 MapReduce jobs. Each MapReduce job comprises multiple phases of execution where the next phase can

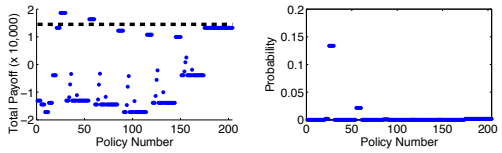

Figure 5: Evaluation under highly dynamic distribution (hourly spot prices alternate between 0.3 and zero): (a) Total payoff for processing 10k jobs across each of the 204 policies (algorithm's payoff is shown as a dashed black line), and (b) the final probability assigned per policy by our learning algorithm.

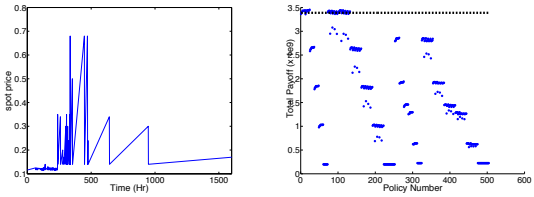

Figure 6: Evaluation on real dataset: (a) Amazon EC2 spot pricing data (subset of data from Figure 1) for Linux instances of type 'large'. The fixed on-demand price is 0.34; (b) Total payoff for processing 20k jobs across each of the 504 resource allocation policies (while algorithm's payoff is shown as a dashed black line)

start only after all tasks in the previous phase have completed. The trace includes the runtime of the job in server CPU hours ($totCPUHours$), the total number of servers allocated to it ($totServers$) and the maximum number of servers allocated to a job per phase ($maxServersPerPhase$). Since our job model differs from the MapReduce model in terms of phase dependency, we construct the parallelism constraint from the trace as follows: since the average running time of a server is $\frac{totCPUHours}{totServers}$, we set the parallelism bound c_j for each job to be $c_j = maxServersPerPhase * \frac{totCPUHours}{totServers}$. Note that this bound is in terms of CPU hours as required. Since the deadline values per job are not specified, we use the job completion time as its deadline. For assigning values per job, we generate them using the same approach as for synthetic datasets. Specifically, we assign a random value for each job j equal to its total size (in CPU hours) times the on-demand price times $B = (\alpha + N_j)$ where $\alpha = 5$ and $N_j \in [0,1]$ is drawn uniformly at random. The job trace is replicated to generate 20k jobs.

Spot Prices. We use a subset of the historical spot price from Amazon EC2 as shown in Figure 1 for 'large' Linux instances. Figure 6(a) shows the selected sample of spot price history showing significant price variation over time. Intuitively, we expect that overall that policies that use a large ratio of spot instances will perform better since on average, the spot price is about half of the on-demand price.

Resource Allocation Prices. We generated a total of 504 policies, half rate-centric and half deadline-centric. In each half, the first 72 are fixed-bid policies (i.e. policies that use the fixed-bid method) in increasing order of (on-demand rate, bid price). The remaining 180 variable-bid policies are in increasing order of (on-demand rate, weight, safety parameter). The possible values for the different parameters are as described for the synthetic data experiments, with the exception that we allow more options for the fixed bid price, $b \in \{0.15,0.2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.7\}$.

Evaluating our online algorithm on the real trace poses several new challenges compared to the synthetic datasets in Section 4.1. First, jobs sizes and hence their values are highly variable, to the effect that the difference in size between *small* and *large* jobs can be of six orders of magnitude. Second, spot prices can exhibit high variability, or alternatively be almost stable towards the end as exemplified in Figure 6(a).

Simulation results: Figure 6(b) shows the results for a typical run of this experiment. Notably, the payoff of our algorithm outperforms the performance of most of individual policies, and obtains comparable performance to the best individual policies (which are a subset of the rate-centric policies). We repeated the experiment 20 times, and obtained the following results: The average regret per job for our learning algorithm is 2071 ± 1143, while the average regret across policies is 70654 ± 12473. Note that the average regret of our algorithm is around 34 times better (on average) than the average regret across policies.

