Dynamic Cloud Resource Reservation via Cloud Brokerage

Wei Wang*, Di Niu†, Baochun Li*, Ben Liang*
* Department of Electrical and Computer Engineering, University of Toronto
† Department of Electrical and Computer Engineering, University of Alberta
Email: weiwang@eecg.toronto.edu, dniu@ualberta.ca, bli@eecg.toronto.edu, liang@utoronto.ca

***Abstract*—Infrastructure-as-a-Service clouds offer diverse pricing options, including on-demand and reserved instances with various discounts to attract different cloud users. A practical problem facing cloud users is how to minimize their costs by choosing among different pricing options based on their own demands. In this paper, we propose a new *cloud brokerage service* that reserves a large pool of instances from cloud providers and serves users with price discounts. The broker optimally exploits both pricing benefits of long-term instance reservations and multiplexing gains. We propose dynamic strategies for the broker to make instance reservations with the objective of minimizing its service cost. These strategies leverage dynamic programming and approximate algorithms to rapidly handle large volumes of demand. Our extensive simulations driven by large-scale Google cluster-usage traces have shown that significant price discounts can be realized via the broker.**

I. Introduction

Infrastructure-as-a-Service (IaaS) cloud enables IT services to elastically scale *computing instances* to match their time-varying computational demands. Thanks to the economies of scale, an IaaS cloud is capable of offering such on-demand computational services at a low cost [1]. Cloud users usually pay for the usage (counted by the number of instance-hours incurred) in a pay-as-you-go model, and are therefore freed from the prohibitive upfront investment on infrastructure, which is usually over-provisioned to accommodate peak demands.

A cloud provider prefers users with predictable and steady demands, which are more friendly to capacity planning. In fact, most cloud providers offer an additional pricing option, referred to as the *reservation option*, to harvest long-term risk-free income. Specifically, this option allows the user to prepay a one-time reservation fee and then to reserve a computing instance for a long period (usually in the order of weeks, months or years), during which the usage is either free or charged under a significant discount [2], [3], [4], [5], [6], [7]. If fully utilized, such a *reserved instance* can easily save its user more than 50% of the expense.

However, whether and how much a user can benefit from the reservation option critically depends on its demand pattern. Due to the prepayment of reservation fees, the cost saving of a reserved instance is realized only when the accumulated instance usage during the reservation period exceeds a certain threshold (varied from 30% to 50% of the reservation period). Unless heavily utilized, the achieved saving is not significant. For this reason, users with sporadic and bursty demands only launch instances on demand.

Unfortunately, on-demand instances are economically inefficient to users, not only because of the higher rates, but also because there is a fundamental limit on how small the billing cycle can be made. For example, Amazon Elastic Compute Cloud (EC2) charges on-demand instances based on running hours. In this case, an instance running for only 10 minutes is billed as if it were running for a full hour [2], [3], [4], [5], [6], [7]. Such billing inefficiency becomes more salient for cloud providers adopting longer billing cycles (e.g., in VPS.NET [8], even a single hour is charged at a daily rate), and for sporadic demands with a substantial amount of partial usage.

In general, to what extent a cloud user can enjoy cost savings due to reservation, while avoiding its inefficiency due to coarse-grained billing cycles, is limited by its own demand pattern. A natural question arises: Can we go beyond this limitation to further lower the cost for all cloud users? Especially, can users with any demand pattern benefit from reservation options while reducing the costs of instance-hours that are not fully utilized?

In this paper, we propose a *cloud brokerage* service to address these challenges. Instead of trading directly with cloud providers, a user will purchase instances on demand from our *cloud broker*, who has reserved a large pool of instances from IaaS clouds. Intuitively, the cloud broker leverages the "wholesale" model and the pricing gap between reserved and on-demand instances to reduce the expenses at all the users. More importantly, the broker can optimally coordinate different users to achieve additional cost savings. On one hand, when the broker aggregates user demands, bursts in demand will be smoothed out, leading to steadier aggregated demand that is amenable to the reservation option. On the other hand, for multiple users, each incurring partial usage during the same billing cycle, the broker can time-multiplex them with the bet that one user's wasted idle time in the billing cycle can be "recycled" to serve other users. It is through these mechanisms that the broker reduces the expenses for cloud users, while turning profits for itself.

However, a major challenge in operating such a broker is the decision on how many instances the broker should reserve, how many instances it should launch on demand, and when to reserve, as the demands change dynamically over time. To solve this challenge, we formulate the problem of dynamic instance reservation given user demand data, and derive the optimal reservation strategy via dynamic programming. Unfortunately, such dynamic programming is computationally prohibitive. We propose two efficient approximate algorithms that we prove to offer worst-case cost guarantees. We also propose an effective online algorithm that makes reservation decisions without having access to future demand information.

We conduct large-scale simulations driven by 180 GB of Google cluster usage traces [9] involving 933 cloud users' workload in a recent month. We empirically evaluate the aggregate and individual cost savings brought forth by the broker, under the proposed reservation strategies. Our results suggest that the broker is the most beneficial for users with medium demand fluctuations, reducing their total expenses by more than 40%. As for general users, 70% of them receive discounts more than 25%. This amounts to a total saving of over $100K for all the users tested in one month. Such cost savings are more significant in IaaS clouds adopting longer reservation periods or longer billing cycles.

The remainder of this paper is organized as follows. We propose our cloud broker in Sec. II, and formulate the dynamic resource reservation problem in Sec. III. We use dynamic programming to characterize the optimal solutions in Sec. IV and point out the related complexity issues. In Sec. V, we propose efficient approximate solutions to the reservation problem. The empirical evaluations based on real-world traces are presented in Sec. VI. We discuss related work in Sec. VII and conclude the paper in Sec. VIII.

II. A Profitable Cloud Broker

Most IaaS clouds provide users with multiple purchasing options, including on-demand instances, reserved instances, and other instance types [2], [3], [4], [5], [6], [7]. *On-demand instances* allow users to pay a fixed rate in every *billing cycle* (e.g., an hour) with no commitment. For example, if the hourly rate of an on-demand instance is p, an instance that has run for n hours is charged $n \cdot p$. As another purchasing option, a *reserved instance* allows a user to pay a one-time fee to reserve an instance for a certain amount of time, with reservation pricing policies subtly different across cloud providers. In most cases, the cost of a reserved instance is *fixed*. For example, in [3], [4], [5], [6], [7], [8], the cost of a reserved instance is equal to the reservation fee. As another example, in Amazon EC2 [2], the cost of a Heavy Utilization Reserved Instance is a reservation fee plus a heavily discounted hourly rate charged over the entire reservation period, no matter whether it is used or not. EC2 also offers other reservation options (e.g., Light/Medium Utilization Reserved Instances), with cost dependent on the actual usage time of the reserved instance. Throughout the paper, we limit our discussions to reservations with fixed costs, which represent the most common cases in IaaS clouds.

We propose a profitable cloud broker that can save expenses for cloud users. As illustrated in Fig. 1, the broker reserves a large pool of instances from the cloud providers to serve a major part of incoming user demand, while accommodating request bursts by launching on-demand instances. The broker pays IaaS clouds to retrieve instances while collecting revenue from users through its own pricing policy. From the perspective of users, their behavior resembles launching instances "on demand" provided by the broker, yet at a lower price. The broker can reduce the total service cost and reward the savings to users mainly through demand aggregation, with the following benefits:

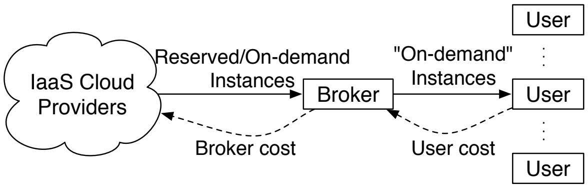


Fig. 1. The proposed cloud broker. Solid arrows show the direction of instance provisioning; dashed arrows show the direction of money flow.

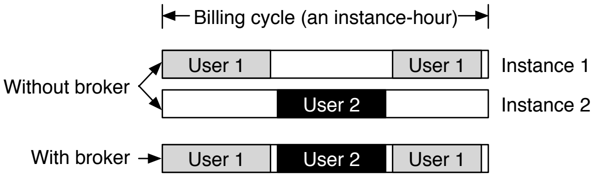


Fig. 2. The broker can time-multiplex partial usage from different users in the same instance-hour. In this case, serving two users only takes one instance-hour, instead of two.

