Abstract The recent years witnessed rapid emergence and proliferation of cloud computing. To fully utilize the compute resources, some cloud operators provide spot resources. Spot resources allow customers to bid on unused capacity. However, pricing policy of the spot resources should be carefully designed and the impact on both present and future should be considered. For the present, the cloud provider can set a higher price to gain extra revenue. For the future, higher price will shift more requests with lower prices to later time and reduce the revenue of future. Meanwhile, the quality of service should be considered either since bad QoS will incur loss of potential users.

In this paper, we present a demand curve to model the impact of pricing on the present and future revenue. Then we formulate the revenue maximization problem as a time-average optimization problem. Next, since this basic model fails to provide information of service delay, we extend it to a more generalized one that ensures the worst-case delay of user requests. While the future knowledge of arrival requests is unknown, it is necessary to design online algorithms for the optimization problems. We apply Lyapunov optimization framework and design an efficient online algorithm which does not require any future knowledge of requests arrival. Evaluations based on real-life datacenter workload and Amazon EC2 Spot Price illustrate efficiency of our algorithms.

I. INTRODUCTION

The recent years witnessed rapid emergence and proliferation of cloud computing. Cloud computing provides many benefits including easy access to user data, ease of management for users and reduction of amortized cost of ownership. One important feature of cloud computing is pay-as-you-go manner. The cloud provider sells the services or compute resources to the customers, and changes the economics of computing by allowing customers to pay only for capacity that they actually use. Therefore, the pricing of compute resources is crucial for cloud computing, and it determines the profit of cloud provider.

In order to fully utilize the compute resources, some cloud operators provide spot resources. Spot resources allow customers to bid on unused capacity and utilize these resources for as long as their bid exceeds the current spot price. Usually, the spot price is much lower than the static price. Thereby, customers can significantly reduce their costs if their application can handle the potential for interruption. An example is Amazon Elastic Compute Cloud [1], which provides Spot Instances packaged in form of Virtual Machines. The spot price fluctuates periodically depending on the supply and demand of Spot Instance capacity.

However, pricing for spot resources is a non-trivial task. The spot price influences the amount of resources can be sold, and thereby the profit of cloud provider. First, when customers choose one provider and bid for the unused capacity, this cloud provider is the sole seller of these resources. Thus, it possesses market power to determine the price and does not need to accept all requests. The provider can set a higher price and only serve part of the customers to gain extra revenue, i.e. maximizing revenue for the present. Second, the requests which is not accepted will be shifted to later time when the provider sets a higher price. These shifted demands increase the risk of setting a lower price and lowering revenue in the future. Therefore, while cloud provider sets the spot price, it should not only price to the present but also to the future. However, future pricing is another challenge due to the lack of future information of the requests. Finally, for the cloud providers, in addition to maximizing the total revenue, how to guarantee the quality of service (QoS) such as service delay to the end users is equally important. Users might go to another service provider if the pricing schema is not customer friendly. In conclusion, how to maximize the total revenue while guaranteeing QoS is an important problem.

In this paper, we investigate the problem that how cloud provider sets the spot price to maximize its revenue. We first present a demand curve model which accurately captures the characteristic of spot resources for current cloud computing. This model also takes the impact of pricing on the future into consideration. We then formulate the time-average revenue maximization problem for cloud providers. For cloud computing, the demand and supply of spot resources vary fast and the statistics of them may be unknown. Hence, it’s important to design efficient online algorithms which do not depend on the prior knowledge of the future information. To this end, we propose an efficient solution based on the recently developed technique of Lyapunov optimization [2].

The main contribution of this paper is three-fold: First, we present demand curve to model the spot instances. This model accurately captures the relationship between price
Second, we systematically investigate the pricing problem of spot resources for the cloud provider. We present two models to describe the different situation about the spot instances. The first one can achieve the required tradeoff between revenue and request queue backlog. Since this model fails to provide the information of service delay, we extend it to ensure the worst case delay guarantee. However, the second one needs to drop some requests if their price is too low. Our models do not only accurately capture the feature of the spot resources, but also consider pricing to the future.

Third, we present efficient online algorithms for both models, which only make use of the current system state values and do not require any knowledge of the statistics of the requests. They apply Lyapunov optimization to efficiently solve the time-average revenue maximization problem. We show that our algorithms provide a smooth tradeoff between the revenue and the delay of requests.

The rest of this paper is organized as follows. Section II illustrates the motivation and challenges of pricing to spot instances. Section III and IV present the basic model and online algorithm for this model, respectively. Section V extends the basic model to ensure the worst-case delay guarantee. Section VI shows the numerical results. Section VII discusses the related work and section VIII concludes the paper.

II. MOTIVATION AND CHALLENGE

The goal of this work is to maximize the time-average revenue of spot resources for the cloud service provider. We illustrate the motivation of this work using example of spot instances of Amazon Elastic Compute Cloud (EC2) [1]. Spot Instances allow customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current Spot Price. To use Spot Instances, a customer places a Spot Instance request, specifying the instance type, the Availability Zone desired, the number of Spot Instances, and the maximum price per instance hour. When the system receives the requests, it matches the requests (demand) against unused capacity (supply). It accepts the highest priced requests and then `fstacksf` down lower priced requests until all requests have been accepted or the supply has `fexhaustedf`. All accepted requests are charged the same price (Spot Price) which is based on the last request accepted. The Spot Price is adjusted periodically as requests come in and the available supply changes. However, we argue that above pricing policy which tries to meet all demand is not optimal to maximize the revenue.

