Cutting Your Cloud Computing Cost for Deadline-Constrained Batch Jobs

Min Yao*, Peng Zhang†, Yin Li*, Jie Hu*, Chuang Lin* Xiang-Yang Li‡,
*Tsinghua National Laboratory for Information Science and Technology (TNList)
Department of Computer Science and Technology, Tsinghua University, Beijing
†Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an
‡Department of Computer Science, Illinois Institute of Technology, Chicago

***Abstract*—Many web service providers use commercial cloud computing infrastructures like Amazon for flexible and reliable service deployment. For these web service providers, the cost of cloud computing usage becomes a big part of their IT department cost. Facing the diverse pricing models including on-demand, reserved, and spot instance, it is difficult for web service providers to optimize their cost.**

This paper introduces a new *cloud brokerage service* to help web service providers to minimize their cloud computing cost for deadline-constrained batch jobs, which have been a significant workload in web services. Our cloud brokerage service associates each batch job with deadline, and always tries to use cheaper reserved instances for computation to maintain a minimum cost. We achieve this with the following two steps: (1) given a set of jobs' specifications, determine the scheduling of jobs; (2) given the scheduling and pricing options, find an optimal instance renting strategy. We prove that both problems in two steps are computation intractable, and propose approximation algorithms for them. Trace-based evaluation shows that our cloud brokerage service can reduce up to 57% of the cloud computing cost.

I. Introduction

Cloud computing, e.g., Infrastructure-as-a-Service (IaaS), enables customers to obtain the computing resources (in the form of virtual machines) in an on-demand way. As a result, cloud computing customers can save the cost incurred by one-time infrastructure investment as well as daily maintenance.

To meet the requirements of a wide variety of customers, major IaaS providers (e.g., Amazon Web Services, Google Compute Cloud, Microsoft Windows Azure) are offering diverse pricing options. Common pricing options include (i) on-demand instance, (ii) reserved instance, and (iii) spot instance. When using on-demand instance, customers pay their hourly usage by without long-term commitment. Reserved instance, on the other hand, let users make a one-time payment for instance reservation, and enjoy a lower hourly charge when using these instances. For spot instance, users can bid for the unused capacity in the cloud, and whenever the bidding price is no lower than the spot price, the users can get the spot instances for the next hour. While the instance would be terminated abruptly when the user's bid falls below the spot price.

Recently, many cloud brokerage service [1] emerged to help IaaS customers to reduce their cost by intelligently using different pricing options. It is anticipated that global cloud brokerage market will grow from $1.57 billion in 2013 to $10.5 billion by 2018 [1]. However, the intelligence inside these cloud brokerage service is little known to the academy. In addition, we are unaware of any public cloud brokerage service for batch jobs, which is currently a significant part of computing workload in web services. Examples of batch jobs include MapReduce/DryadLINQ jobs, web search index update, monte carlo simulations, software testing, etc. Most of the batch applications have large computing workload but no real-time requirement, indicating that the computation can be spread over a finite time window specified by the job arrival time and due date. We term these jobs as deadline-constrained batch jobs, define its *slackness* as the ratio of the time window and earliest finish time, and defines its *spare_time* as the difference between the due date and earliest finish time. For example, if a job requests a instance for 4 hours and its deadline constraint is 6 hours, the *slackness* and *spare_time* of the job is 3/2 and 2 respectively.

Different from the traditional resource management system within cloud providers [12][13], our proposed brokerage service does not need to maintain a large pool of instances in advance. Instead, our broker accepts deadline-constrained batch jobs from web service providers during a certain period, and then give an optimal renting strategy, which minimizes the total cost. Though both of the reserved instance and spot instance can be used to reduce cost, however, here we only choose to use reserved instances (which also has a lot of different options). The reasons are as follows. (i) spot price is determined by the provider according to undisclosed rules, and thus very hard to predict. (ii) The heuristic method, which launches spot instances first and then uses on-demand instance when spot instances fail will introduce extra delay to jobs. (iii) The unexpected termination of the running spot instances is terrible for users if they do not backup job's status frequently.

Even without usage of spot instances, we can still significantly reduce the cloud cost by properly choosing the reserved instances. For example, a 3-year term reserved instance plan can save up to 65% of the total cost [8]. Our basic idea is to minimize the peak demand of instances by spreading the computation load in a time window. The smaller the peak demand, the more opportunities for exploiting long-term reserved instance. Take Fig. 1 as an example, there are four jobs with deadline constraints. The straightforward approach would need 2 instances, while our approach can reduce the

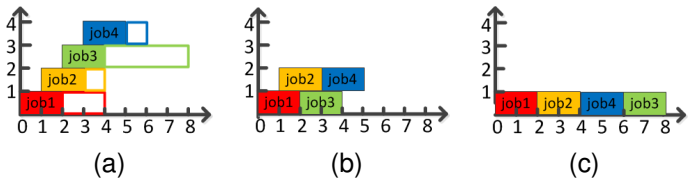

Fig. 1. A motivating example of cloud brokerage service: (a) four jobs with different deadlines arrive; (b) the straightforward approach: two instances needed; (c) taking job deadline into account: only one instance needed.

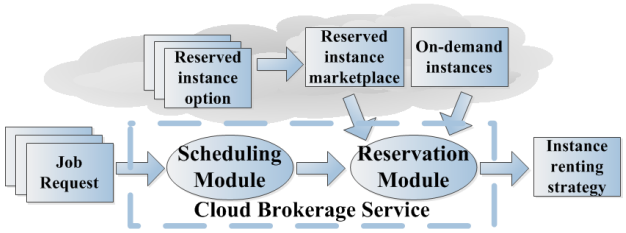

Fig. 2. Framework of our cloud brokerage service.

number to 1, thereby uses only one long-term instance.