Better exploiting reservation options: The broker aggregates the demand from a large number of users for service, smoothing out individual bursts in the aggregated demand curve, which is more stable and suitable for service through reservation. In contrast, individual users usually have bursty and sporadic demands, which are not friendly to the reservation option.

Reducing wasted cost due to partial usage: Partial usage of a billing cycle always incurs a full-cycle charge, making users pay for more than what they use. As illustrated in Fig. 2, without the broker, Users 1 and 2 each have to purchase one instance-hour, and pay the hourly rate even if they only use the hour partially. In contrast, the broker can use a single instance-hour to serve both users by time-multiplexing their usage, reducing the total service cost by one half. Such a benefit can be realized at the broker by scheduling the aggregated user demands to the pooled instances. It is worth noting that such a benefit is conditioned on whether switching users on an instance incurs additional cost charged by the cloud, which we will further discuss in Sec. VI-F.

Enjoying volume discounts: Most IaaS clouds offer significant volume discounts to those who have purchased a large number of instances. For example, Amazon provides 20% or even higher volume discounts in EC2 [2]. Due to the sheer volume of the aggregated demand, the cloud broker can easily qualify for such discounts, which further reduces the cost of serving all the users.

The main technical challenge to operate such a brokerage service is how to serve the aggregated user demands at the minimum cost, by dynamically and efficiently making instance reservation decisions based on the huge demand data collected from users. This will be the theme of the following sections.

III. Dynamic Resource Reservation

In this section, we formulate the broker's optimal instance reservation problem to satisfy given demands, with an objective of cost minimization. The broker asks cloud users to submit their demand estimates over a certain horizon, based on which dynamic reservation decisions are made. Note that even if a user trades directly with cloud providers, it needs to estimate its future demand to decide how many instances to reserve at a particular time. In the case where users are unable to estimate demand at all, we propose an online reservation strategy in Sec. V-C to make decisions based on history.

Suppose cloud users submit to the broker their demand estimates up to time T into the future (in terms of billing cycles). The broker aggregate all the demands. Suppose it requires d_t instances in total to accommodate all the requests at time t, $t = 1, 2, \dots, T$. The broker makes a decision to reserve r_t instances at time t, with $r_t \geq 0$. Each reserved instance will be effective from t to $t+\tau-1$, with τ being the *reservation period*.

At time t, the number of reserved instances that remain effective is $n_t = \sum_{i=t-\tau+1}^{t} r_i$. Note that these n_t reserved instances may not be sufficient to accommodate the aggregate demand d_t. The broker thus needs to launch $(d_t - n_t)^+$ additional on-demand instances at time t, where $X^+ := \max\{0, X\}$.

Let γ denote the one-time reservation fee for each reserved instance, and p denote the price of running an on-demand instance per billing cycle. Hence, the total cost to accommodate all the demands $d_1, \dots, d_T$ is

$$\sum_{t=1}^{T} r_t\gamma + \sum_{t=1}^{T}(d_t - n_t)^+ p \ , \qquad (1)$$

where the first term is the total cost of reservations and the second is the cost of all on-demand instances. The broker's problem is to make dynamic reservation decisions $r_1, \dots, r_T$ to minimize its total cost, i.e.,

$$\min_{\{r_1,\dots,r_T\}} \quad \text{cost} = \sum_{t=1}^{T} r_t\gamma + \sum_{t=1}^{T}(d_t - n_t)^+ p \ . \qquad (2)$$

Problem (2) is integer programming. In general, complex combinatorial methods are needed to solve it.

IV. Dynamic Programming: the Optimality and Limitations

In this section, we resort to dynamic programming to characterize the optimal solution to problem (2). Using a set of recursive Bellman equations, the original combinatorial optimization problem can be decomposed into a number of subproblems, each of which can be solved efficiently. However, we also point out that computing such a dynamic programming is practically infeasible, and is highly inefficient to handle a large amount of data.

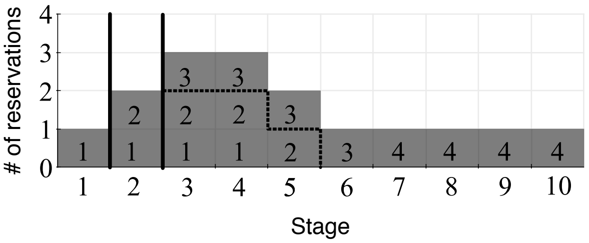


Fig. 3. State illustration. The reservation period is $\tau = 4$. All four reservations are highlighted as the shaded area. At stage 2, $\mathbf{s}_2 = (2, 2, 2, 1)$.

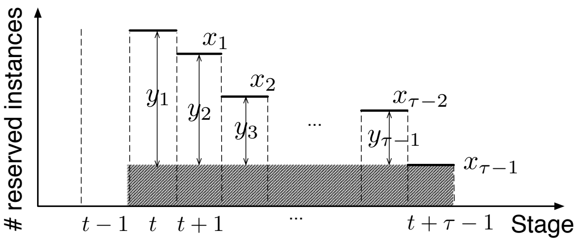


Fig. 4. Illustration of a transition into state $\mathbf{s}_t = (t, x_1, \dots, x_{\tau-1})$ from state $\mathbf{s}_{t-1} = (t-1, y_1, \dots, y_{\tau-1})$ according to (3). The shaded area stands for the $x_{\tau-1}$ reservations made in hour t.

A. Dynamic Programming Formulation

We start by defining stages and states. The decision problem (2) consists of T *stages*, each representing a billing cycle. A *state* at stage t is denoted by a τ-tuple $\mathbf{s}_t := (t, x_1, \dots, x_{\tau-1})$, where x_i denotes the number of instances that are reserved no later than t and remain effective at stage $t+i$, for $i = 1, \dots, \tau-1$. Here, we use a τ-tuple to represent a state because no instance reserved before or at stage t will remain effective after stage $t+\tau-1$. And it is easy to check that $x_1 \geq \cdots \geq x_{\tau-1}$. For example, in Fig. 3, three instances are reserved at stage 1, 2, and 3, respectively, with a reservation period $\tau = 4$. We see that at stage 2, $\mathbf{s}_2 = (2, 2, 2, 1)$, where the second element is 2 because two instances are reserved no later than stage 2 and remain effective at stage 3.

We note that given state $\mathbf{s}_{t-1} := (t-1, y_1, \dots, y_{\tau-1})$ at the previous stage, the current state $\mathbf{s}_t := (t, x_1, \dots, x_{\tau-1})$ is independent of the states $\mathbf{s}_{t-2}, \mathbf{s}_{t-3}, \dots$. In fact, $\mathbf{s}_t$ can be characterized by $\mathbf{s}_{t-1}$, with *state transition equations*:

$$x_i = y_{i+1} + x_{\tau-1}, \quad i = 1, \dots, \tau-2. \qquad (3)$$

To see the rationale behind (3), let us consider a state $\mathbf{s}_t$ in Fig. 4. At stage $t+\tau-1$, there are $x_{\tau-1}$ reservations that remain effective. Clearly, all these reservations are made at stage t (because the reservations made before stage t have all expired at $t+\tau-1$), i.e., $r_t = x_{\tau-1}$, r_t being the number of instances reserved at stage t. These $x_{\tau-1}$ reserved instances (the shaded area in Fig. 4) will add to the effective reservations starting from stage t. Since $\mathbf{s}_{t-1} := (t-1, y_1, \dots, y_{\tau-1})$, we know that at stage $t+i$, there remain y_{i+1} effective reservations made before t, $i = 1, \dots, \tau-2$. Adding the $x_{\tau-1}$ instances reserved at stage t leads to $x_i = y_{i+1} + x_{\tau-1}$ instances that are reserved no later than stage t and remain effective at stage $t+1$.