First, once a user chooses EC2, Amazon is the sole seller in the market of spot instances. It can set a higher price and serve less requests to gain extra revenue. This is so-called market power in economics. (In the long term, customers might go to another service provider if the pricing schema is not friendly to the customers. That’s why we must consider the QoS.) We now show how this works with a simple example. Assume that there are 3 customers who are willing to use spot instances. Their maximum prices are 1, 0.8, and 0.5, respectively. In the first time slot, the numbers of requests are 500, 800, and 700, respectively. We assume that the supply is always sufficient to meet all demand. Table I shows the example. According to above pricing policy, the spot price is set at 0.5 and all requests are accepted. The total revenue is 1000. However, we can set the price at 0.8 to obtain more revenue. At this price, only the requests of 8 customers are accepted. The requests of the third customer remain in a pending state, i.e. they are deferred to later time. Thereby, there are 1300 requests accepted and the total revenue is 1040. The third row in Table I shows achieved revenue at different spot prices. We can see that 0.8 is the optimal price and 700 requests is deferred to later time. The numbers of deferred requests associated with the corresponding spot prices are shown in the last row.

Second, making decision in a single slot may result in suboptimal solution. The lower priced requests are deferred to later time, if we set a high spot price. This may incur lower revenue in the future. Continuing our example, the 3 customers keep placing requests in slot 2 and 3. The numbers of requests are shown in Table II. If we make decision in single slot and do not consider the future, the prices of each slot are set as follows. As discussed above, the price of fOptimal f is set at 0.8. We gain revenue of 1040 in this slot and defer 700 requests to slot 2. In slot 2, the number of requests placed by the third customer becomes 1000(700 deferred requests plus 300 new ones). The price is set at 0.5 in this slot and all requests are accepted. The revenue generated in this slot is 1250. Similarly, the price is set at 0.5 and the revenue is 250 in slot 3. As a result, the total revenue generated in 3 slots is 2540. The decisions of above policy are shown in the row named fSinglef. However, we can gain more revenue if we have full knowledge of the future. As shown in Table II, the

**Table I**  
<table>
<thead>
<tr>
<th>Slot</th>
<th>Maximum price</th>
<th>0.8</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>Number of requests</td>
<td>500</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td>Revenue</td>
<td>1040</td>
<td>1040</td>
</tr>
<tr>
<td></td>
<td>Deferred requests</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table II**  
<table>
<thead>
<tr>
<th>Slot</th>
<th>Maximum price</th>
<th>0.8</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>Number of requests</td>
<td>600</td>
<td>900</td>
</tr>
<tr>
<td></td>
<td>Revenue</td>
<td>1250</td>
<td>1250</td>
</tr>
<tr>
<td></td>
<td>Deferred requests</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Optimal</td>
<td>Number of requests</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Revenue</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Deferred requests</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Optimal</td>
<td>Number of requests</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Revenue</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Deferred requests</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
optimal prices of all slots are 0.5, 0.8, and 0.5, respectively. We can obtain a revenue of 2600 in total. Obviously, we should consider the future when we set the spot price.

However, new challenge arises considering pricing to the future. Cloud operators face time-varying demand of spot instances with possibly unknown statistics. Even if the request statistics can be learned, traditional approaches involve the use of Markov Decision Theory and Dynamic Programming. It is well known that these methods suffer from seriously state space explosion problem and the computational complexity significantly increases with the state of system. In this paper, we make use of a different approach that can overcome the challenges associated with dynamic programming. It is based on the recently developed technique of Lyapunov optimization [2] and operates without requiring any knowledge of the system statistics.

III. BASIC MODEL

In this paper, we consider a system that operates in discrete time over slots \( t \in \{1, 2, \ldots\} \). The cloud provider sells its computing resources in the form of virtual machines, named spot instances. Instances are charged the Spot Price, which is set by the provider and fluctuates periodically. Cloud operator should choose a price that maximizes its revenue. In the basic model, we assume that all requests cannot be dropped. We will relax this assumption in Section V.

A. Spot Instance and Price Model

In this paper, we consider the same type of spot instances in single data center. Since different types of spot instances are in different price systems, the algorithms presented in this paper can be generalized to multi types of instances without any changes. At the beginning of each time slot \( t \), all customers place their requests, specifying the number of instances and the maximum price (If a user sends the requests in the middle of the slot, these requests will be considered at the beginning of next slot). For simplicity, we assume that each instance is sustained for one time slot after launching. If the length of a request is more than 1 slot, we can handle this situation by repeating the request with the same number of instances and the same maximum price. Every slot \( t \), cloud operator observes the requests and determines the spot price \( P(t) \). At the same time, the number of launched instances \( N(t) \) is determined as well. \( N(t) \) is a function of \( P(t) \). We denote this by:

\[
N(t) = N(P(t)).
\]

We utilize demand curve to describe this function. Demand curve is an important economics concept which represents the relationship between price and quantity demanded. The demand curve of spot instances can be represented by a step line and a typical demand curve is illustrated in Fig.1. Based on the information of all requests, it is easy to obtain the demand curve in current slot. Sort the requests by their maximum price in a descending order. Plot a horizontal line segment related to the instances with the same maximum price: the height of the segment is the price, and the width equals to the number of requested instances with the same price. The initial polygonal line constructed by all line segments is demand curve. Note that the actual demand curve is represented by the solid line. The maximum number of instances which can be launched equals to the supply \( S(t) \). The supply represents the unused capacity in the data center and varies with time. Thus, we have for all \( t \):

\[
0 \leq N(P(t)) \leq S(t). \tag{1}
\]

