Dynamic Cloud Instance Acquisition via IaaS Cloud Brokerage

Wei Wang, *Student Member, IEEE,* Di Niu, *Member, IEEE,* Ben Liang, *Senior Member, IEEE,* and Baochun Li, *Senior Member, IEEE*

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 approximation 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.

Index Terms—Cloud computing, cloud brokerage, cost management, instance reservation, approximation algorithm.

1 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 [2]. 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 [3], [4], [5], [6], [7], [8]. 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 [3], [5], [6]). 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 [3], [4], [5], [6], [7], [8]. Such billing inefficiency becomes more salient for cloud providers adopting longer billing cycles (e.g., in VPS.NET [9], 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 the *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 of 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

• *W. Wang, B. Liang and B. Li are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada. E-mail: weiwang@ece.toronto.edu, liang@comm.utoronto.ca, bli@ece.toronto.edu*
D. Niu is with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada. E-mail: dniu@ualberta.ca
• *Part of this paper has appeared in [1]. This new version contains substantial revision with new algorithm designs, analysis, proofs, and simulation results.*

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 a profit 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. As an initial attempt to overcome 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. Therefore, we propose two efficient approximation algorithms that incur at most twice the minimum cost. We also propose an effective online algorithm that makes reservation decisions dynamically without having access to future demand information. Theoretical analysis shows that the proposed online algorithm is 4-competitive.

We conduct large-scale simulations driven by 180 GB of Google cluster usage traces [10] 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. 2 and formulate the dynamic resource reservation problem in Sec. 3. We use dynamic programming to characterize the optimal solutions in Sec. 4 and point out the related complexity issues. In Sec. 5, we propose efficient approximation solutions to the reservation problem. The empirical evaluations based on real-world traces are presented in Sec. 6. We discuss other practical issues and future works in Sec. 7. We then survey the related work in Sec. 8 and conclude the paper in Sec. 9.

2 A PROFITABLE CLOUD BROKER

Most IaaS clouds provide users with multiple purchasing options, including on-demand instances, reserved instances, and other instance types [3], [4], [5], [6], [7], [8]. *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 np. 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 [4], [5], [6], [7], [8], [9], the cost of a reserved instance is

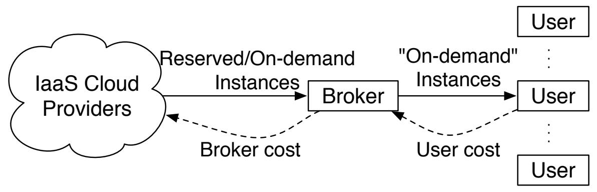


Fig. 1. The proposed cloud broker. Solid arrows show the direction of instance provisioning; dashed arrows show the direction of money flow.

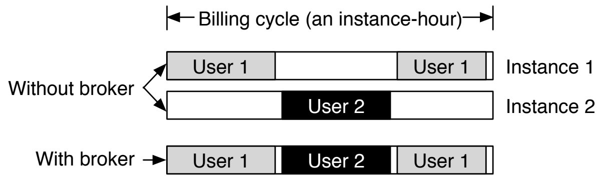


Fig. 2. The broker can time-multiplex partial usage from different users in the same instance-hour. In this case, serving two users only takes one instance-hour, instead of two.

equal to the reservation fee. As another example, in Amazon EC2 [3], the cost of a Heavy Utilization Reserved Instance is a reservation fee plus a heavily discounted hourly rate charged over the entire reservation period, irrespective of the actual instance usage. EC2 also offers other reservation options (e.g., Light/Medium Utilization Reserved Instances), with cost linearly 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 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:

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 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. 7.

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 [3]. 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.

A brokerage service is profitable if it can achieve cost savings to serve the aggregate demands: the broker can always turn an agreed-upon portion of the savings to its own profit. We omit the discussion on the detailed pricing implementations, as it is irrelevant to the paper's focus. Instead, 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 main theme of the following sections.