In sum, our main contribution is three-fold:

1) We introduce a new cloud brokerage service, consisting of a scheduling module and a reservation module, to help reduce the cloud cost incurred by deadline-constrained batch jobs.
2) We formulate these two modules as two separate optimization problems, and show they are both very hard to solve. To make these problems computation tractable, we propose heuristic algorithms to approximately solve them.
3) We use real traces collected from one of Google's large-scale computing clusters, and conduct extensive simulations to demonstrate that our cloud broker service can significantly reduce the cost incurred by deadline-constrained batch jobs.

The rest of this paper is structured as follows. A brief discussion of the related work is presented in Section II. System overview and problem formulations are given in Section III. Section IV presents our scheduling algorithm and reservation algorithm to approximately solve the problem. Section V reports the evaluation of the algorithms with the real dataset from Google cluster-usage trace. Section VI concludes this paper.

II. Related work

Cloud brokerage has become a hot topic in both industrial and academic field. A lot of companies running cloud brokerage services emerged these years [1]. All these commercial cloud brokerage service platforms do not provide the specific service for deadline constrained batch jobs. Recently, there are also several research effort on cloud brokerage. For example, Wieder *et al.* [3] proposed *Conductor* that help user choose cloud services to deploy MapReduce computations in the cloud. Zhao *et al.* [4] developed a resource rental planning for elastic applications with the computational and storage resources in the cloud market to reduce the operation costs. Zafer *et al.* [5] designed a dynamic bidding policy for spot instances in order to minimize the average cost of a job with deadline constraint. Although the cloud brokerage services of [3]-[5] guarantee the deadline constraint for the job requests, they accommodate individual user job requests separately. Song *et al.* [6] designed a bidding strategy from a cloud brokerage's perspective, where the brokerage receives job requests from cloud users and leverages the opportunistic spot instances to maximize the its revenue. Wang *et al.* [7] proposed a cloud brokerage that receives job requests and exploits benefits of long-term instances reservation to minimize its service costs. Jain *et al.* [2] studied an online learning algorithm for resource allocation incorporating history of spot instances and workload characteristics. However, the cloud brokerages in [6][7][2] are not suitable for deadline constrained batch jobs. Our cloud services brokerage exploits the *slackness* of jobs to aggregate them, so as to enjoy the benefits of cheaper reservation options and resource multiplexing gains.

Our brokerage service is different from the resource management system within cloud providers. Both of the resource management systems in [12][13] manage a large pool of computation resources and discuss how to efficiently utilize the computation resources to fulfill the user requests and reduce the cost of the cloud providers. However, our brokerage service doesn't need to reserve a pool of instance in advance.

The work of [14][15] are similar to the scheduling module of our brokerage service. The strip packing with slicing problem [14] is different since it didn't consider the deadline constraints of each job. [15] proposed a method to check whether there exist a preemptive schedule on m machines that complete n jobs within its release time and deadline, but this method can only applied to single-task job.

III. Problem Statement and Formulation

A. Problem Statement

Our problem setup focuses on the web service providers who submit their batch jobs demand estimates over a certain period of time. Specially, the time horizon is divided into T time slots, which represents an time interval of one hour in IaaS cloud. In our model, each batch job is characterized by a tuple with four parameters $R_i = (D_i, c_i, s_i, d_i)$, where D_i is the total computation resource demand in instance-hour unit, c_i is the concurrency requirement which denotes the degree of parallelism, i.e., the number of instances that should be simultaneously assigned to a job, s_i and d_i denote the start time and due date of a job respectively. Our brokerage should make sure that a job is completed in finite time window $[s_i, d_i]$. Thus, the job i is said to be active at time t if $s_i \leq t \leq d_i$ and the instances assigned to it so far are less than D_i.

B. System Framework

As shown in Fig. 2, our cloud brokerage service consists of two components: the scheduling module and the reservation module. The scheduling module exploits the *slackness* of

Parameter	Meaning
M	Total number of deadline-constrained batch jobs
V_t	Total number of instances needed at time t
x_{it}	Number of instances allocated to job i at time t
I_t	Set of jobs that can be allocated instances at time t
τ_i	Reservation period for option-i reserved instance
r_t^i	Number of option-i reserved instance rented at time t
u_t^i	Number of option-i reserved instances used at time t
o_t	Number of on-demand instances used at time t
γ_i	One-time charge for option-i reserved instance
α_i	Hourly charge for option-i reserved instance
β	Hourly charge for on-demand instance

TABLE I
Notations used in this paper.

job deadlines for possible aggregation, in order to minimize the peak instance demand over a given period. Recall the example in Fig. 1, the scheduling model aims to achieve (c) where the peak instance demand is 1. Given the aggregated instance demand calculated by the scheduling module, the reservation module aims to find an optimal renting strategy to minimize the instance rental cost. In the following, we present the formulations for the scheduling module and reservation module, respectively.

C. The Scheduling Module

The main function of scheduling module is to increase the opportunity of exploiting long-term reserved instance, which has a lower hourly charge, through two ways: (i) Take full advantage of *slackness* specified by batch jobs to aggregate them and smooth out individual bursts; (ii) Multiplex an instance in time horizon as much as possible. The goal of scheduling module is to minimize the maximum instances needed per hour. We formulate the aggregation mechanism of scheduling module as the following optimization problem. Notations used in the following are given in Table I.

$$\begin{aligned} \text{minimize} \quad & \max_{0\le t<T} V_t = \sum_{i\in I_t} x_{it} \qquad (1)\\ \text{subject to} \quad & \sum_{j=s_i}^{d_i-1} x_{ij} \ge D_i,\ \forall i \in [1,M] \\ & x_{it} \in \{0, c_i\},\ \forall i \in [1,M], \forall t \in [s_i, d_i) \\ & x_{it} = 0,\ \forall i \in [1,M], \forall t \notin [s_i, d_i) \end{aligned}$$

The first set of constraints mean the total number of instances should be no less than the total requests of all jobs; the second set of constraints mean the concurrency requirements of each job should be met; and the third set of constraints mean instances are only assigned to a job which is during its time window (the time between its arrival and deadline). Optimization problem (1) is a special case of 0-1 integer linear program with many variables, whose number scales with the number of batch jobs and the time window length of each job.