We denote by \( P_S(t) \) the price while the demand equals to the supply, i.e. \( N(P_S(t)) = S(t) \). While the price drops below \( P_S(t) \), the launched instances always achieve the upper bound \( S(t) \). For notational convenience, we will use \( N(t) \) to denote the number of launched instances in the rest of the paper noting that the dependence of \( N(t) \) on \( P(t) \) is implicit. Further, we assume that the supply \( S(t) \) is upper bounded by a constant \( S_{\text{max}} \), which corresponds to the data center capacity. Then, we have the following constraint for all \( t \):

\[
0 \leq S(t) \leq S_{\text{max}}. \tag{2}
\]

According to the demand curve, we can easily obtain the number of instances that should be launched at any spot price. For example, system will launch \( D_0 \) instances if the spot price is set at \( P_0 \). As long as \( P_0 > P_S \), all requests with the maximum price \( P_0 \) will be accepted. However, while the spot price equals to \( P_S \), only part of the instances with the maximum price \( P_S \) can be launched (shown in Fig.1). It is worth noting that the demand curve varies with time. The above process of constructing the demand curve is repeated in every time slot. At the beginning of each slot, system observes the information of the requests and the supply, then obtains the demand curve in this slot.

\[
\text{Fig. 1. Typical Demand Curve}
\]

B. Revenue Maximization Problem

Once the spot price \( P(t) \) is determined, the requests with lower maximum price remain in a pending state. These requests will wait for next time slot to be accepted. Let \( Q(t) \) denote the number of instances in pending state. Then, the dynamics of \( Q(t) \) can be expressed as:

\[
Q(t + 1) = Q(t) - N(t) + A(t), \tag{3}
\]

where \( A(t) \) denotes the instances of new request arrivals in slot \( t \). The underlying probability distribution or statistical characterization of \( A(t) \) is not necessarily known. We only assume that its maximum value is \( A_{\text{max}} \), i.e. we have:

\[
0 \leq A(t) \leq A_{\text{max}}. \tag{4}
\]
For simplicity, we assume that \( A(t) \) is i.i.d. over slots. The algorithm presented in this paper can be generalized to the case that \( A(t) \) evolves according to the more general random processes by using T slot drift technique [2]. Further, for purpose of the system stability, we assume that the request arrival rate is less than the supply throughout the paper, i.e.:

\[
\mathbb{E}\{A(t)\} \leq \mathbb{E}\{S(t)\}. \tag{5}
\]

We call the set of arrival processes that satisfy above constraint capacity region. This inequality ensures that the system is schedulable. In another word, there exist a pricing policy that ensure the queue \( Q(t) \) is stable. We denote by \( Q \) the time average queue backlog for the requests and define queue stability as follows:

\[
Q_k \triangleq \lim_{t \to \infty} \frac{1}{t} \sum_{\tau=0}^{t-1} \mathbb{E}\{Q_k(\tau)\} < \infty. \tag{6}
\]

We assume that the spot price is upper bounded by a constant \( P_{\max} \). Thus, we have for all \( t \):

\[
0 \leq P(t) \leq P_{\max}. \tag{7}
\]

The goal of the cloud provider is to maximize the time-average revenue. The revenue generated in slot \( t \) is given by \( R(t) = P(t)N(t) \). Let \( P(t) \) be the control decision made in slot \( t \) under a feasible policy, which ensure constraints (6) and (7). The time-average revenue of this policy is defined as:

\[
R_{av} \triangleq \lim_{t \to \infty} \frac{1}{t} \sum_{t=0}^{t-1} \mathbb{E}\{P(t)N(t)\}. \tag{8}
\]

We are interested in finding a price policy \( P(t) \) that maximizes this time-average revenue subject to the constraints (6) and (7).

IV. ALGORITHM DESIGN AND PERFORMANCE ANALYSIS

In this section, we present our algorithm to solve the time-average revenue problem. The algorithm applies Lyapunov optimization [2] which operates without any knowledge of the future requests. We first present our algorithm, next, we derive performance bounds for the algorithm. Then we use some simple examples to get some intuition and insights about our algorithm.

A. Algorithm Design

To begin with, we define a queuing state variable:

\[
X(t) = Q(t) - A_{\max}. \tag{9}
\]

\( X(t) \) is a new queue that adds a shift on \( Q(t) \). Thus, they have the same dynamics:

\[
X(t+1) = X(t) - N(t) + A(t). \tag{10}
\]

Then, we define the Lyapunov function \( L(t) \) as follows:

\[
L(t) \triangleq \frac{1}{2}X^2(t). \tag{11}
\]

\( L(t) \) measures the queue backlog in the system. Then, we define a one-slot conditional Lyapunov drift, which represents the expected change in the Lyapunov function over one slot, given the current state \( X(t) \) in slot \( t \):

\[
\Delta(X(t)) \triangleq \mathbb{E}\{L(t+1) - L(t)\mid X(t)\}. \tag{12}
\]

If we greedily minimize the Lyapunov function \( \Delta(X(t)) \), the backlogs are consistently pushed towards a lower congestion state (i.e. stability). Further, the revenue should be maximized at the same time. Thereby, we incorporate the revenue term to (12) to obtain the drift-plus-penalty expression:

\[
\Delta(X(t)) + V\mathbb{E}\{-R(t)\mid X(t)\}. \tag{13}
\]

The basic idea of Lyapunov optimization is to minimize a bound on the drift-plus-penalty term. \( V \) is a non-negative control parameter. Choosing \( V > 0 \) includes the weighted revenue term in the control decision and allows a smooth tradeoff between backlog reduction and revenue maximization. Thus, the key is to find an upper bound on this expression. The following lemma provides the upper bound for our problem.