Define $V(\mathbf{s}_t)$ as the minimum cost of serving the demands $d_1, \dots, d_t$ up to stage t, conditioned on that state $\mathbf{s}_t$ is reached at stage t. Then we have the following recursive Bellman equations:

$$V(\mathbf{s}_t) = \min_{\mathbf{s}_{t-1}} \big\{ V(\mathbf{s}_{t-1}) + c(\mathbf{s}_{t-1}, \mathbf{s}_t) \big\}, \quad t > 0, \tag{4}$$

where the minimization is over all states $\mathbf{s}_{t-1}$ that can transit to $\mathbf{s}_t$ following (3). In (4) the minimum cost of reaching $\mathbf{s}_t$ is given by the minimum cost of reaching a previous state $\mathbf{s}_{t-1}$ plus a *transition cost* $c(\mathbf{s}_{t-1}, \mathbf{s}_t)$, minimized over all possible previous states $\mathbf{s}_{t-1}$. The transition cost is defined as

$$\begin{cases} c(\mathbf{s}_{t-1}, \mathbf{s}_t) := \gamma x_{\tau-1} + p(d_t - y_1 - x_{\tau-1})^+, \\ \mathbf{s}_t := (t, x_1, \dots, x_{\tau-1}), \\ \mathbf{s}_{t-1} := (t-1, y_1, \dots, y_{\tau-1}) \ , \end{cases} \tag{5}$$

where the transition from $\mathbf{s}_{t-1}$ to $\mathbf{s}_t$ follows (3).

The rationale of (5) is straightforward. As has been noted, $x_{\tau-1}$ instances are reserved at stage t, and y_1 instances are reserved before t and remain effective at t. To accommodate the demand d_t at stage t, the broker needs to launch $(d_t - y_1 - x_{\tau-1})^+$ on-demand instances. As a result, the broker pays $\gamma x_{\tau-1}$ to reserve the $x_{\tau-1}$ instances, and $p(d_t - y_1 - x_{\tau-1})^+$ for on-demand instances, leading to a transition cost of (5).

The boundary conditions of (4) are given by

$$V(\mathbf{s}_0) := \gamma z_1, \qquad \text{for all } \mathbf{s}_0 := (0, z_1, \dots, z_{\tau-1}) \ , \tag{6}$$

since an initial state $\mathbf{s}_0 := (0, z_1, \dots, z_{\tau-1})$ indicates that the broker has already reserved z_1 instances at the beginning (time 1) and paid γz_1.

Through the above analysis, we have converted problem (2) into an equivalent dynamic programming problem:

Proposition 1: The dynamic programming defined by (3), (4), (5), and (6) gives an optimal solution to problem (2).

The proposed dynamic programming can be viewed as solving a canonical shortest path problem on a trellis graph. As illustrated in Fig. 5, a state $\mathbf{s}_t$ is represented by a node at stage t. If state $\mathbf{s}_{t-1}$ can transit to state $\mathbf{s}_t$, i.e., they satisfy the state transition equations (3), then node $\mathbf{s}_{t-1}$ is connected to node $\mathbf{s}_t$ by an edge with length $c(\mathbf{s}_{t-1}, \mathbf{s}_t)$. Therefore, $V(\mathbf{s}_t)$ is the length of a shortest path from a node at stage 0 to node $\mathbf{s}_t$.

To find the minimum resource provisioning cost, we start from stage 1 and compute $V(\mathbf{s}_1)$ for all $\mathbf{s}_1$ at stage 1, using (4). For each $\mathbf{s}_1$, the *optimal previous state* $\mathbf{s}_0^*$ on the shortest path to $\mathbf{s}_1$ is recorded, such that $V(\mathbf{s}_1) = V(\mathbf{s}_0^*) + c(\mathbf{s}_0^*, \mathbf{s}_1)$. We then step to stage 2 and for all $\mathbf{s}_2$, calculate $V(\mathbf{s}_2)$ while recording the optimal previous state $\mathbf{s}_1^*$. The computation proceeds *forward* through stages until reaching the last state $\mathbf{s}_T := (T, 0, \dots, 0)$, where we output the minimum resource provisioning cost as $V(\mathbf{s}_T)$. The optimal reservation decisions can be obtained by tracking *backward* from stage T to 0, retrieving the recorded optimal previous states $\mathbf{s}_{T-1}^*, \dots, \mathbf{s}_0^*$ one by one.

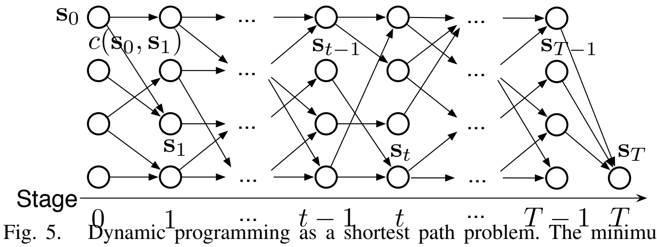


Fig. 5. Dynamic programming as a shortest path problem. The minimum cost is the output $V(\mathbf{s}_T)$.

B. The Curse of Dimensionality

Although the dynamic programming presented above is optimal, it is computationally infeasible for large data. This is because to derive the minimum cost $V(\mathbf{s}_T)$, one has to compute $V(\mathbf{s}_t)$ for all nodes $\mathbf{s}_t$ at all stages t. Since each node $\mathbf{s}_t$ is defined as a high dimensional tuple $(t, x_1, \dots, x_\tau)$, there exist *exponentially* many such nodes in the trellis graph. Therefore, going through all of them results in exponential time complexity. Also, since the computed $V(\mathbf{s}_t)$ has to be stored for every node $\mathbf{s}_t$, the space complexity is exponential as well. This is known as *the curse of dimensionality* suffered by all high-dimensional dynamic programming [10].

A classical method to handle the curse of dimensionality is to use *Approximate Dynamic Programming* (ADP) [10]. ADP estimates the minimum cost at each node first and refines such estimates in an iterative fashion. We next describe how ADP can be applied to our problem as well as its limitations.

Denote $\tilde{V}^{(0)}(\mathbf{s}_t)$ the initial estimate of $V(\mathbf{s}_t)$ and $\tilde{V}^{(k)}(\mathbf{s}_t)$ its updated estimate at iteration k. At each iteration k, referring to the trellis in Fig. 5, ADP picks a shortest path $P^k = \{\mathbf{s}_T^{(k)}, \dots, \mathbf{s}_0^{(k)}\}$ from stage T to 0, using the cost estimates $\tilde{V}^{(k-1)}(\mathbf{s}_t)$ from the previous iteration, and updates the cost estimates of the visited nodes. Specifically, we start from $\mathbf{s}_T^{(k)} := \mathbf{s}_T$ and proceed backwards. Suppose we are at node $\mathbf{s}_t^{(k)}$. The next node picked by the algorithm is

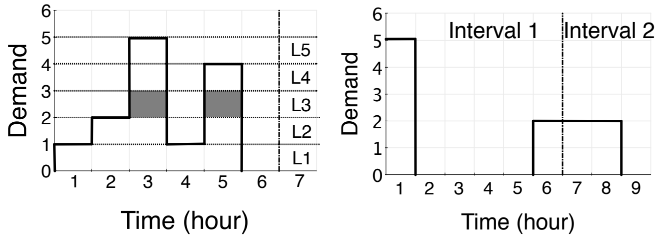
$$\mathbf{s}_{t-1}^{(k)} := \arg\min_{\mathbf{s}_{t-1}} \left\{ \tilde{V}^{(k-1)}(\mathbf{s}_{t-1}) + c(\mathbf{s}_{t-1}, \mathbf{s}_t^{(k)}) \right\} \ .$$

In the meantime, we update the estimate of $V(\mathbf{s}_t^{(k)})$ as

$$\tilde{V}^{(k)}(\mathbf{s}_t^{(k)}) := \min_{\mathbf{s}_{t-1}} \left\{ \tilde{V}^{(k-1)}(\mathbf{s}_{t-1}) + c(\mathbf{s}_{t-1}, \mathbf{s}_t^{(k)}) \right\} \ .$$

Then we move to the next node $\mathbf{s}_{t-2}^{(k)}$ until stage 0 is reached. For all nodes $\mathbf{s}_t$ that are not visited at iteration k, their estimates remain unchanged, i.e., $\tilde{V}^{(k)}(\mathbf{s}_t) := \tilde{V}^{(k-1)}(\mathbf{s}_t)$. We keep running the above iterations until no estimate has changed at an iteration.

It is known that ADP converges to the shortest path if the initial estimates $\tilde{V}^{(0)}(\mathbf{s}_t)$ are *optimistic*, i.e., they do not exceed the optimal solution $V(\mathbf{s}_t)$ [10]. However, if $\tilde{V}^{(0)}(\mathbf{s}_t)$ is too optimistic, e.g., $\tilde{V}^{(0)}(\mathbf{s}_t) = 0$, the convergence will be extremely slow. We will propose a smart way to set $\tilde{V}^{(0)}(\mathbf{s}_t)$ in Sec. V-A, leveraging the approximate algorithms proposed there. However, through extensive simulations, we will show in Sec. VI-B that although smart initial estimates significantly accelerate ADP, as an iterative method, its convergence speed



(a) $T = 5, \tau = 6$.