Figure 7 shows the evolution of policy weights over time for a typical run, until converging to final policy weights (after handling the entire 20000 jobs). We observe that our algorithm evolves from preferring a relatively large subset of both deadline-centric and rate-centric policies (at around 150 jobs) to preferring only rate-centric policies, both fixed-bid and variable-bid (at around 2000 jobs). Eventually, the algorithm converges to a single rate-centric policy with fixed bid. This behavior can be explained based on spot pricing data in Figure 6(a): Due to initially high variability in spot prices, our algorithm "alternates" between fixed-bid policies and variable-bid policies, which try to learn from past prices. However, since the prices show little variability for the remaining two thirds of the data, the algorithm progressively adapts its weight for the fixed-bid policy, which is commensurate with the almost stable pricing curve.

5 Related literature

While there exist other potential approaches to our problem, we considered an online learning approach due to its

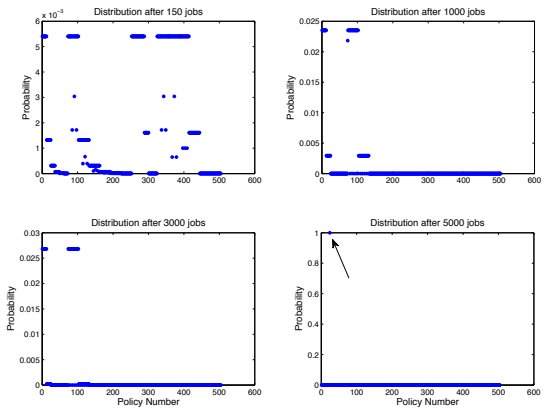

Figure 7: Evaluation on real dataset: The probability assigned per policy by our learning algorithm after processing 150, 1000, 3000 and 5000 jobs. The algorithm converges to a single policy (fixed-bid rate-centric policy) marked by an arrow.

lack of any stochastic assumptions, its online (rather than offline) nature, its capability to work on arbitrary policy sets, and its ability to adapt to delayed feedback. The idea of applying online learning algorithms for sequential decision-making tasks is well known ([9]), and there are quite a few papers which study various engineering applications (e.g., [10, 5, 11, 15]). However, these efforts do not deal with the problem of delayed feedback as it violates the standard framework of online learning. The issue of delay has been previously considered (see [14] and references therein), but are either not in the context of the online techniques we are using, or propose less-practical solutions such as running many multiple copies of the algorithm in parallel. In any case, we are not aware of any prior study of delay-tolerant online learning procedures for our application domain.

The launch of commercial cloud computing offerings has motivated the systems research community to investigate how to exploit this market for efficient resource allocation and cost reductions. Some solution concepts are borrowed from earlier works on executing jobs in multiple grids (e.g., [20] and references therein). However, new techniques are required in the cloud computing context, which directly incorporate cost considerations and a variety of instance renting options. The have been numerous works in this context dealing with different provider and customers scenarios. One branch of papers consider the auto-scaling problem, where an application owner has to decide on the right number and type of VMs to be purchased, and dynamically adapt resources as a function of changing workload conditions (see, e.g., [17, 6] and references therein).

We focus the reminder of our literature survey on cloud resource management papers that include spot instances as one of the allocation options. Some papers focus on building statistical models for spot prices which can be then used to decide when to purchase EC2 spot instances (see, e.g., [13, 1]). Similarly, [24] examines the statistical properties of customer workload with the objective of helping the cloud determine how much resources to allocate for spot instances.

In the context of large-scale batch applications, [4] proposes a probabilistic model for bidding in spot prices while taking into account job termination probabilities. However, [4] focuses on pre-computation of a fixed (non-adaptive) bid, which is determined greedily based on existing market conditions; moreover, the suggested framework does not support an automatic selection between on-demand and spot instances. [22] uses a genetic algorithm to quickly approximate the pareto-set of makespan and cost for a bag of tasks; each underlying resource configuration consists of a different mix of on-demand and spot instances. The setting in [22] is fundamentally different than ours, since [22] optimizes a global makespan objective, while we assume that jobs have individual deadlines. Finally, [21] proposes near-optimal bidding strategies for cloud service brokers that utilize the spot instance market to reduce the computational cost while maximizing the profit. Our work differs from [21] in two main aspects. First, unlike [21], our online learning framework does not require any distributional assumptions on the spot price evolution (or the job model). Second, our model may associate a different *value* and *deadline* for each job, whereas in [21] the value is only a function of job size, and deadlines are not explicitly treated.