**Lemma 1.** Suppose the arrival requests \( A(t) \) and the supply \( S(t) \) are i.i.d. over slots. Under any control algorithm \( P(t) \), the drift-plus-penalty expression has the following upper bound:

\[
\Delta(X(t)) + V\mathbb{E}\{-R(t)\mid X(t)\}
\]

\[
\leq B + V\mathbb{E}\{-R(t)\mid X(t)\} + X(t)\mathbb{E}\{A(t) - N(t)\mid X(t)\}, \tag{14}
\]

where \( B \) is a positive constant and defined by \( B \triangleq \frac{1}{2}(A_{\max}^2 + S_{\max}^2) \).

**Proof:** This result follows from the framework in [2] and is omitted for brevity.

Rather than directly minimize the drift-plus-penalty expression (13), our algorithm actually minimizes the right-hand-side of the bound (14). As a result, our algorithm observes the system state \( X(t) \), \( A(t) \) and \( S(t) \) in every slot \( t \), and chooses the spot price \( P(t) \) as the solution to the following problem, named **Problem One**:

\[
\max N(t)(VP(t) + X(t)), \tag{15a}
\]

\[
s.t. 0 \leq P(t) \leq P_{\max}. \tag{15b}
\]

Recall that \( N(t) \) is a function of \( P(t) \). Thus, the constraint (1) is ensured by the demand curve. It is easy to prove that the solution to problem (15) must be one of the maximum prices of the requests in the current slot. Thus, spot price \( P(t) \) can only take finite possible values in each slot \( t \). The algorithm evaluates the objective function (15a) over all possible prices and chooses the best one. After choosing the price, the number of launched instances \( N(t) \) is determined by the demand curve and the queue \( X(t) \) is updated according to (10). This procedure is repeated every slot. Note that solving **Problem One** only depends on the current system state and does not require any knowledge of the statistics of the user requests. The queueing state \( X(t) \) is crucial for the algorithm which makes decisions purely based on current requests. The algorithm tries to launch more instances when the queue backlog is large (\( X(t) \) is positive), and defer more requests when the queue backlog is small (\( X(t) \) is negative).

B. Performance Bounds

Let \( R_{av}^* \) denote objective value of the time-average maximization problem (8) under an optimal policy (possibly with future knowledge). We have the following theorem:
Theorem 1. Suppose the arrival requests $A(t)$ and the supply $S(t)$ are i.i.d. over slots. If there exist an $\delta > 0$ such that:

$$\mathbb{E}\{A(t)\} \leq \mathbb{E}\{S(t)\} - \delta.$$  

(16)

The algorithm above results in the following performance guarantees:

$$\lim_{t \to \infty} \frac{1}{t} \sum_{\tau=0}^{t-1} \mathbb{E}\{p(t)N(t)\} \leq R^*_{av} + \frac{B}{V},$$

(17)

$$Q \triangleq \lim_{t \to \infty} \frac{1}{t} \sum_{\tau=0}^{t-1} \mathbb{E}\{Q(\tau)\} \leq \frac{B + VP_{\text{max}}S_{\text{max}}}{\epsilon} + A_{\text{max}}.$$  

(18)

Proof: This result follows from the framework in [2] and is omitted for brevity.

Constraint (16) means that the requests should be inside the capacity region (stronger than schedulability condition (5)). This assumption is crucial for queue stability under our algorithm, it is not crucial for the performance of achieved revenue. According to Theorem 1, by choosing large $V$, the revenue can be arbitrarily close to the optimal solution. However, the average queue backlog increases with $V$. This yields a tradeoff between revenue and QoS.

C. Simple Examples

To give the readers some intuition about the algorithm, we show some simple examples here. First, we extend the example in section II. Suppose there are 3 users and their requests follow a periodic fashion. One period consists of 3 time slots. The requests number and corresponding maximum prices in one period are shown in Table II. We assume the system capacity is a constant ($S(t) = 2500$). The maximum number of requests in single slot $A_{\text{max}}$ is set at 2000. In this example, the results from an algorithm that only maximize the revenue of current slot is presented in section II, and the average revenue in each slot is given as 2540/3=847. If the provider tries to accept the requests as many as possible, the average revenue will be 717.

We then simulate our algorithm for different values of $V$ for 1000 periods (3000 slots). The parameter $V$ increases from 0 to 2500. We show the average revenue achieved by the algorithm with different values of $V$ in Table III. While $V$ is set at 0, the average cost 808.1 is lower than the result 847 from single slot revenue maximum algorithm. This is because our algorithm tries to minimize the queue drift (does not consider the revenue) while $V$ equals to 0. As a result, the queue backlog keeps around the shift between $Q(t)$ and $X(t)$, i.e. $A_{\text{max}}$. As $V$ increases, the achieved average revenue is greater than that from single slot algorithm, since our algorithm incorporates revenue into the objective function. It is worth noting that the average revenue does not increase monotonically with $V$. This is because Theorem 1 only provides an lower bound about the performance of the algorithm. The effect of the algorithm can achieve better solution (for small $V$) in reality. In general, the distance between the results of our algorithm and the optimal solution decreases with increasing $V$. As shown in Table III, the average revenue achieves its maximum value 1030 while $V$ equals to 2500.

The above example is somehow complicated. Here we illustrate a simpler example and show our algorithm can make the same decisions with the optimal of single algorithm. In this example, 3 users repeat placing their requests every 3 time slots, too. The price and the number of requests are shown in Table IV. The supply and the maximum number of arrival requests are 900 and 600, respectively. Intuitively, an optimal of single algorithm which knows future request arrivals would set the price at 1, 0.8, and 0.5 in 3 slots, respectively. Thus, the maximum revenue achieved is 410 in average. Further, the single slot algorithm will set the price at 0.8, 0.8, and 0.5, respectively. The average revenue can be easily given by 390. Also, the average revenue achieved by the algorithm that accepts all requests is given by 300.
that, our algorithm learns to achieve the optimal solutions. It's exciting that our algorithm makes optimal decisions without any knowledge of the future information.