Theorem 1. *The optimization problem (1) is NP-Complete.*

The proof is given in Appendix A.

D. The Reservation Module

Given the aggregated demand V_t obtained by the scheduling module, the reservation module is responsible for determining the combinational use of on-demand instance and various options of reserved instance. Most cloud service providers offer a series of reservation options. For example, Amazon provides three options: light(low-usage), medium(middle-usage), and heavy(high-usage) instances [8]. A higher-usage option has a higher one-time upfront charge, while a lower hourly charge, and vice versa. For each option, Amazon offers a one-year term and a three-year term. Suppose there are w kinds of reservation options $\{(\gamma_i, \alpha_i, \tau_i), i \in [1,w]\}$, where γ_i denotes the one-time upfront charge, α_i denotes the hourly usage charge, and τ_i denotes the length of reservation period. Then we can model this reservation problem as the following optimization problem. Notations used in the following are given in Table I.

$$\begin{aligned} \text{minimize} \quad & \sum_{t=0}^{T-1}(\sum_{i=1}^{w} r_t^i\gamma_i + \sum_{i=1}^{w} u_t^i\alpha_i + o_t\beta) \qquad (2)\\ \text{subject to} \quad & u_t^i \le \sum_{j=t-\tau_i+1}^{t} r_j^i,\ \forall t \in [0,T-1], \forall i \in [1,w] \\ & V_t \le \sum_{i=1}^{w} u_t^i + o_t,\ \forall t \in [0,T-1] \\ & r_t^i, u_t^i, o_t \ge 0,\ \forall t \in [0,T-1], \forall i \in [1,w] \end{aligned}$$

The objective function of optimization problem (2) is to minimize the total rental cost, which consists of three parts: one-time upfront charges, hourly charges incurred by reserved instance usage, hourly usage charges incurred by on-demand instance usage. The first set of constraints mean the usage number of reserved instance should not be greater than the number of reserved instance that are still effective. The second set of constraints make sure that the usage number of reserved and on-demand instances satisfies the aggregated demand curve. The last set of constraints put limits on variables.

The optimization problem (2) can be seen as an integer linear program (ILP) with many variables, and is thus very difficult to solve. An straightforward method is dynamic programming, while the complexity is exponential even in the case of single reservation option. For a detailed analysis on the computation complexity of problem (2), please refer to Appendix B.

IV. Algorithms

As shown in the Section III, the two optimization problems corresponding to the scheduling module and reservation module are both very difficult to solve. Thus, in this section we propose a heuristic scheduling algorithm for optimization problem (1) and a greedy reservation algorithm for optimization problem (2).

A. Scheduling Algorithm

The basic idea of scheduling algorithm is to search for the maximum instances needed per hour, V_{max}, as small as

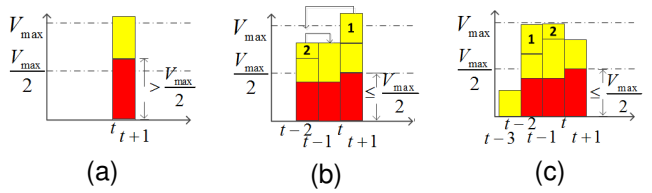

Fig. 3. Scenarios for the proof. The red part denotes the allocation can't be moved. The yellow part denotes the allocation can be moved.

possible using binary search. The initial lower and upper bound are set to be 1 and $\sum_i c_i$, respectively. For any given $V_{max} \in [1, \sum_i c_i]$, we use a function *verifyK* to check whether the given V_{max} is suitable for all job requests. *verifyK* does the following work: it first select the set of jobs that are still active at time t, denoted as JA_t; then it calls the function *chooseJobsToServe* to pick the set of jobs with smallest *spare_time*, denoted as JC_t, and assigns the V_{max} instances to jobs in JC_t. If there exists a job whose *spare_time* is equal to 0, but not in JC_t, then we increase the lower bound of V_{max}. If the V_{max} can meet the demand of all jobs, then we decrease the upper bound.

In the procedure of *chooseJobsToServe*, we can choose the job with the smallest *spare_time* one by one. However, this straightforward method is inefficient. For example, suppose there are four jobs with the same $spare\_time = 1$, and their concurrency requirements are $c_1 = 1, c_2 = 2, c_3 = 3, c_4 = 4$. Now considering $V_{max} = 5$, the above method may pick up the first and second jobs at time t, and the last two jobs at time $t + 1$. However, $V_{max} = 5$ is too small for time $t + 1$. If we pick up the first and last jobs at time t and the second and third jobs at time t+1, $V_{max} = 5$ can be enough for both. Thus, in the procedure of *chooseJobsToServe*, we choose the jobs from first smallest *spare_time* level, then the second level, etc. In each level, we call *exactPickup*, which is similar with knapsack problem, to pick up the proper set of jobs and put them into the set JC_t. Algorithm 1 describes the main framework of aggregation algorithm, whose computation complexity is $O(MT(\sum_{i=1}^{M} c_i)\log(\sum_{i=1}^{M} c_i))$.