(b) $T = 8, \tau = 6$.

Fig. 6. The Periodic Decisions algorithm, with $\gamma = \$2.5$ and $p = \$1$. The algorithm is (a) optimal when $T \le \tau$, (b) not optimal when $T > \tau$.

Algorithm 1 Heuristic: Periodic Decisions

```
1. Segment $T$ into intervals $\{I_i\}$, each with length $\tau$.
2. for all intervals $I_i$ do
3.     Reserve $l$ instances at the beginning of this interval, such that $u_l^i \ge \gamma/p > u_{l+1}^i$, where $u_l^i := \sum_{t \in I_i} d_t^l$ is the utilization of level $l$ in interval $i$.
4. end for
```

is still not satisfactory to handle the large amount of demand data in our problem.

V. Approximate Algorithms

To overcome the prohibitive complexity of dynamic programming, in this section, we develop approximate algorithms to solve (2). These algorithms are highly efficient and are proved to have worst-case performance guarantees. Furthermore, we also propose an online reservation strategy which can be applied when future demand data is not available.

A. A 2-Competitive Heuristic

First, we present a simple heuristic that in the worst case, incurs twice the minimum cost, given any demands. We start off by dividing the demands into *levels*. Let $\bar{d} := \max_t d_t$ be the peak demand. As shown in Fig. 6a, the total demands are divided into $\bar{d} = 5$ levels, with level 1 being the bottom (labeled as "L1" in Fig. 6a) and level $\bar{d}$ being the top. Define d_t^l as the demand at time t in level l, such that $d_t^l = 1$ if $d_t \ge l$, and $d_t^l = 0$ otherwise. For example, in Fig. 6a, level 3 has demands only at time 3 and 5 (i.e., $d_3^3 = d_5^3 = 1$).

We now consider a special case, when all given demands are within a single reservation period, i.e., $T \le \tau$. In this case, it is sufficient to *make all the reservations at time* 1, since a reservation made anytime will remain effective for the entire horizon T. The question becomes how many instances to reserve at time 1.

Initially, we consider the first reserved instance that will be used to serve demands in level 1. Define *utilization* u_1 as the number of billing cycles where this reserved instance will be used. It is easy to check $u_1 = \sum_{t=1}^{T} d_t^1$. The use of this reserved instance will be well justified if the reservation fee satisfies $\gamma \le p u_1$; otherwise, launching it on demand would be more cost efficient.

Next, suppose $l - 1$ instances are already reserved in the bottom $l-1$ levels. We check if an instance should be reserved in level l. Define *utilization* u_l as the number of billing cycles where the lth reserved instance will be used, i.e.,

$$u_l := \sum_{t=1}^{T} d_t^l, \quad l > 0 \ . \tag{7}$$

For convenience, we let $u_0 := +\infty$ (for reasons to be clear). Again, the broker will adopt the lth reserved instance only if $\gamma \le p u_l$. Noting that u_l is non-increasing in l, we obtain a very simple optimal algorithm: reserve l instances at time 1, such that $u_l \ge \gamma/p > u_{l+1}$.

Fig. 6a shows an example with $\gamma = \$2.5$ and $p = \$1$. To run the algorithm, we first plot the demand curve d_t. We find u_l is the intersection area of a horizontal stripe in level l with the area below d_t, e.g., $u_3 = 2$, as shown by the shaded area. In this case, the optimal strategy is to reserve 2 instances in the bottom 2 levels, as $u_2 = 3 > 2.5 = \gamma/p$ while $u_3 = 2 < \gamma/p$.

When demands last for more than one reservation period, i.e., $T > \tau$, a natural idea is to extend the above algorithm by letting the broker make periodic decisions. We segment the time axis into intervals, each with the same length τ as the reservation period. The broker makes decisions for each τ-interval separately, only at the beginning of that interval, by running the above algorithm. This leads to the Periodic Decisions algorithm described in Algorithm 1. It is easy to check that Algorithm 1 only costs $O(\bar{d}T)$ time and $O(T)$ space.

Unfortunately, Algorithm 1 is not optimal, as reservations are placed *only* at the beginning of each interval, while in reality, a reservation can be made at any time. For example, in Fig. 6b, by running Algorithm 1, all instances are launched on demand, incurring a cost of \$11. However, the optimal strategy is to reserve 2 instances at the beginning of time 6, with a lower total cost of \$10 = 2.5×2+5. We bound the worst-case performance of Algorithm 1 in the following theorem.

Proposition 2: Algorithm 1 is *2-competitive*. That is, for any demands, the cost incurred by Algorithm 1 is no more than twice the minimum cost.

Proof: Divide time into non-overlapping intervals $\{I_k\}$, each of length τ. Let $I_k := [l_k, h_k]$ be the kth interval from $l_k = (k-1)\tau + 1$ to $h_k = k\tau$. We call a reservation *interval-based* if instances are reserved only at the beginnings of intervals. If r_t^I denotes the reservation decision of the interval-based strategy, then $r_t^I = 0, \forall t \ne l_k, k = 1, 2, \ldots$.

Given an arbitrary strategy, we construct the corresponding interval-based strategy as follows. Suppose at time t, the strategy reserves r_t instances, which are effective in $R_t = [t, t + \tau - 1]$. The corresponding interval-based strategy then reserves r_t instances at the beginnings of all I_k such that $I_k \cap R_t \ne \emptyset$. For simplicity, let $l_0 := 0$. We then have

$$r_t^I = \begin{cases} \sum_{i=l_{k-1}+1}^{h_k} r_i, & t = l_k \ , \\ 0, & t \ne l_k \ . \end{cases}$$

Denote by n_t and n_t^I the numbers of reserved instances that remain effective at time t for the given strategy and the constructed interval-based strategy, respectively. Clearly,

$n_t^I \geq n_t$. Let c and c^I be the costs of the given strategy and corresponding interval-based strategy, respectively. We have

$$\begin{aligned} c^I &= \textstyle\sum_{t=1}^{T} r_t^I \gamma + \sum_{t=1}^{T} (d_t - n_t^I)^+ p \\ &= \textstyle\sum_k \sum_{i=l_{k-1}+1}^{h_k} r_i \gamma + \sum_{t=1}^{T} (d_t - n_t^I)^+ p \\ &\leq 2 \textstyle\sum_{t=1}^{T} r_t \gamma + \sum_{t=1}^{T} (d_t - n_t)^+ p \leq 2c \ , \end{aligned} \qquad (8)$$

where the first inequality is due to $n_t^I \geq n_t$ and the last inequality is due to the definition of the cost c (see (1)). Equation (8) indicates that the cost of the constructed interval-based strategy is at most twice the original strategy.

We make an important observation, that Algorithm 1 incurs the minimum cost among all interval-based strategies, i.e., $c_1 \leq c^I$ with c_1 being the cost of Algorithm 1. By (8), we have $c_1 \leq c^I \leq 2c$, meaning that Algorithm 1 incurs no more than twice the cost of any strategy. ■

With the above performance guarantee, it is worth noting that Algorithm 1 can be used to compute the initial estimates for the aforementioned ADP algorithm and speed up its convergence. Specifically, let H_t be the cost incurred by Algorithm 1 for demands $d_1, \dots, d_t$ up to time t. For each state $\mathbf{s}_t := (t, x_1, \dots, x_{\tau-1})$, we set its initial estimate to be

$$\tilde{V}^{(0)}(\mathbf{s}_t) := \max\{H_t/2, \gamma x_1\} \ . \qquad (9)$$

Such an initial estimate is guaranteed to be optimistic, i.e., $\tilde{V}^{(0)}(\mathbf{s}_t) \leq V(\mathbf{s}_t)$ for all states $\mathbf{s}_t$, so that the ADP will converge to the optimality. To see this, we note that $H_t/2 \leq V(\mathbf{s}_t)$, as Algorithm 1 incurs no more than twice the minimum cost. Furthermore, by definition, at state $\mathbf{s}_t$, at least x_1 instances have been reserved, which implies $\gamma x_1 \leq V(\mathbf{s}_t)$. We will show in Sec. VI-B that the initial estimate (9) significantly accelerates ADP convergence.