Besides the achieved revenue, QoS is another concern. Thus, we compute the average number of requests in pending state for above 2 examples. While our algorithm achieves optimal solutions (V=2500 and V=700 respectively), the average numbers of pending requests are 833 and 400, respectively. They are quite small which can be serviced in 1 slot. Thus, from the cloud provider's point of view, our algorithm maximizes the average revenue while guaranteeing the cloud provider's point of view, our algorithm maximizes the vertices of pending requests are 833 and 400, respectively. They are quite small which can be serviced in 1 slot. Thus, from the cloud provider's point of view, our algorithm maximizes the average revenue while guaranteeing the virtual queue backlog. However, for the cloud users, the delay of each request is more crucial than average queue backlog for the user experience. In both examples, it is easy to see that the delay of each request is less than 3 time slots. Unfortunately, small queue backlog does not always mean short delay. We illustrate this using another example. The cloud user sends 600 requests in every slot. The other 2 users send 100 and 1 request only in the first slot, then no request for the rest slots. It is easy to see that our algorithm will always set the price at 1, and the requests of last 2 users will never be accepted. Although the queue backlog keeps 101 which is very small, the delay of these requests will be infeasible that our algorithm makes optimal decisions without any knowledge of the future information.

Besides the achieved revenue, QoS is another concern. Thus, we compute the average number of requests in pending state for above 2 examples. While our algorithm achieves optimal solutions (V=2500 and V=700 respectively), the average numbers of pending requests are 833 and 400, respectively. They are quite small which can be serviced in 1 slot. Thus, from the cloud provider's point of view, our algorithm maximizes the average revenue while guaranteeing the cloud provider's point of view, our algorithm maximizes the vertices of pending requests are 833 and 400, respectively. They are quite small which can be serviced in 1 slot. Thus, from the cloud provider's point of view, our algorithm maximizes the average revenue while guaranteeing the virtual queue backlog. However, for the cloud users, the delay of each request is more crucial than average queue backlog for the user experience. In both examples, it is easy to see that the delay of each request is less than 3 time slots. Unfortunately, small queue backlog does not always mean short delay. We illustrate this using another example. The cloud user sends 600 requests in every slot. The other 2 users send 100 and 1 request only in the first slot, then no request for the rest slots. It is easy to see that our algorithm will always set the price at 1, and the requests of last 2 users will never be accepted. Although the queue backlog keeps 101 which is very small, the delay of these requests will be infeasible.

To guarantee the worst-case delay, we extend the basic model in the following section.

V. EXTENDED MODEL

The algorithm presented in the previous section ensures virtual queue backlog. However, it does not provide any information about the worst-case delay. In this section, we extend the basic model to achieve worst-case delay guarantees.

A. Queue Model and Revenue Maximization Problem

To provide the worst-case delay, we utilize the virtual queue technology (section V-B). For using this technology, the actual queue must be accepted in a FIFO order to ensure the worst-case delay. However, the queue model in section III does not obey FIFO manner. Obviously, the requests with higher price will be serviced before the ones with lower price. Thus, we need a new model to ensure the queue is serviced in a FIFO order. Here, we separately set up the queues for the requests with different prices. Assume there are K maximum prices corresponding to all requests. \( P_k(t), k = 1, 2, \ldots, K \) denote all maximum prices in slot \( t \) and are sorted in a descending order, i.e. \( P_k(t) > P_{k+1}(t) \). Let \( a_k(t) \) denote the number of requests with maximum price \( P_k(t) \) arrival in slot \( t \). The underlying probability distribution or statistical characterization of \( a_k(t) \) is not necessarily known. We only assume that it is i.i.d. and its maximum value is \( A_{max} \), i.e. we have for all \( t \):

\[
0 \leq a_k(t) \leq A_{max}.
\]  

(19)

Let \( N_k(t) \) be the number of accepted requests with maximum price \( P_k(t) \). Thus, we have:

\[
N_k(t) = \begin{cases}  
Q_k(t) & P_k(t) > P(t) \\
\min(S(t) - \sum_{j=1}^{k-1} Q_j(t), Q_k(t)) & P_k(t) = P(t) \\
0 & P_k(t) < P(t) 
\end{cases}
\]

(20)

Thus, the total number of accepted requests is the sum of all requests with different prices:

\[
N(t) = \sum_{k=1}^{K} N_k(t).
\]  

(21)

Let \( Q_k(t) \) be the number of instances for pending requests with maximum price \( P_k(t) \) in slot \( t \). Then, the dynamics of \( Q_k(t) \) can be expressed as follows:

\[
Q_k(t + 1) = Q_k(t) - N_k(t) + a_k(t).
\]  

(22)

Since the customers can define arbitrary maximum prices for their requests, there may exist a few requests with very low prices. If the system accepts these requests, the spot price of other requests will obviously drop down (example in section IV-C). To achieve more revenue, the cloud operator would drop these requests rather than accept them. We define requests drop decisions \( d_k(t) \) for each queue \( k = 1, 2, \ldots, K \). These allow the requests to be dropped if their price is too low or their delay is too large. Drop decisions are chosen subject to the following constraint:

\[
0 \leq d_k(t) \leq A_{max}.
\]  

(23)

As a result, the dynamics of queue is given as:

\[
Q_k(t + 1) = \max(Q_k(t) - N_k(t) - d_k(t), 0) + a_k(t).
\]  

(24)

However, dropping requests may lose the customers. Thus, we add a penalty for dropping requests to the revenue function:

\[
R(t) = P(t)N(t) - \sum_{k=1}^{K} \beta d_k(t),
\]

(25)

where \( \beta \) is a positive constant.