Algorithm 1: Scheduling Algorithm

Input: $\boldsymbol{R}$: job requests $\{R_i\}(i \in [1, M])$
Output: V_{max}: maximum instances needed per hour
$\boldsymbol{V}$: number of instances needed per hour $\{V_j\}(j \in [0, T))$

$V_{low} \leftarrow 1$; $V_{high} \leftarrow \sum_{i=1}^{M} c_i$; $V_{max} = (V_{low} + V_{high})/2$;
while $V_{low} \leq V_{max} \leq V_{high}$ **do** `/* Binary search */`
  Modify V_{low}(or V_{high}) and V_{max} based on the return value of *verify*(V_{max}, $\boldsymbol{R}$, $\boldsymbol{V}$);
Function *verifyK*(k, $\boldsymbol{R}$, $\boldsymbol{V}$): *bool*
  New a $1 \times T$ vector $\boldsymbol{V}'$ with value 0;
  for $t \leftarrow 0$ **to** *T-1* **do**
    Clear the sets JA_t and JC_t;
    Delete jobs that have been finished, i.e., $D_i = 0$;
    $JA_t \leftarrow$ select jobs that are still active at time t;
    if *chooseJobsToServe*(JA_t, JC_t, k, t)=*false* **then**
      `// exist a job` $\notin JC_t$ *spare_time*=0
      return *false*;
    else
      $D_j \leftarrow D_j - c_j$, ($\forall$ job $j \in JC_t$);
      $V_t \leftarrow \sum_{j \in JC_t} c_j$;
  $\boldsymbol{V} \leftarrow$ copy the set $\boldsymbol{V}'$;
  return *true*;
Function *chooseJobsToServe*(JA_t, JC_t, k, t): *bool*
  while $k > 0$ & $JA_t \neq \emptyset$ **do**
    $candidate \leftarrow$ Choose jobs in JA_t with smallest $spare\_time$;
    $exactPickup(candidate, pickup, k)$;
    if $spare\_time = 0$ & $candidate - pickup \neq \emptyset$ **then**
      return *false*;
    else
      $JC_t \leftarrow JC_t \cup pickup$;
      $k \leftarrow k - \sum_{i \in pickup} c_i$;
      $JA_t \leftarrow JA_t - candidate$;
  return *true*;

Proposition 1. *Algorithm 1 is 2-competitive, meaning that for any job requests, the maximum number of instances needed per hour V_{max} obtained by Algorithm 1 is no more than twice the optimal solution V_{opt}.*

Proof: We will use the rule of converse-negative proposition for the proof of Proposition 1. The converse-negative proposition is stated as: *if there is not a schedule in Algorithm 1 with a given V_{max}, then the optimal solution V_{opt} must be greater than $V_{max}/2$.* For a given V_{max}, suppose Algorithm 1 stops at time $t \in [0, T-1]$. As shown in Fig. 3, the instances allocation of jobs has two types: yellow part denotes the allocation can be moved, and red part denotes the allocation can't be moved. If the red part at time *t* is greater than $V_{max}/2$, as shown in Fig. 3(a), the V_{opt} must be greater than $V_{max}/2$. If the red part for any time $t' \in [0, t]$ is not more than $V_{max}/2$, as shown in Fig. 3(b), we need to move some of yellow parts at time *t*, such as the no. 1 yellow part. Notes that the yellow parts at time *t* can only be moved forward and there doesn't exist a time $t' \in [0, t-1]$ that can directly accept the no. 1 yellow part, we need to find some yellow parts at time $t' \in [0, t-1]$, whose due dates are less than *t*, move them to a later time before their due dates and make a room for no. 1 yellow part. For example in Fig. 3(b), we can move no. 2 yellow part to time *t-1* and move no. 1 yellow part to time *t-2*, just like Fig. 3(c). After the adjustment, if the instance allocation is greater than $V_{max}/2$ for all $t' \in [0, t]$, the V_{opt} must be greater than $V_{max}/2$. Otherwise, there exists some moments at which instance allocation is less than $V_{max}/2$. In Fig. 3(c), we suppose $V_{t-3} < V_{max}/2$. However, it is impossible to move some yellow parts at time $t' \in [t-2, t]$ to time *t-3* since algorithm 1 determines that those jobs either have been allocated instance at time *t-3* or doesn't start at time *t-3*. Therefore, the V_{opt} must be greater than $V_{max}/2$. Summarizing the analysis above, we can conclude that the converse-negative of Proposition 1 is true. Thus, Proposition 1 is proven. ■

B. Reservation Algorithm

After the aggregated demand curve V_t is generated from scheduling module, we divide it into levels, as shown in Fig. 4, which contains five levels. The main idea of our greedy renting algorithm is to calculate the optimal renting strategy from top level to bottom level with dynamic programming. We use $d_t^l = \{0, 1\}$ to denote whether there is a demand at level l at time t, $c_t^l = \sum_{i=0}^{t} d_i^l$ to denote the accumulative demand at level l from time 0 to time t, and L_t^i to denote the number of option-i reserved instances which are rented at upper layers and left over to level l if they are not used at time t. Then we define C_t^l to be the minimum cost of serving demand from time 0 to time t at level l. For every level, the recursive Bellman equations are

$$C_t^l = \min\{\min_{i\in[1,w]}\{C_{t-\tau_i}^l + \gamma_i + (c_t^l - c_{t-\tau_i}^l)\alpha_i\}, C_{t-1}^l + m_t^l\} \tag{3}$$

$$m_t^l = \begin{cases} 0 & \text{if } d_t^l = 0 \\ \min_{i:L_t^i>0}\alpha_i & \text{if } d_t^l \neq 0 \text{ and } \exists i \in [1,w] : L_t^i > 0 \\ \beta & \text{if } d_t^l \neq 0 \text{ and } \forall i \in [1,w] : L_t^i = 0 \end{cases} \tag{4}$$

which choose the minimum cost among different choices. The first set of choices are to optimally serve the demand up to time $t - \tau_i$ and rent a option-i reserved instance. The second set of choices are to use either reserved instance left over from upper levels or on-demand instance. We calculate the minimum cost at level l from time 0 to $T-1$. After the C_{T-1}^l is attained, we need to find a time path, such as $(t_0 = 0, t_1, t_2, \cdots, t_n = T-1)$ that follows equation (3) to get C_{T-1}^l.