6 Conclusion

In this paper we design and evaluate an online learning algorithm for automated and adaptive resource allocation for executing batch jobs over cloud computing platforms. Our basic model can be extended to solve other resource allocation problems in cloud domains such as renting small vs. medium vs. large instances, choosing computing regions, and different bundling options in terms of CPU, memory, network and storage. We expect that the learning framework developed here would be useful in addressing these extensions. An interesting direction for future research is incorporating reserved instances, for long-term handling of multiple jobs. This makes the algorithm stateful, in the sense that its actions affect the payoffs of policies chosen in the future. This does not accord with our current theoretical framework, but may be handled using different tools from competitive analysis.

Acknowledgements. We thank our shepherd Alexandru Iosup and the ICAC reviewers for the useful feedback.

References

[1] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation*, 1(3):16, 2013.

[2] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 63–74. ACM, 2010.

[3] AWS Case Studies. `http://aws.amazon.com/solutions/case-studies/`.

[4] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under sla constraints. In *MASCOTS*, 2010.

[5] I. Ari, A. Amer, R. Gramacy, E. Miller, S. Brandt, and D. Long. Acme: Adaptive caching using multiple experts. In *WDAS*, 2002.

[6] Y. Azar, N. Ben-Aroya, N. R. Devanur, and N. Jain. Cloud scheduling with setup cost. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 298–304. ACM, 2013.

[7] Browsermob. `https://aws.amazon.com/solutions/case-studies/Browsermob`.

[8] N. Cesa-Bianchi and G. Lugosi. *Prediction, learning, and games*. Cambridge University Press, 2006.

[9] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, 1997.

[10] R. Gramacy, M. Warmuth, S. Brandt, and I. Ari. Adaptive caching by refetching. In *NIPS*, 2002.

[11] D. Helmbold, D. Long, T. Sconyers, and B. Sherrod. Adaptive disk spin-down for mobile computers. *MONET*, 5(4):285–297, 2000.

[12] N. Jain, I. Menache, J. Naor, and J. Yaniv. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. In *SPAA*, pages 255–266, 2012.

[13] B. Javadi, R. Thulasiram, and R. Buyya. Statistical modeling of spot instance prices in public cloud environments. In *Fourth IEEE International conference on Utility and Cloud Computing*, 2011.

[14] P. Joulani, A. György, and C. Szepesvári. Online learning under delayed feedback. In *ICML*, 2013.

[15] B. Kveton, J. Y. Yu, G. Theocharous, and S. Mannor. Online learning with expert advice and finite-horizon constraints. In *AAAI*, 2008.

[16] Litmus. `https://aws.amazon.com/solutions/case-studies/Litmus`.

[17] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 49. ACM, 2011.

[18] I. Menache, O. Shamir, and N. Jain. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. Technical report, Microsoft Research, May 2014. Available from `http://research.microsoft.com/apps/pubs/default.aspx?id=217154`.

[19] S. Shen, K. Deng, A. Iosup, and D. Epema. Scheduling jobs in the cloud using on-demand and reserved instances. In *Euro-Par 2013 Parallel Processing*, pages 242–254. Springer, 2013.

[20] M. Silberstein, A. Sharov, D. Geiger, and A. Schuster. Gridbot: execution of bags of tasks in multiple grids. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 11. ACM, 2009.

[21] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. In *INFOCOM*, pages 190–198, 2012.

[22] A. Vintila, A.-M. Oprescu, and T. Kielmann. Fast (re-)configuration of mixed on-demand and spot instance pools for high-throughput computing. In *Proceedings of the first ACM workshop on Optimization techniques for resources management in clouds*, pages 25–32. ACM, 2013.

[23] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on Amazon cloud spot instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2012.

[24] Q. Zhang, E. Gürses, R. Boutaba, and J. Xiao. Dynamic resource allocation for spot markets in clouds. In *Hot-ICE*, 2011.