In conclusion, the time-average revenue maximization problem is changed to:

\[
\max R_{av}^{ext} \triangleq \lim_{t \to \infty} \frac{1}{t} \sum_{\tau=0}^{t-1} \mathbb{E}\{P(t)N(t) - \beta d_k(t)\},
\]

(26a)

s.t. \( \frac{Q_k}{N_k} \triangleq \lim_{t \to \infty} \frac{1}{t} \sum_{\tau=0}^{t-1} \mathbb{E}\{Q_k(\tau)\} < \infty, \)

(26b)

\[
0 \leq P(t) \leq P_{max},
\]

(26c)

\[
0 \leq d_k(t) \leq A_{max}.
\]

(26d)

B. Delay Aware Queue

To ensure the worst-case delay is bounded, we define an \( \epsilon \)-persistent service queue [2], being a virtual queue \( Z_k(t) \) with \( Z_k(0) = 0 \) for each \( k \in \{1, 2, \ldots, K\} \). The dynamics of the virtual queue is given as follow:

\[
Z_k(t + 1) = \begin{cases}  
Z_k(t) - N_k(t) - d_k(t) + \epsilon Q_k(t) & Q_k(t) > N_k(t) + d_k(t) \\
Q_k(t) & Q_k(t) \leq N_k(t) + d_k(t) 
\end{cases}
\]

(27)

where \( \epsilon \) is a positive parameter and upper bounded by \( A_{max} \), i.e. \( 0 < \epsilon \leq A_{max} \). The condition \( Q_k(t) \leq N_k(t) + d_k(t) \) is satisfied as long as all requests with price \( P_k(t) \) are accepted or dropped in slot \( t \). The goal of this virtual queue is to provide worst-case delay guarantee on all queues of the requests. We have the following lemma:
Lemma 2. Suppose \( Q_k(t) \) and \( Z_k(t) \) evolve according to (24) and (27), and that an algorithm is used that ensures \( Q_k(t) \leq Q_{k,\text{max}} \) and \( Z_k(t) \leq Z_{k,\text{max}} \) for all slots \( t \in \{0, 1, 2, \ldots \} \). Assume all requests with same price are accepted in FIFO order. Then the worst-case delay of the requests with lowest price in queue \( k \) is \( W_{k,\text{max}} \), defined:

\[
W_{k,\text{max}} \triangleq \max \left( \left( Q_{k,\text{max}} + Z_{k,\text{max}} \right) / \epsilon \right). \tag{28}
\]

C. Algorithm Design

We now present our online algorithm to solve the time average optimization problem (26). The dynamic algorithm chooses the control decisions as the solution to the following problem, named Problem Two:

\[
\begin{align*}
\max & \quad V[\sum_{k=1}^{K} P(t) N(t) - \sum_{k=1}^{K} \beta d_k(t)] + \sum_{k=1}^{K} Q_k(t)(N_k(t) + d_k(t)) + \sum_{k=1}^{K} Z_k(t)(N_k(t) + d_k(t)) \\
\text{s.t.} & \quad 0 \leq P(t) \leq P_{\text{max}}, \tag{29a} \\
& \quad 0 \leq d_k(t) \leq A_{\text{max}}. \tag{29b}
\end{align*}
\]

Problem Two can be decomposed into 2 subproblems. The first one decides the price which is similar with Problem One:

\[
\begin{align*}
\max & \quad V \sum_{k=1}^{K} P(t) N(t) + \sum_{k=1}^{K} Q_k(t)N_k(t) + \sum_{k=1}^{K} Z_k(t)N_k(t), \tag{30a} \\
\text{s.t.} & \quad 0 \leq P(t) \leq P_{\text{max}}. \tag{30b}
\end{align*}
\]

The second one is request drop problem:

\[
\begin{align*}
\max_{k} & \quad \sum_{k=1}^{K} d_k(t)(Q_k(t) + Z_k(t) - V \beta), \tag{31a} \\
\text{s.t.} & \quad 0 \leq d_k(t) \leq A_{\text{max}}. \tag{31b}
\end{align*}
\]

Problem (31) yields a threshold based decisions:

\[
d_k(t) = \begin{cases} 
A_{\text{max}} & Q_k(t) + Z_k(t) > V \beta \\
0 & Q_k(t) + Z_k(t) \leq V \beta.
\end{cases} \tag{32}
\]

In each slot, we choose the prices and request drops based on (30) and (31), respectively. After computing these quantities, the algorithm updates both actual and virtual queues according to (24) and (27). The algorithm repeats the above procedure every time slot. Again, our online algorithm does not depend on the knowledge of future information.