If the distance between two adjacent time steps t_{j+1} and t_j is equal to the reservation period of some reserved instance, e.g. τ_i, then we need to rent a option-i reserved instance at time $t_j + 1$ and update the left over matrix during the period $(t_j, t_j + \tau_i]$ as

$$\begin{aligned} &r_{t_j+1}^i = r_{t_j+1}^i + 1; \\ &L_k^i = L_k^i + 1 - d_k^l, \qquad \forall k \in (t_j, t_j + \tau_i]; \end{aligned} \tag{5}$$

where $\mathbf{r}=\{r_t^i\}_{w\times T}$ is used to denote the number of option-i reserved instances that should be rented at time t.

For the case of $t_{j+1} - t_j = 1$, if $d_{t_{j+1}} = 1$, then we use either the reserved instance left over from upper levels or the on-demand instance as the rules shown in (6); otherwise, nothing is done since there is no demand at time t_{j+1} at level l. We define $p_t = \{i | i \in [1,w] \& \alpha_i = \min_{k\in[1,w]\&L_t^k>0}\alpha_k\}$, which means the type of left over reserved instance with cheapest hourly usage fees.

$$\begin{cases} L_{t_{j+1}}^{p_{t_{j+1}}} = L_{t_{j+1}}^{p_{t_{j+1}}} - 1 & \text{if } \exists i \in [1,w] : L_{t_{j+1}}^i > 0 \\ o_{t_{j+1}} = o_{t_{j+1}} + 1 & \text{if } \forall i \in [1,w] : L_{t_{j+1}}^i = 0 \end{cases} \tag{6}$$

where $\mathbf{o}=\{o_t\}_T$ is used to denote the number of on-demand instance that should be rented at time t.

The details of the reservation algorithm are presented as Algorithm 2. The computation complexity of the algorithm 2 is $O(l_{max}wT)$, and the space complexity is $O(wT)$, which are better than the exponential level required in the original dynamic programming method.

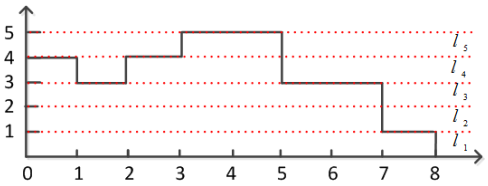

Fig. 4. Leveling of aggregated demand curve.

Algorithm 2: Reservation Algorithm

Input: $\boldsymbol{V}$: The aggregated demand curve $\{V_t\}$;
$\beta, \{(\gamma_i, \alpha_i, \tau_i)|i \in [1,w]\}$: Parameters of pricing options
Output: Optimal renting plan $\mathbf{r} = \{r_t^i\}_{w\times T}$ and $\mathbf{o} = \{o_t\}$

```
l_max ← max{V_i};
New a matrix L=={L_t^i}_{w×T} to record the number of reserved instances left over from upper levels;
for l ← l_max to 1 do
    Generate demand vector and cumulative demand vector at level l: d^l={d_t^l}, c^l={c_t^l};
    for t ← 0 to T-1 do
        Use eq.(3)(4) to calculate the optimal C_t^l ;
    t ← T − 1;
    while t > 0 do
        if ∃i ∈ [1,w] : C_t^l = C_{t−τ_i}^l + γ_i + (c_t^l − c_{t−τ_i}^l)α_i then
            Use eq.(5) to update r and L with t_j = t − τ_i;
            t ← t − τ_i;
        else
            if d_t^l = 1 then
                Use eq.(6) to update o and L with t_{j+1} = t;
            t ← t − 1;
```

Proposition 2. *The difference of total cost between the optimal solution and the solution generated from Algorithm 2 is not more than* $((\tau_{max}-1)(\beta-\alpha_{max})-\gamma_{max})l_{max}$, *where* $\alpha_{max} = \max_{1\le i\le w}\alpha_i, \tau_{max} = \max_{1\le i\le w}\tau_i$.

Proof: Given a level l, suppose the optimal solution for this level is composed of *k+1* time fragments $\{(t_0, t_1, \phi_0), (t_2, t_3, \phi_1), \cdots, (t_{2k}, t_{2k+1}, \phi_k)\}$, $(\phi_i \in [0, w])$. The symbol ϕ_i attached to a fragment represents the demands during the fragment are all serviced with the corresponding instance, e.g., $\phi_i = 0$ represents on demand instance, $\phi_i > 0$ represents option-ϕ_i reserved instance. Except the last fragment, the length of fragment with symbol $\phi_i > 0$ must be equal to τ_{ϕ_i}. And based on Algorithm 2, we can get the optimal value for C_t^l when $t = t_{2i+1}$, $(0 \le i < k)$. If $t_{2k+1} - t_{2k} = \tau_{\phi_k}$, then we can get the optimal value for C_{T-1}^l. Otherwise, if $t_{2k+1} - t_{2k} < \tau_{\phi_k}$, the difference between C_{T-1}^l derived from Algorithm 2 and the optimal value is not more than $(\tau_{\phi_k} - 1)(\beta - \alpha_{\phi_k}) - \gamma_{\phi_k}$. Since there are l_{max} levels, the difference between Algorithm 2 and optimum is not more than $((\tau_{max} - 1)(\beta - \alpha_{max}) - \gamma_{max})l_{max}$. ■

Pricing Model		Option ID	Upfront	Hourly	Break-even Point
On-demand		0	$0	$0.060	\
Light	1month	1	$5.014	$0.034	26.78%
	2month	2	$6.452	$0.0305	15.19%
	3month	3	$7.890	$0.027	11.07%
Medium	1month	4	$11.425	$0.021	40.69%
	2month	5	$14.548	$0.019	24.64%
	3month	6	$17.671	$0.017	19.03%
Heavy	1month	7	$13.890	$0.014	41.94%
	2month	8	$17.509	$0.013	25.87%
	3month	9	$21.123	$0.012	20.37%

TABLE II
Pricing model of on-demand and Light/Medium/Heavy 1/2/3 month terms reserved m1.small instance.