B. An Improved Greedy Algorithm

Algorithm 1 divides problem (2) into reservation subproblems, each solved in a separate level. But in each level, the reservations are made only at the beginnings of the τ-intervals. A direct improvement of Algorithm 1 is to allow *arbitrary* reservation time in each level: we still consider from the bottom level up to the top, whereas in each level, we solve an optimal reservation problem using dynamic programming. Clearly, this strategy incurs less resource provisioning cost than Algorithm 1 in each level due to the relaxation on reservation time. However, just like Algorithm 1, such a strategy remains inefficient, since it ignores the dependencies across different levels.

A simple fix can incorporate inter-level dependencies: instead of reserving bottom-up, we start to make reservations in the top level $\bar{d}$ and proceed *top-down*. Every instance reserved in level l that is not used at time t will be passed over to the lower level $l-1$, so that it can be used to serve the demand at time t in level $l-1$. We then step to level $l-1$ and make optimal reservations there, taking into account the "leftover" reserved instances passed over from the upper level. Undoubtedly, the algorithm becomes more efficient, since each level tries to utilize such "leftover" reserved instances from upper levels. Note that this is not possible for a bottom-up approach, where no "leftover" reserved instances can be passed from a lower level up.

Algorithm 2 Greedy Reservation Strategy

```
Initialization: $m_t^{\bar{d}} := 0$ for all $t = 1, \dots, T$.
for $l = \bar{d}$ down to 1 do
    Make optimal reservations in level $l$ via dynamic programming defined by (10), (11), and (12).
    Update $m_t^{l-1}$.
end for
```

In each level of the above procedure, optimal reservations can be efficiently made via dynamic programming. Suppose before we make reservations in level l, m_t^l reserved instances are passed over from upper levels at time t, all of which can be utilized. Let $V_l(t)$ be the minimum cost of serving demands $d_1^l, \dots, d_t^l$ in level l up to time t . The Bellman equation is given by

$$V_l(t) = \min\{V_l(t-\tau) + \gamma, \ V_l(t-1) + c_l(t)\}. \qquad (10)$$

which chooses the minimum between two options. The *first* is to serve the demand d_t^l with an instance reserved in the current level l. The best strategy is to optimally serve demands up to time $t-\tau$ and reserve an instance at time $t-\tau+1$, incurring a cost of $V_l(t-\tau)+\gamma$. The *second* option is to optimally satisfy demands up to $t-1$ and serve the demand at t using an on-demand instance, with a cost $c_l(t)$ given by

$$c_l(t) = \begin{cases} p, & \text{if } d_t^l = 1 \text{ and } m_t^l = 0 \ , \\ 0, & \text{otherwise,} \end{cases} \qquad (11)$$

which means that we pay p to launch an on-demand instance only if there is demand at t, i.e., $d_t^l = 1$, yet no reserved instance is left over from upper levels to use at t, i.e., $m_t^l = 0$. Clearly, the boundary conditions are

$$V_l(t) = 0, \quad t \leq 0 \ . \qquad (12)$$

After reservations have been made in level l, we update m_t^{l-1}, the number of reserved instances to be passed to level $l-1$ at time t as follows: $m_t^{l-1} := m_t^l + 1$, if an instance is reserved in level l but is not used at time t; $m_t^{l-1} := m_t^l - 1$, if demand d_t^l is served using a reserved instance passed over from upper levels; and $m_t^{l-1} = m_t^l$, otherwise.

Algorithm 2 summarizes the above greedy reservation strategy, which has time complexity $O(\bar{d}T)$ and space complexity $O(T)$. This is because solving the dynamic programming in each level requires only $O(T)$ time and $O(T)$ space. As has been analyzed earlier in this subsection, in each level, Algorithm 2 is more cost efficient than Algorithm 1, leading to the following proposition:

Proposition 3: Algorithm 2 incurs a cost no more than Algorithm 1, and is thus 2-competitive.

C. An Online Reservation Strategy

Previous algorithms apply to the case that users submit their future demands up to time T. In the case when no demand

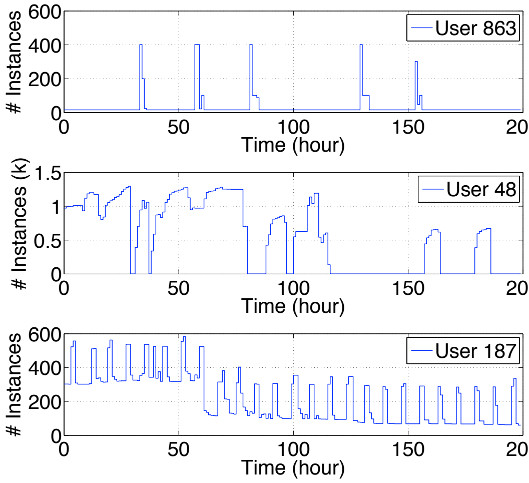


Fig. 7. The demand curves of three typical users.

estimate is available, we propose an *online strategy* that makes reservation decisions based only on history. Recall that n_t is the number of reserved instances that remain effective at time t. At time t, the online algorithm makes a reservation decision r_t based on the historical information $g_{t-\tau+1}, \ldots, g_t$ in the past reservation period, where

$$g_i := (d_i - n_i)^+, \quad i = t - \tau + 1, \ldots, t \ . \tag{13}$$

We call g_i the *reservation gap* between demand d_i and the number of reservations n_i that remain effective at time i. Clearly, all these g_i reservation gaps are filled by launching on-demand instances at time i.

We now make a "regret" to calculate how many *more* instances we should have reserved at time $t - \tau + 1$, if we knew that we would have to launch g_i instances on demand at time $i = t - \tau + 1, \ldots, t$. Such a "regret" can be computed by Algorithm 1, with outstanding gaps $g_{t-\tau+1}, \ldots, g_t$ as the input demands. Suppose the found value is x, we then reserve $r_t = x$ instances at the current time t. In the meantime, we update the reservation history *as if* we had reserved x additional instances at time $t - \tau + 1$, by setting $n_i := n_i + x$ for all $i = t - \tau + 1, \ldots, t$, which will be used in computing the next decision r_{t+1}.

Apparently, the computational complexity of the above online reservation strategy is the same as Algorithm 1 at every time t.

VI. Performance Evaluation

In this section, we conduct simulations driven by a large volume of real-world traces to evaluate the performance of the proposed brokerage service and reservation strategies, with an extensive range of scenarios.

A. Dataset Description and Preprocessing

Workload traces in public clouds are often confidential: no IaaS cloud has released its usage data so far. For this reason, we use Google cluster-usage traces that were recently released [9] in our evaluation. Although Google cluster is not a public IaaS cloud, its usage traces reflect the computing demands of Google engineers and services, which can represent demands of public cloud users to some degree. The dataset contains 180 GB of resource usage information of 933 users over 29 days in May 2011, on a cluster of 12,583 physical machines. In the Google traces, a user submits work in the form of *jobs*. A job consists of several *tasks*, each of which has a set of resource requirements on CPU, disk, memory, etc.

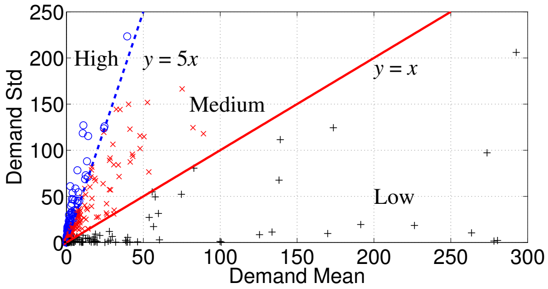


Fig. 8. Demand statistics and the division of users into 3 groups according to demand fluctuation level.