D. Performance Analysis

For the generalized model, we have the similar Theorem:

Theorem 2. Suppose the arrival requests \( a_k(t), k = 1, \ldots, K \) and supply \( S(t) \) are i.i.d. over slots. Implementing the algorithm above with fixed parameter \( \epsilon \) such that \( 0 < \epsilon \leq A_{\text{max}} \) and parameter \( V \) such that \( V > 0 \) for all slots, results in the following performance guarantees:

1. The queues \( Q_k(t) \) and \( Z_k(t), k = 1, 2, \ldots, K \) are upper bounded and the bounds are defined respectively as follow:

\[
Q_{k,\text{max}} \triangleq V \beta + A_{\text{max}}, \tag{33}
\]

2. The worst-case delay of requests with price \( P_k(t) \) is given by:

\[
W_{k,\text{max}} = 2V \beta + A_{\text{max}} + \epsilon. \tag{34}
\]

3. The time-average revenue achieved by the algorithm can get within \( O(1/V) \) of the optimal solution

\[
\lim_{t \to \infty} \frac{1}{t} \sum_{r=0}^{t-1} E[p(t)N(t)] \leq R^*_{\text{av,gen}} + \frac{B}{V}. \tag{35}
\]

Proof: 1. Requests drop decision (32) yields \( d_k(t) = A_{\text{max}} \) whenever \( Q_k(t) + Z_k(t) > V \beta \). Because \( a_k(t) \leq A_{\text{max}} \), the arrivals are less than or equal to the the drops whenever \( Q_k(t) > V \beta \), and so \( Q_k(t) \leq V \beta + A_{\text{max}} \). Similar, the queue backlog of \( Z_k(t) \) will also decrease if \( Z_k(t) > V \beta \). Therefore, \( Z_k(t) \leq V \beta + \epsilon \).

Proof: 2. This follows from part 1 and Lemma 2.

Proof: 3. This result follows from the framework in [2] and is omitted for brevity.

E. Discussion

We now have a detailed discussion about our 2 models. First, we show the effectiveness of our extended model to guarantee the worst case delay. We use the example in section IV-C. There are 3 users. The first user sends 600 requests every slot. The other two users send 100 and 1 request in the first 20 slots, respectively. Then they 2 do not send any requests in the following slots. Further, we set the parameters: \( \epsilon = A_{\text{max}} = 600, \beta = 2 \) and \( V = 1000 \). We simulate our algorithm for 1000 slots. The sample paths of price and requests drop decision during the first 20 slots are shown in Fig.4. By comparison with the example in section IV-C, new algorithm sets the price at 0.8 in the second slot and drops the request from user 3 in the third slot. As a result, the worst-case delay of all requests is no more than 5 slots. This is due to the increase of virtual queue of these 2 users. The virtual queue for user 2 is 600 in slot 2. Although the actual queue is relative small (100), the virtual queue can lower the price to accept his requests. However, the actual queue is too small for user 3. Thus the virtual queue for this user continues to increase. Finally, it increases to 2400 which is larger than \( V \beta = 2000 \) in slot 5. The algorithm drops this request and incur a penalty of 2.
is desired and Qos is also considered. However, for some small group of users, their requests cannot not be serviced due to either low price or small number of requests. The extended model tries to guarantee the worst-case delay of all requests. It either accepts the requests with relative high price and large quantity of instances or drops the requests with too low price and small quantity of instances. The penalty is revenue loss. The extended model does not only incur a penalty of request drops, but also loses revenue due to lowering the spot price to accept more requests. In above example, the algorithm chooses the price at 0.8 in slot 2. This decision incurs a revenue loss of 600-560=40.

VI. PERFORMANCE EVALUATION

In this section, we evaluate our algorithms based on real-life data center workload and Amazon EC2 historical Spot Price.

A. Experimental Setup

**Spot instance and its maximum price** Recently, Google published a new sample dataset about workloads running on Google compute cells [3]. This trace includes data from a cluster of 11k servers over 29 days. We map the jobs which run in the system to the spot requests. All jobs in the trace have a property that roughly represents how latency-sensitive it is. We use this property to distinguish the requests. We assume that the requests with higher latency-sensitive value have higher maximum price. Thus, we have four groups of requests (with the property value of 0, 1, 2, and 3). The number of requests is shown in Fig.5.

We use Amazon EC2 historical Spot Price as the maximum price of the requests. We assume that the requests in the same group have the same maximum price. We consider 2 types of instances for different time duration. The first type is m1.large, its duration lasts for 1 day. The other type is m1.small, and its duration lasts for 1 month. The price history is shown in Fig.6. As shown in the figure, the price of Instance 2 has wide range. It varies from 0.007 to 2$. In comparison, the price Instance 1 only varies from 0.026 to 0.0305$. We will show that the performance of our algorithms varies a lot due to the different characteristics of the price.

**System Parameters** The length of time slot is 1 hour. The supply is set by 3200 which is approximated to the maximum number (3164) of requests in a single time slot. Also, the upper bound of arrival requests is set at 1600. The penalty for dropping a request is assumed as 2$.

**Schemas for Comparison** To evaluate the performance of our algorithms, we simulate 4 schemas. Match All tries to serve requests as many as possible. In our setting, the supply is always sufficient to the requests in the current slot. Thus, Match All will accept all requests and set the price at the lowest level. Single Slot greedily maximizes the revenue of current slot. It considers all arrival and pending requests, then sets a price to maximize current revenue. All requests with lower prices will be in pending state and wait for service in the future. Basic Model and Extended Model are the algorithms presented in section IV-A and V-C, respectively.

B. Effectiveness of Our Algorithms

First, we simulate 4 algorithms for instance 1. Match All and Single Slot does not depend on the value of V. The average revenues of these 2 schemas are 24.9573 and 25.7863, respectively. To compare the revenue of 4 algorithms, we use Match All as a baseline. For all the other schemes, we compute the percentage of average revenue increase as compared to Match All. The performance of our 2 algorithms depends on the value of parameter V. We show the results under different V in Fig.7. As we expect, all 3 algorithms obtain more revenue than Match All, since Match All always set the price at lowest level. Basic Model always generates the most revenue among all schemas. This is consistent with the results of Theorem 1. Basic Model can be arbitrarily close to the optimal solution. Because it does not only maximize the revenue of current slot, but also considers the future. Single Slot comes next, since it greedily maximizes current revenue. Extended model obtains less revenue than single slot, since extended model will lower the price for reducing the worst-case delay. Further, it also incurs penalty for dropping requests. From the figure, the performance of our 2 algorithms varies a little with V increase. This is due to the small range of the maximum prices for different requests.