V. Trace-Based Evaluation

In this section, we first introduce the evaluation setup, including the Amazon EC2 pricing options and Google cluster-usage trace. Then, we conduct a series of evaluations, and analyze the results to demonstrate the of effectiveness of our cloud brokerage service.

A. Setup

Amazon EC2 Pricing Options. Different IaaS provider offers different kinds of instance services. In this paper, we mainly focus on the instance services provided by Amazon Web Service. Amazon has offered 23 types of EC2 instances [8], which offers different compute, memory, and storage capabilities, to users. For each type, there are three kinds of pricing options, including *on-demand instance*, *reserved instance* and *spot instance*. Here, we focus on on-demand instance and reserved instance. For the simpleness, we only consider single instance type: m1.small. From the Amazon website, we know that the on-demand price of m1.small in US East is $0.060 per hour. Amazon provides three reserved instance types to address users' projected utilization of the instance: *Heavy Utilization*, *Medium Utilization*, and *Light Utilization*. And in each utilization level, there are 2 terms: 1 year term and 3 year term. However, using the Reserved Instance Marketplace platform, we have the flexibility to purchase Reserved Instances from AWS Reserved Instance Marketplace Sellers for terms ranging between 1 month to 36 months (depending on available selection) [10]. Since the Google cluster usage trace only spans one month, we only consider month-level reserved instances. Based on the upfront fees and hourly fees for 1 and 3 year terms listed in AWS website, we assume 1, 2, and 3 month terms as shown in table II, which are based on the break-event point of 1 and 3 year terms. The break-event point denotes the smallest ratio between execution period over total period, by which the reserved instance can benefit compared to the on-demand instance. For example, if we rent a 1 month term light utilization m1.small, we will benefit from it when the usage hours are more than 192 hours, which is 26.78% of one month term.

Google Cluster-Usage Trace. Workload traces in public IaaS clouds are often confidential. Recently, Google released limited system traces from one of their production clusters. This trace is about scheduler request and utilization data across a large-scale cluster with 12583 machines over 29 days. This trace has been widely used in the academic research. The trace contains 180GB of resource demand/usage information of 933 users, 650 thousands of jobs, 25 millions of tasks. Tasks are organized into jobs. A job is comprised of one or more identical tasks with the same resource requirement. Although Google cluster usage trace is not a real workload of public IaaS cloud, it reflects the computation requirements of Google engineers and services, which can represent demands of cloud users to a certain extent.

Since the instance billing cycle is hour level, we select 39068 long jobs spanning longer than 1 hour from the trace to evaluate the performance of our cloud brokerage service. Because the data has been obfuscated to hide exact machine configurations, exact number of CPU cores and bytes of memory are unavailable; instead, CPU and memory size measurements are normalized to the configuration of the largest machines. The 12583 machines in the Google cluster consist of there kinds of CPU configuration: 0.25, 0.5 and 1. Since 92.7% of machines has CPU configuration 0.5, we suppose the instance m1.small has the same computing capacity as the Google machine with 0.5 CPU configuration. Among all the 39068 jobs, 39.8% of jobs have more than one task. 10.73% of these multiple-task jobs have constraints on different machine requirement, which indicates that a task must be scheduled to execute on a different machine than any other currently running task in the same job.

B. Results and Analysis

In the evaluation, we first need to determine how many instance-hours would each job require if it were to execute in a public IaaS cloud. Since there exist single-task jobs and multiple-task jobs, we would reschedule the jobs into instances with the rules: for single-task job, the tasks within the same job are consolidated into the same instance as much as possible; Whereas, for multiple-task job, tasks which can not share the same machine (e.g., tasks of MapReduce) are scheduled to different instances.

Effectiveness of Scheduling Module. Fig. 5 shows the actual number of instances needed per hour under different values of *slackness*, as defined before. When $slackness = 1$, the demand curve is highly fluctuant since the cloud brokerage should meet the demands of all jobs immediately when receiving the job requests. As the value of *slackness* increases, our scheduling module with *exactPickup* can decrease the maximum instances needed per hour, thereby smooth aggregated demand curve (we will show that this will translate into lower prices in the next evaluation). In real scenarios, different customers may specify different *slackness* for their jobs. Thus, we also consider the case of $slackness \in rand[1, 2]$, meaning that the *slackness* for each job is chosen uniformly at random between 1 and 2. We can observe that our broker service is still effective in this case.

Cloud Cost Reduction. To demonstrate the cost reduction of our cloud brokerage service, we conduct 6 experiments as

Experiment ID	Method Used	Slackness
1	Straightfoward Reservation	N/A
2	Per-Job Optimum Reservation	N/A
3	Our Cloud Brokerage Service	1
4		1.5
5		2
6		rand(1,2)

TABLE III
Specification of the six experiments.

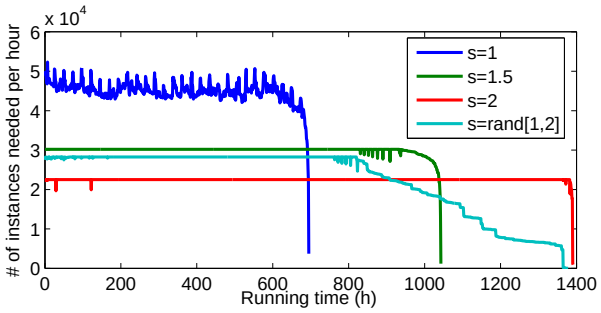

Fig. 5. The number of instances per hour obtained by our reservation module under different slackness s.

specified in table III. In the Exp.1, we use a straightforward reservation strategy that uses on-demand instances for all jobs. In Exp.2, we use a per-job optimum reservation strategy that considers each job separately and chooses the proper option of instance based on the instance-hours needed by each job. Exp.3-6 corresponds to the cost when using our brokerage service under different values of *slackness*.