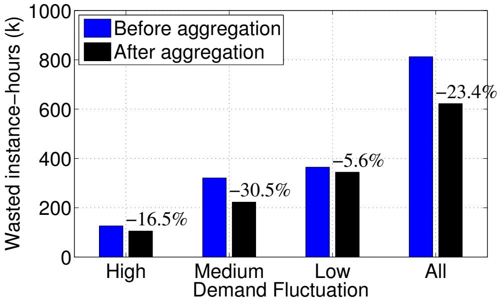
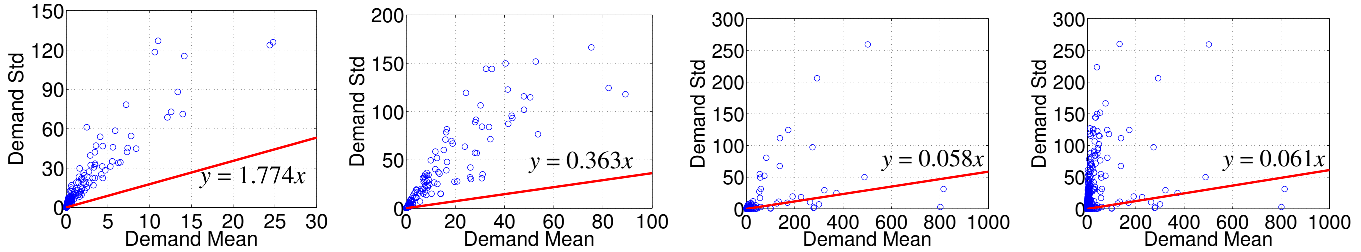


Fig. 10. Aggregation reduces the wasted instance-hours due to partial usage.

Instance Scheduling. We take such a dataset as input, and ask the question: How many computing instances would each user require if she were to run the same workload in a public IaaS cloud? It is worth noting that in Google cluster, tasks of different users may be scheduled onto the same machine, whereas in IaaS clouds each user will run tasks only on her own computing instances.

Therefore, we reschedule the tasks of each user onto instances that are exclusively used by this user. We set the instances to have the same computing capacity as Google cluster machines (most Google cluster machines are of the same computing capability, with 93% having the same CPU cycles), which enables us to accurately estimate the task run time by learning from the original traces.

For each user, we use a simple algorithm to schedule her tasks onto available instances that have sufficient resources to accommodate their resource requirements. Tasks that cannot share the same machine (e.g., tasks of MapReduce) are scheduled to different instances. (For simplicity, we ignore other complicated task placement constraints such as on OS versions and machine types.) Whenever the capacity of available instances is reached, a new instance will be launched. In the



(a) Group 1: high fluctuation (b) Group 2: medium fluctuation (c) Group 3: low fluctuation (d) All the users.

Fig. 9. Aggregation suppresses the demand fluctuation of individual users. Each circle represents a user. The line indicates the demand fluctuation level (the ratio between the demand standard deviation and mean) in the aggregate demand curve.

end, we obtain a demand curve for each user, indicating how many instances the user requires in each hour. Fig. 7 illustrates the demand curves of three typical users in the first 200 hours.

Pricing. Unless explicitly mentioned, we set the on-demand hourly rate to \$0.08, the same as Amazon EC2 small instances [2]. Since our data only spans one month, we assume each reservation is effective for one week, with a *full-usage discount* of 50%: the reservation fee is equal to running an on-demand instance for half a reservation period, which is a general pricing policy in most IaaS clouds [2], [4], [5], [8].

Group Division. To further understand the demand statistics of users, we calculate the demand mean and standard deviation for each user and plot the results in Fig. 8. As has been mentioned, to what extent a user can benefit from reservations critically depends on its demand pattern: the more fluctuated it is, the less is the benefit from using reserved instances. We hence classify all 933 users into the following three groups by the *demand fluctuation level* measured as the ratio between the demand standard deviation and mean:

Group 1 (High Fluctuation): Users in this group have a demand fluctuation level no smaller than 5. A typical user's demand is shown in the top graph of Fig. 7. There are 271 users in this group, represented by "o" in Fig. 8. These users have small demands, with a mean less than 30 instances.

Group 2 (Medium Fluctuation): Users in this group have a demand fluctuation level between 1 and 5. A typical user's demand is shown in the middle graph of Fig. 7. There are 286 users in this group, represented by "x" in Fig. 8. These users demand a medium amount of instances, with a mean less than 100.

Group 3 (Low Fluctuation): Users in this group have a demand fluctuation level less than 1, represented by "+" in Fig. 8. A typical user's demand is shown in the bottom graph of Fig. 7. Almost all big users with a mean demand greater than 100 belong to this group.

Our evaluations are carried out for each group. We start to quantify to what extent the aggregation smooths out demand bursts of individual users. Fig. 9 presents the results, with "o" being the statistics of individual users and the line representing the fluctuation level of the aggregated demand. We see from Fig. 9a and 9b that aggregating bursty users (i.e., users in Group 1 and 2) results in a steadier curve, with a fluctuation level much smaller than that of any individual user. For users that already have steady demands, aggregation does not reduce fluctuation too much (see Fig. 9c). In addition, Fig. 9d presents the result of aggregating all the users. In all cases, the aggregated demand is stabler and more suitable for service via reserved instances.

TABLE I
COMPARISONS IN TERMS OF COST AND CONVERGENCE.

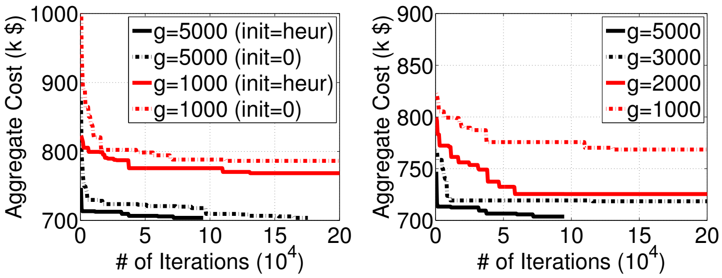
Algorithm	Cost (\$)	Converged	Run time[1] (sec)
ADP ($g = 5000$)	709,716	Yes	65
ADP ($g = 3000$)	718,483	No	388
ADP ($g = 2000$)	725,489	No	1645
ADP ($g = 1000$)	768,509	No	2732
Heuristic ($g = 1$)	702,305	N/A	1
Greedy ($g = 1$)	701,004	N/A	6

Another benefit of demand aggregation is that it reduces the wasted instance-hours incurred by partial usage. To see this, for each user, we count the total time during which it is billed but does not run any workload, when this user purchases directly from the cloud. In each group, we do the same count for the aggregate demand and compare it with the sum of the wasted instance-hours of all users in that group. Fig. 10 plots the results. As expected, we observe a reduction of wasted instance-hours in all four cases. Interestingly, the waste reduction is the most significant for users with medium fluctuation, instead of highly fluctuated users. This is because we do not have a sufficient amount of highly fluctuated demands to aggregate.

B. The Ineffectiveness of Conventional ADP

Before evaluating cost savings of the broker under different reservation strategies, we first show the ineffectiveness of conventional ADP algorithms to handle our problem. We use two methods to speed up the convergence of ADP. *First,* following (9), we use the Heuristic strategy (Algorithm 1) as a good initial estimate. *Second,* we adopt *coarse-grained* reservations. That is, every time any reservation is made, we only reserve a number of instances that is a multiple of a certain integer g, defined as *the reservation granularity*. Although such a coarse-grained reservation strategy leads to a sub-optimal solution when $g > 1$, it can accelerate the convergence, as the strategy space is exponentially reduced.

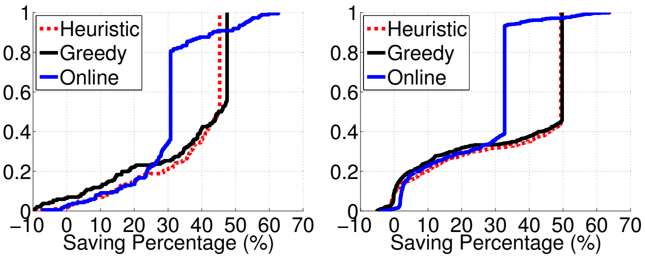
[1] All the algorithms are run on a machine with 1.7GHz Intel Core i5 and 4GB RAM.



(a) Impact of initial estimates.

(b) Reservation granularities.

Fig. 11. The convergence of ADP in different scenarios.



(a) Group 2: medium fluctuation

(b) All the users

Fig. 14. CDF of price discounts for individual users due to the brokerage service, under different algorithms.

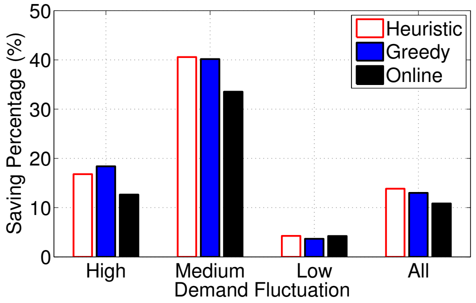


Fig. 12. Aggregate cost savings in different user groups due to the brokerage service.

The choice of granularity strikes a tradeoff between optimality and convergence speed.