We also compute the average queue backlog for all algorithms and show the results in Table V. Compared with the revenue, the backlogs of 4 algorithms stay in an inverse order. This yields the tradeoff between revenue and the quality of service. As shown in the table, Basic Model has much more backlog than all the other schemas. This is the cost of high revenue. Furthermore, Match All always have no backlog in the queue. Because the supply is sufficient to the requests in the current slot, Match All will serve all arrival requests. Similar with the revenue, the queue backlogs of our 2 algorithms change little with V increase. We also check request drop decisions of Extended Model. For all above simulation, it rarely drops requests. The algorithm only drops 2 requests while V equals to 700 and 800, respectively. This is easy to understand. Since the prices are similar for different requests, the backlog (both actual and virtual queue) plays a dominant role as determining the price. As a result, the actual queue backlog does not accumulate too much and very few requests are dropped. This infers that the penalty of dropping requests is not the only reason of lower revenue for Extended Model.
The effectiveness of our algorithms is relatively modest for instance 1 due to the similar prices of the requests. Here, we show the experimental results for instance 2. The results are quite different from instance 1. The average revenue is shown in Fig. 8. Match All is not shown in the figure due to its awful revenue. It only obtains 6.7193 in average. This is due to very low price for the last group of requests. The revenue of Extended Model does not stabilize while V equals to 1000, so we continue increasing V to 10000. As shown in Fig. 8, the revenues of Basic Model and Extended Model increase with V and are clearly stable. The difference is: the revenue of Basic Model increases while V is small and stabilizes quickly, the revenue of Extended Model begins to increase while V is relatively large and stabilizes slowly. It is unexpected that Single Slot obtains more revenue than Basic Model. This seems to counter Theorem 1 that Basic Model get close to the optimal solution. We analyze the results deeply and find that Single Slot has very large queue backlog. The average backlog of different algorithms is shown in Table VI. We can see from the table that the backlog of Single Slot is much larger than the other schemas. Indeed, the backlog of Single Slot keeps increasing and looks like to be infinite. This is due to very large range of prices for different requests. The requests with lowest price can not be serviced and the queue will be cumulated. This violates the constraint of finite queue backlog (6). Similar with instance 1, Basic Model has a longer queue than Extended Model.

We also compute the number of dropped requests for extended model and show the results in Table VII. From the table, the number is impacted by the parameter V dramatically. The number first increases with V while V is small. Then it decreases as V keeps increasing. The drop decision (32) is to compare the queue backlog with $V^\beta$. When V is small, increasing V makes the queue cumulate fast, thereby more requests are dropped. As V becomes larger, queue cumulation becomes slow. In contrast, the target value $V^\beta$ keeps increasing. Thus, request drop is delayed to later time. As a results, although the queue also increases, less requests are dropped. However, this increases the worst-case delay (shown in Table VI and equation (28)).

### VII. Related Work

Pricing is an active research area in cloud computing. Many studies on pricing for different resources have be proposed. Kantere et al. [4] proposes a novel pricing scheme designed for a cloud cache that offers querying services and aims at the maximization of the cloud proR. Upadhyaya et al. [5] consider the situation that multiple users access common datasets and cloud provider implements optimizations to accelerate queries over these datasets. The authors study how a cloud data service provider should price optimizations to share their cost among users. Niu et al. [6], [7] study the pricing problem for
bandwidth reservations in cloud computing.

Spot Instances provide many benefits including fully utilization of compute resources and reduction of users cost. A study of how spot price is set is presented in [8]. As one of major advantages of spot instances, many researchers pay attention to how can users utilize spot instances to reduce their monetary cost[9], [10], [11], [12], [13]. Besides the user cost, many works also focus on the pricing schema for maximizing the profit of cloud providers[14], [15], [16], [17]. However, all above works do not consider the impact of pricing on future revenue. The most related work to this paper is [18]. The authors investigate dynamic pricing problem for spot instances to maximize the expected revenue of provider for long time. They propose a arrival-departure model to characterize the impact of current price on the future demand. Then, a dynamic programming based approach is proposed to solve the dynamic pricing problem. However, this approach requires the statistics of request arrival which is usually hard to obtain in practice. The algorithms presented in this paper can get within \(O(1/V)\) of the optimal solution without requiring any knowledge of the system statistics. Also, the demand curve model in this paper is more suitable for spot instances in current cloud computing environment. Furthermore, our algorithms can provide QoS guarantee for the user requests.

VIII. CONCLUSION

In this paper, we investigate the problem of pricing the spot instances to maximize the revenue of cloud providers. We argue that pricing to the instances dose not only determine the revenue for the present, but also impacts on the future. To characterize the impact of pricing on the present and future revenue, we present a demand curve model. Then we formulate the revenue maximization problem as a time-average optimization problem. While this model fails to provide the worst-case delay guarantee for user requests, we extend this model to ensure this delay. Using Lyapunov optimization, we design online algorithms for both models. The evaluation results show that both of our algorithms can obtain high revenue while providing finite queue backlog.

ACKNOWLEDGEMENT

This work was supported in part by National Natural Science Foundation of China under Grant No.61272460, 60933003 and Science Foundation of Ministry of Education of China under Grant No.20120201110010.

REFERENCES