3 Dynamic Instance Acquisition

In this section, we formulate the broker's optimal instance reservation problem to accommodate given demands, with an objective of minimizing instance acquisition cost. 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. 5.3 to make decisions based on history only.

Suppose cloud users submit to the broker their estimates of computing demand up to time T into the future. The broker aggregates 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 $r_t > 0$. 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

$$n_t = \sum_{i=t-\tau+1}^{t} r_i \ ,$$

where $r_i := 0$ for all $i \leq 0$. Note that these n_t reserved instances may not be sufficient to accommodate the aggregate demand d_t. Let

$$X^+ := \max\{0, X\} \ .$$

The broker thus needs to launch $(d_t - n_t)^+$ additional on-demand instances at time t.

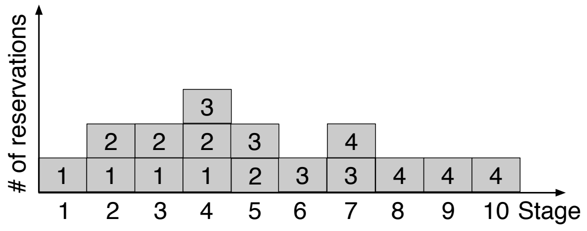


Fig. 3. State illustration. The reservation period is $\tau = 4$. All four reservations made at time 1, 2, 4, and 7 are highlighted as the shaded area. It is easy to verify that $\mathbf{s}_1 = (0,0,0), \mathbf{s}_2 = (1,0,0), \mathbf{s}_3 = (1,1,0), \mathbf{s}_4 = (0,1,1), \mathbf{s}_5 = (1,0,1)$, etc.

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

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

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.,

$$\min_{r_t \in \mathbb{Z}^+} \quad \sum_{t=1}^{T} r_t \gamma + \sum_{t=1}^{T} (d_t - n_t)^+ p \ , \tag{2}$$

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

4 Dynamic Programming: 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.

4.1 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 $(\tau - 1)$-tuple

$$\mathbf{s}_t := (r_{t-1}, r_{t-2}, \dots, r_{t-\tau+1}) \ , \tag{3}$$

i.e., $\mathbf{s}_t$ represents the instance reservation decisions made in the recent $\tau - 1$ stages. Here, we use a $(\tau-1)$-tuple to define a state because instances reserved earlier than stage $t - \tau + 1$ all expire at stage t and will have no effect at this stage. For example, in Fig. 3, four instances are reserved at time 1, 2, 4, and 7, respectively, and the reservation period is $\tau = 4$. We have $r_1 = r_2 = r_4 = r_7 = 1$, while $r_t = 0$ for all other stages. It is easy to verify that $\mathbf{s}_1 = (0,0,0), \mathbf{s}_2 = (1,0,0), \mathbf{s}_3 = (1,1,0), \mathbf{s}_4 = (0,1,1), \mathbf{s}_5 = (1,0,1)$, etc.

With the state definition (3), it is easy to characterize the following state transition equation and the corresponding cost function. In particular, suppose state $\mathbf{s}_t = (r_{t-1}, \ldots, r_{t-\tau+1})$ is reached at stage t. Also, suppose the broker decides to reserve r_t instances at the same stage. Such a reservation decision leads state $\mathbf{s}_t$ to transit to its next state $\mathbf{s}_{t+1} = (r_t, \ldots, r_{t-\tau+2})$, i.e.,

$$s_t \xrightarrow{r_t} s_{t+1} : (r_{t-1}, \ldots, r_{t-\tau+1}) \xrightarrow{r_t} (r_t, \ldots, r_{t-\tau+2}) \ . \quad (4)$$

The corresponding state transition cost is

$$c(\mathbf{s}_t, \mathbf{s}_{t+1}) = \gamma r_t + p(d_t - \sum_{i=t-\tau+1}^{t} r_i)^+. \quad (5)$$

The transition cost is composed of two terms, the reservation cost γr_t due to r_t newly reserved instances and the (potential) on-demand cost incurred when demand d_t cannot be accommodated by $\sum_{i=t-\tau+1}^{t} r_i$ reserved instances currently available.