From Fig. 6(a) we observe that the per-job optimum reservation strategy is better than straightforward reservation strategy, by reducing the cost by around 32%. Exp.3 shows that even all jobs are tight (with *slackness* being 1), our brokerage service still performs better than the per-job optimum approach, reducing the total cost by 42% compared with the staightforward reservation. We can also observe that more cost saving can be achieved with the increase of job *slackness*. Even under the case of $slackness = rand[1, 2]$ (Exp.6), our brokerage service can still achieve around 51% saving.

Fig. 6(c) and Fig. 6(b) shows the number of instances rented and the number of instance actually used. As we can see, none of Exp.1-6 use the option-*2*, option-*4*, option-*5*, option-*6* or option-*9* instance. In Fig. 6(b), we observe that the cost saving ratio increases from Exp.1 to Exp.5, since the instance usage of longer-term (and thus cheaper) option increases. Although Exp.4 and Exp.5 have the same amount of usage of option-*8* instance, the number of option-*8* reserved instance rented in Exp.4 is 7699 more than Exp.5, as shown in Fig. 6(c), and thus has a lower cost saving ratio. By comparing the results of Exp.3-5 in Fig. 6(b), we are confirmed the effectiveness of the the scheduling module, that is the smoother the aggregated demand curve is, the more opportunities for using cheaper reserved instances.

We try different values of *slackness* to show its effect on the performance of our brokerage service. In Fig. 6(d), we can see that the cost saving ratio becomes higher with the increase of *slackness*. The cost saving ratio gradually approaches the upper bound of 65%, which can only be achieved by solely reserving the option-*9* instances, and fully utilize them during the reservation period.

Discussion: Mutual Benefits for Brokers and Users The above results show that our cloud brokerage service can reduce the cost of IaaS cloud usage. To attract more cloud customers to use our brokerage service, we can offer discounts. As shown in Fig. 6(a), the per-job optimum strategy incurs a total cost of $1,268,603.527. If we can offer 25% discount compared with the original pricing options (including on-demand and reserved instance as provided by Amazon Web Service), the total cost of per-job optimum reservation strategy will drop to $951,452.645. In the case of $slackness = rand[1, 2]$, the total cost incurred by our brokerage service is $900,570.842. It means that our brokerage service can rent the instances with the original price options from Amazon Web Service, and charge the users with 25% discount, and we can still gain a profit of $50,881.803, when $slackness = rand[1, 2]$. Besides, the Amazon Web Service also provides reserved instance volume discounts [13], which can further reduce the upfront fees and usage fees for future renting of reserved instance after a user has rented a large number of reserved instance in an AWS region. So there is a large space for our brokerage service to offer discount for users, while still making profits.

VI. Conclusion

In this paper, we proposed a new cloud brokerage service to minimize the cloud cost incurred by deadline-constrained batch jobs. By exploiting the *slackness* of these jobs, we design a scheduling module to smooth the aggregated demand curve, and design a reservation module to obtain the renting strategy with the minimum cost. We test our brokerage service with real traces from one of Google's large-scale computing cluster. Evaluation results show that our cloud brokerage can significantly reduce the total cost for web service providers running deadline-constrained batch jobs. The online reservation mechanism will be my future work.

Acknowledgment

The work described in the paper is supported by the National Grand Fundamental Research 973 Program of China (No. 2010CB328105), the Tsinghua University Initiative Scientific Research Program (No. 20121087999).

References

[1] Cloud Services Brokerage Company List and FAQ, http://www.cloudbroker.com.
[2] N. Jain, I. Menache, and O. Shamir, "Learning-Based Resource Allocation for Delay-Tolerant Batch Computing," Microsoft technical report.
[3] A. Wieder, P. Bhatotia, A. Post and R. Rodrigues, "Orchestrating the Deployment of Computations in the Cloud with Conductor," in *NSDI*, 2012.
[4] H. Zhao, M. Pan, X. Liu, X. Li and Y. Fang, "Optimal Resource Rental Planning for Elastic Applications in Cloud Market," in *IEEE IPDPS*, 2012.
[5] M. Zafer, Y. Song, and K. W. Lee, "Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs," in *IEEE CLOUD*, 2012.
[6] Y. Song, M. Zafer, and K. W. Lee, "Optimal Bidding in Spot Instance Market," in *IEEE INFOCOM*, 2012.

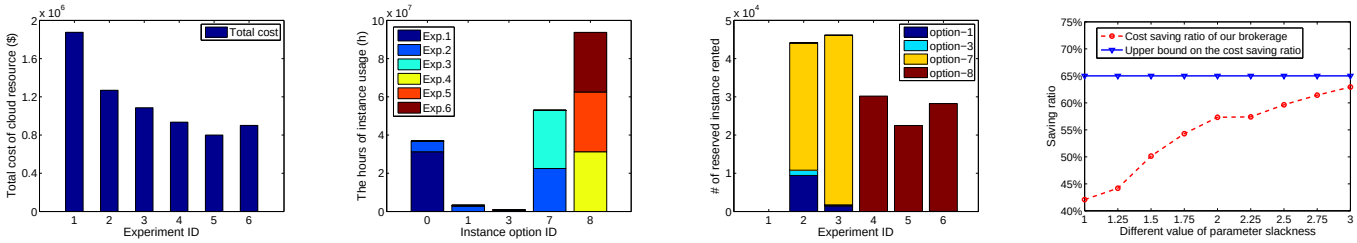

Fig. 6. (a) the total cost of each experiment. (b) the number of different instances used in each experiment. (c) the number of different reserved instances rented in each experiment. (d) the cost saving ratio of our brokerage service under different slackness parameter.