However, even with the above acceleration, the convergence remains intolerably slow. As shown in Fig. 11a, though a good initial estimate reduces the convergence iterations by an order compared with naively setting the initial estimate to 0, convergence still takes over 90K iterations even for an extremely coarse-grained reservation with $g = 5000$. As shown in Fig. 11b, for more fine-grained reservations (g=1000, 2000, or 3000), the ADP shows no sign of convergence even after 200K iterations, where the achieved aggregate cost remains higher than a more coarse-grained strategy with $g = 5000$. Table I further compares ADP with the proposed Heuristic and Greedy strategies. We see that conventional ADP is inefficient in terms of both cost savings and run time for the scale of our problem. Therefore, we will focus on evaluating the proposed approximate algorithms.

C. Aggregate Cost Savings

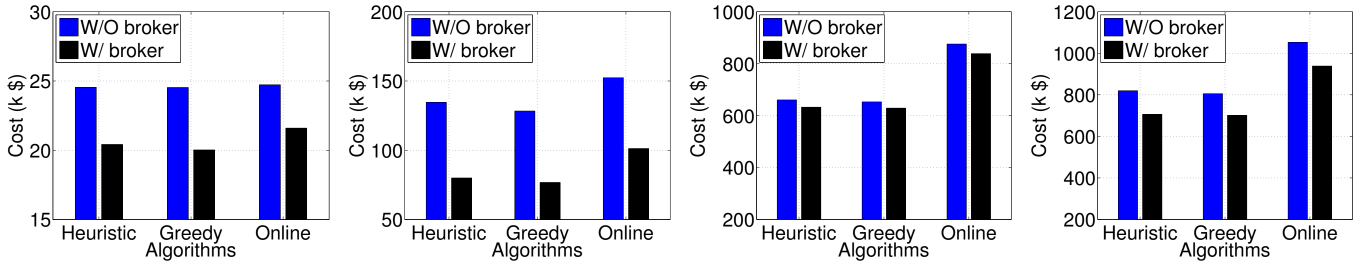
We now evaluate the aggregate cost savings offered by the broker, under three different reservation strategies, namely, the Heuristic (Algorithm 1), Greedy (Algorithm 2) and Online. Assuming a specific strategy is used, we compare the total service cost if users are using the broker with the sum of costs if each user individually makes reservations without using the broker. Fig. 13 shows such comparisons in each user group, while Fig. 12 shows the percentage of cost savings due to the use of a broker.

From Fig. 12, we see that the broker can bring a cost saving of close to 15% when it aggregates all the user demands. In terms of absolute values, the saving is over \$100K, as shown in Fig. 13d. However, the broker's benefit is different in different user groups: cost saving is the highest for users with medium demand fluctuation (40%), and the lowest for users with low demand fluctuation (5%). This is because when user demands are steady, they are heavily relying on reserved instances, regardless of whether they use the brokerage service or not. The broker thus brings little benefit, as shown in Fig. 13c. In contrast, for fluctuated demands, as shown in Fig. 13b, the broker can smooth out the demand curve through aggregation, better exploiting discounts of reserved instances. However, when users are *highly* fluctuated with bursty demands, as shown in Fig. 13a, even the aggregate demand curve is not smooth enough: these users can only leverage a limited amount of reserved instances, leading to less reservation benefit than for users with medium fluctuation. However, there is still $15 - 20\%$ cost saving, partly due to aggregation and the reduction of partial usage.

We now compare the costs of different reservation strategies. Fig. 13 verifies that Greedy is the best strategy while Online is inferior due to the lack of future knowledge. However, as shown in Fig. 13a, the three strategies are similar for highly fluctuated users, because for bursty demands, on-demand instances are mainly used (especially without broker) and reservation strategies become less critical.

D. Individual Cost Savings

We next evaluate the price discount each individual user can enjoy from the brokerage service. We consider a straightforward usage-based pricing scheme adopted by the broker. That is, for each user, the broker calculates the area under its demand curve to find out the instance-hours it has used. The broker then lets users share the aggregate cost in proportion to their instance-hours. In Fig. 14, we plot the CDF of price discounts of individual users due to using the broker. In Fig. 15, we plot the costs with and without the broker for each individual user (represented by a circle), under Greedy strategy, where such costs are the same if the circle is on the straight line $y = x$. We do not plot for Group 3 (low fluctuation) because the benefit of broker is less significant. In this sense, users in Group 3 has less motivation to use the broker. Furthermore, we do not plot for Group 1 (high fluctuation) because all their cost saving percentages are observed



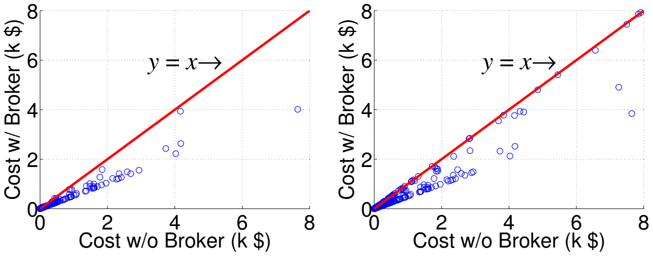
(a) Group 1: high fluctuation

(b) Group 2: medium fluctuation

(c) Group 3: low fluctuation

(d) All the users

Fig. 13. Aggregate service costs with and without broker in different user groups.



(a) Group 2: medium fluctuation

(b) All the users

Fig. 15. Cost without the broker vs. with the broker for individual users, using Greedy strategy. Each circle is a user.

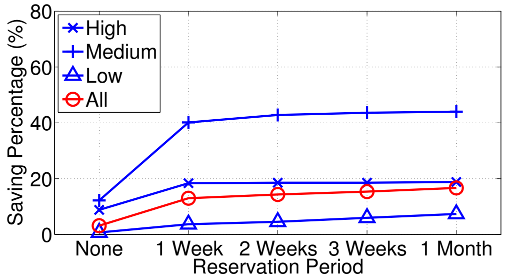


Fig. 16. The aggregate cost saving as the reservation period varies in different user groups, using Greedy strategy, where "None" means no reservation is available.

to be the same as the aggregate saving percentage. The reason is that with highly bursty demands, users in Group 1 will mainly use on-demand instances without the broker, leading to bills proportional to their usage. If these users choose to use the broker, their costs are also proportional to their usage. Therefore, the individual saving percentages are essentially the same as the aggregate saving percentage.

From Fig. 14a, we see that over 70% of users in Group 2 save more than 30%, while in Fig. 14b, we see that the broker can bring more than 25% price discounts to 70% of users if all users are aggregated. Several interesting phenomena are noted from Fig. 14 and Fig. 15. *First*, there is an upper limit on the price discount a user can get under Greedy, which is about 50%. *Second*, with Online, a majority (around $40-50\%$) of users receive a discount of around 30%. *Third*, when the broker charges users based on usage, only very few users (less than 5%) do not receive discounts (with price discount below 0 or circles above the straight line in Fig. 15). Since these users only contribute to a very small portion of the entire demand (around 3%), the broker can easily guarantee to charge them at most the same price as charged by cloud providers, by compensating them with a portion of the profit gained from service cost savings.

It is worth noting that the above usage-based billing is only one of many possible pricing policies that the broker can use. We adopt it here because it is easy to implement and understand. Although it may cause the problem of compensating overcharged users as mentioned above, it is not typically an issue in our simulations. We note that more complicated pricing polices, such as charging based on users' Shapley value [11], can resolve this problem with guaranteed discounts for everyone. The discussion of these policies is out of the scope of this paper. As long as the cost saving is achieved by the broker, there are rich methods to effectively share the benefits among all participants (see Ch. 15 in [12]).

E. Reservation Periods and Billing Cycles

We now quantify the impact of other factors on the performance of the broker. The first factor we consider is the length of the reservation period τ. In practice, different reservation periods are defined in different IaaS clouds, ranging from a month to years. To see how this affects the cost saving benefits, we fix the hourly on-demand rate, and try different reservation periods with 50% full-usage discount (i.e., the reservation fee is equal to running on-demand instances for half of the reservation period). The results are plotted in Fig. 16. We observe that, in general, the longer the reservation period, the more significant the cost saving achieved by the broker. It is worth noticing that the broker offers very limited cost savings when there is no reserved instance offered in the IaaS cloud. In this case, the cost saving is only due to the reduction of partial usage.