[7] W. Wang, D. Niu, B. Li, and B. Liang, "Dynamic Cloud Resource Reservation via Cloud Brokerage," in *IEEE ICDCS*, 2013.

[8] http://www.aws.amazon.com/ec2/pricing/

[9] C. P. Low, "An approximation algorithm for the load-balanced semi-matching problem in weighted bipartite graphs," in *Information Processing Letters*, vol. 100, no. 4, pp. 154-161, 2006.

[10] Amazon Web Services, Inc. "Amazon Elastic Compute Cloud User Guide", https://aws.amazon.com/documentation/ec2/

[11] J. Wilkes and C. Reiss, "Google Cluster-Usage Traces", https://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1.

[12] T. Hacker, K. Mahadik, "Flexible Resource Allocation for Reliable Virtual Cluster Computing Systems", in *SC*, 2011.

[13] B. Palanisamy, A. Singh, L. Liu and B. Langston, "Cura: A Cost-Optimized Model for MapReduce in a Cloud", in *IPDPS*, 2013.

[14] S. Alamdari, T. Biedl, T.M. Chan, E. Granty, K.R. Jampaniz, S. Keshav, A. Lubiw, and V. Pathak, "Strip Packing with Slicing".

[15] C. Martel, "Preemptive Scheduling with Release Times, Deadlines, and Due Times", *Journal of the Association for Computing Machinery*, vol. 29, no. 3, pp. 812-829, 1982.

APPENDIX A
PROOF OF THEOREM 1

Proof: We first introduce the following machine scheduling problem [14].

Problem 1. *(Load-balanced Semi-matching Problem (LBSMP))* Given a bipartite graph $G = (U \cup V, E)$ where $E \subseteq U \times V$. Every vertex $u \in U$ is associated with a non-negative weight *w(u)*. A set of edges $X \subseteq E$ is a *semi-matching* on the bipartite graph $G = (U \cup V, E)$ if each vertex $u \in U$ is incident to exactly one edge in X. The sum of weights of vertices $u \in U$ which are assigned to some vertex $v \in V$ under the *semi-matching* X is referred as the load of of vertex v. The Load-balanced Semi-matching Problem ($LBSMP$) is to find a *semi-matching* that minimizes the maximum loads of the vertex $v \in V$.

We next show that a special case of our optimization problem is identical to *LBSMP*. In particular, consider a special case of our optimization problem whereby the instances required by each job should be satisfied in one hour, i.e., $D_i = c_i, \forall i \in [1, M]$. From the viewpoint of the special case of our optimization problem, we may consider the vertex set U as a set of job requests and the vertex set V as a set of time slots $t(\forall t \in [1, T])$ where jobs can be assigned. An edge exists between vertice $u \in U$ and vertice $v \in V$ if the job corresponding to vertice u can be assigned in the time slot, which is between the start time and due date of the job, corresponding to the vertice v. For each vertice $u \in U$, its weight *w(u)* corresponds to the instances required by the job u. It is easy to see that the special case of our optimization problem is identical to the $LBSMP$. Since the $LBSMP$ is known to be NP-Complete [9], our optimization problem is also NP-Complete. ■

APPENDIX B
COMPLEXITY ANALYSIS OF PROBLEM (2)

In the following, we use the case of single reservation option to show that the direct dynamic programming method requires an exponential complexity in both time and space. In the case of single reservation option, we suppose the parameter relate to the option is (γ, α, τ). The state at time t is represented as $s_t = (x_t^0, x_t^1, \ldots, x_t^{\tau-1}, n_t)$, where x_t^i denotes the number of reserved instances rented at time $t-i$ for $i \in [0, \tau-1]$, and n_t denotes the number of reserved instances that are still effective at time t. So the state at time t: $s_t = (x_t^0, x_t^1, \ldots, x_t^{\tau-1}, n_t)$ can be directly transformed from the state at time $t-1$: $s_{t-1} = (x_{t-1}^0, x_{t-1}^1, \ldots, x_{t-1}^{\tau-1}, n_{t-1})$ by the following transformation rules.

$$x_t^i = x_{t-1}^{i-1}, \qquad \forall i \in [1, \tau-1] \tag{7}$$

$$n_t = n_{t-1} - x_{t-1}^{\tau-1} + x_t^0 \tag{8}$$

Then we define $C(s_t)$ to be the minimum cost of serving the demands $V_0, V_1, \ldots, V_t$ up to time t. Since the state in current time is only dependent on the state in the previous time, we get the recursive Bellman equation.

$$C(s_t) = \min_{s_{t-1}} \{C(s_{t-1}) + x_t^0 \gamma + \min(V_t, n_t)\alpha + (V_t - n_t)^+ \beta\} \tag{9}$$

where Y^+ is used to denote $\max\{0, Y\}$ and the minimization is over all states in time $t-1$ that can transmit to state s_t. The minimum cost of reaching state s_t is given by the minimum cost of reaching state s_{t-1} plus the cost for serving the demand V_t. To satisfy the demand V_t at time t, the brokerage needs to rent x_t^0 new reserved instances, use $\min(V_t, n_t)$ reserved instances and launch $(V_t - n_t)^+$ on-demand instances. The calculation process is from time 0 to time $T-1$. In every stage, we enumerate the possible states and search for the minimum cost for them based on the minimum cost of state in the previous stage. As we can see, both of the time complexity and space complexity are exponential. The simple case is so difficult to work out, let along the multiple reservation options.