The second factor that we take into account is how the length of billing cycle affects the cost saving. To see this, we change the billing cycle from an hour to a day, which is the case in VPS.NET [8]. We set the daily on-demand rate to 24 times the original hourly rate (i.e., $24 \times \$0.08 = \$1.92$). The full-usage reservation discount remains 50% (VPS.NET offers 41% full-usage reservation discount, though). Fig. 17a and Fig. 17b present the simulation results using the Greedy strategy. As compared to the case of hourly billing cycle

(Fig. 12 and 14b), we observe a significant cost saving improvement here. Intuitively, adopting a larger billing cycle results in more wasted partial usage, leading to more salient advantages of using the broker.

F. Discussions and Other Practical Issues

Let us further discuss several practical issues. First, the savings from partial usage reduction are conditioned on the pricing details of a specific cloud. It is worth noting that time-multiplexing users on an on-demand instance in EC2 will not save the cost. This is because in EC2, stopping a user on an on-demand instance terminates a billing cycle, while loading a new user onto it opens a new one [2]. As a result, in Fig. 2, time-multiplexing (lower figure) will be billed for 3 instance-hours due to 2 user switches. However, this is generally not an issue for other cloud providers such as ElasticHosts [4] or reserved instances with a fixed cost (e.g., EC2 Heavy Utilization Reserved Instances). Furthermore, since the saving from partial usage reduction does not contribute much to the overall saving, as can be verified from Fig. 16 (none-reservation shows the saving from time-multiplexing alone), the total cost gain will only be degraded slightly (less than 10% in most cases) even without time-multiplexing.

Second, by taking advantage of *volume discounts*, the cost of instance reservations would further be reduced significantly. As mentioned in Sec. II, in practice, most IaaS clouds offer heavy volume discounts to large users. Some clouds even provide bargaining options for large users to enjoy further discounts. For example, in Amazon EC2, such volume discounts offer an additional 20% off on instance reservations [2]. Due to the sheer volume of the aggregated demand, the broker can easily qualify for these discounts.

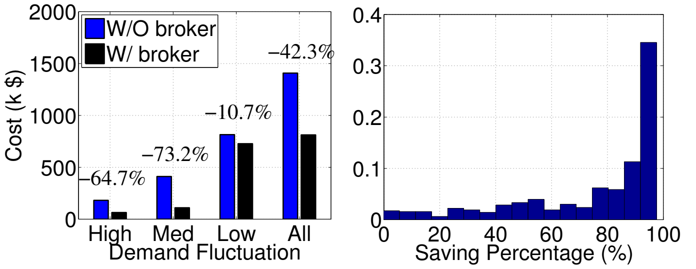
Third, in reality a user may only have rough knowledge of its future computing demands, so the broker's demand estimate may not be accurate. However, the users face exactly the same situation when purchasing directly from the cloud. In this case, they can still benefit from a broker that uses the Online strategy, which does not rely on future information.

Furthermore, in our simulation, we consider the case that the broker rewards all cost savings to users as price discounts. In reality, the broker can turn a profit by taking a portion of the savings as profit or through a commission.

Finally, in addition to savings on the expenses of running instances, the broker can also help lower the costs of other cloud resources such as storage, data transfer, and bandwidth. Since all these prices are *sub-additive* [2], the cost of provisioning aggregated resources is much cheaper than the total cost of purchasing them individually from the cloud.

VII. Related Work

Three types of pricing options are currently adopted in IaaS clouds. Besides the on-demand and reserved instances introduced in Sec. II, we note that some cloud providers charge dynamic prices that fluctuate over time, e.g., the Spot Instances in Amazon EC2 [2]. Some existing works discuss how to leverage these pricing options to reduce instance running costs for an individual user. For example, Zhao et al. [13] propose resource rental planning with EC2 spot price predictions to reduce the operational cost of cloud applications. Hong et al. [14] design an instance purchasing strategy to reduce the "margin cost" of over-provisioning. [14] also presents a strategy to combine the use of on-demand and reserved instances, which is essentially a special case of our Heuristic strategy when all demands are given in one reservation period. Vermeersch [15] implements a prototype software that dynamically retrieves instances from Amazon EC2 based on the user workload. All these works offer a *consulting service* that helps an individual user make instance purchasing decisions.



(a) Aggregate cost savings

(b) Histogram of individual cost savings (aggregating all users)

Fig. 17. Cost savings with a daily billing cycle under the Greedy strategy.

IaaS cloud brokers have recently emerged as *intermediators* connecting buyers and sellers of computing resources. For example, SpotCloud [16] offers a "clearinghouse" in which companies can buy and sell unused cloud computing capacity. Buyya et al. [17] discuss the engineering aspects of using brokerage to interconnect clouds into a global cloud market. Song et al. [18], on the other hand, propose a broker that predicts EC2 spot price, bids for spot instances, and uses them to serve cloud users. Unlike existing brokerage services that accommodate individual user requests separately, our broker serves the aggregated demands by leveraging instance multiplexing gains and instance reservation, and is a general framework not limited to a specific cloud.

We note that the idea of resource multiplexing has also been extensively studied, though none of them relates to computing instance provisioning. For example, [19] makes use of bandwidth burstable billing and proposes a cooperative framework in which multiple ISPs jointly purchase IP transit in bulk to reduce individual costs. In [20], the anti-correlation between the demands of different cloud tenants is exploited to save bandwidth reservation cost in the cloud. [21] empirically evaluates the idea of statistical multiplexing and resource over-booking in a shared hosting platform. Compared with these applications, exploiting multiplexing gains in cloud instance provisioning poses new challenges, mainly due to the newly emerged complex cloud pricing options. It remains nontrivial to design instance purchasing strategies that can optimally combine different pricing options to reduce cloud usage cost.

Finally, we refer readers to [10] for an extensive survey on dynamic programming and ADP algorithms.

VIII. Concluding Remarks

In this paper, we propose a smart cloud brokerage service that serves cloud user demands with a large pool of computing instances that are either dynamically reserved or launched on demand from IaaS clouds. By taking advantage of instance multiplexing gains as well as the price gap between on-demand and reserved instances, the broker benefits cloud users with heavy discounts while gaining profits from the achieved cost savings. To optimally exploit the price benefits of reserved instances, we propose a set of dynamic strategies to decide when and how many instances to reserve, with provable performance guarantees. Large-scale simulations driven by real-world cloud usage traces quantitively suggest that significant cost savings can be expected from using the proposed cloud brokerage service.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, 2010.

[2] Amazon EC2 Pricing, http://aws.amazon.com/ec2/pricing/.

[3] BitRefinery, http://bitrefinery.com.

[4] ElasticHosts, http://www.elastichosts.com/.

[5] GoGrid Cloud Hosting, http://www.gogrid.com.

[6] Ninefold, http://www.ninefold.com.

[7] OpSource, http://www.opsource.net.

[8] VPS.NET, http://vps.net.

[9] Google Cluster-Usage Traces, http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[10] W. Powell, *Approximate Dynamic Programming: Solving the curses of dimensionality*. John Wiley and Sons, 2011.

[11] A. E. Roth, Ed., *The Shapley Value, Essays in Honor of Lloyd S. Shapley*. Cambridge University Press, 1988.

[12] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, *Algorithmic Game Theory*. Cambridge University Press, 2007.

[13] H. Zhao, M. Pan, X. Liu, X. Li, and Y. Fang, "Optimal resource rental planning for elastic applications in cloud market," in *Proc. IEEE IPDPS*, 2012.

[14] Y. Hong, M. Thottethodi, and J. Xue, "Dynamic server provisioning to minimize cost in an IaaS cloud," in *Proc. ACM SIGMETRICS*, 2011.

[15] K. Vermeersch, "A broker for cost-efficient qos aware resource allocation in EC2," Master's thesis, University of Antwerp, 2011.

[16] SpotCloud, http://spotcloud.com/.

[17] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–619, 2009.

[18] Y. Song, M. Zafer, and K.-W. Lee, "Optimal bidding in spot instance market," in *Proc. IEEE INFOCOM*, 2012.

[19] R. Stanojevic, I. Castro, and S. Gorinsky, "CIPT: Using tuangou to reduce IP transit costs," in *Proc. ACM CoNEXT*, 2011.

[20] D. Niu, H. Xu, and B. Li, "Quality-assured cloud bandwidth auto-scaling for video-on-demand applications," in *Proc. IEEE INFOCOM*, 2012.

[21] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," in *Proc. USENIX Symposium on OSDI*, 2002.