Now let $V(\mathbf{s}_t)$ be the minimum cost of serving demands $d_1, \ldots, d_t$ up to stage t, conditioned on that state $\mathbf{s}_t$ is reached at stage t. We have the following recursive Bellman equations:

$$V(\mathbf{s}_{t+1}) = \min_{\mathbf{s}_t} \{V(\mathbf{s}_t) + c(\mathbf{s}_t, \mathbf{s}_{t+1})\}, \quad t = 1, 2, \ldots, \quad (6)$$

where the minimization is taken over all states $\mathbf{s}_t$ that can transit to state $\mathbf{s}_{t+1}$. The Bellman equation (6) essentially indicates that the minimum cost of reaching state $\mathbf{s}_{t+1}$ is given by the minimum cost of reaching a previous state $\mathbf{s}_t$ plus a transition cost $c(\mathbf{s}_t, \mathbf{s}_{t+1})$, minimized over all possible $\mathbf{s}_t$.

The boundary conditions of (6) are given by

$$V(\mathbf{s}_1) = 0 \ , \quad (7)$$

since the initial state $\mathbf{s}_1 = (r_0, \ldots, r_{2-\tau}) = \mathbf{0}$ by definition.

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

Proposition 1: The dynamic programming defined by (4), (5), (6), and (7) 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. 4, a state $\mathbf{s}_t$ is represented by a node at stage t. If state $\mathbf{s}_t$ can transit to state $\mathbf{s}_{t+1}$, i.e., they satisfy the state transition equations (4), then node $\mathbf{s}_t$ is connected to node $\mathbf{s}_{t+1}$ by an edge with length $c(\mathbf{s}_t, \mathbf{s}_{t+1})$. In this sense, $V(\mathbf{s}_t)$ is the length of a shortest path from node $\mathbf{s}_1$ to $\mathbf{s}_t$.

4.2 The Curse of Dimensionality

Dynamic programming is the best algorithm that we are aware of to solve (2). Although it gives the optimal instance acquisition cost, it is computationally prohibitive for large data. This is because to derive the minimum cost, 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 $\tau - 1$ tuple $(r_{t-1}, \ldots, r_{t-\tau+1})$, there exist $O(\bar{d}^{\tau-1})$ such nodes in the trellis graph, where $\bar{d} = \max_t d_t$ is the peak demand. Therefore, going through all states results in

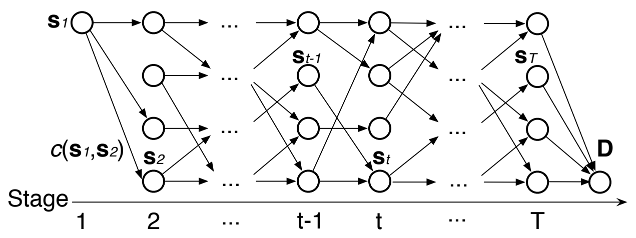


Fig. 4. Dynamic programming as a shortest path problem. The minimum cost is the output $V(\mathbf{s}_T)$.

exponential time complexity. Also, since the computed $V(\mathbf{s}_t)$ has to be stored for every node $\mathbf{s}_t$ at a stage, the space complexity is exponential as well. This is known as *the curse of dimensionality* suffered by all high-dimensional dynamic programming [11].

A classical method to handle the curse of dimensionality is to use *Approximate Dynamic Programming* (ADP) [11]. 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. 4, ADP picks a shortest path $P^k = \{\mathbf{s}_T^{(k)}, \ldots, \mathbf{s}_1^{(k)}\}$ from stage T to 1, 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

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

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

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

Then we move to the next node $\mathbf{s}_{t-2}^{(k)}$ until stage 1 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 $V(\mathbf{s}_t)$ [11]. 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 an intelligent way to set $\tilde{V}^{(0)}(\mathbf{s}_t)$ in Sec. 5.1, leveraging the approximation algorithms proposed there. However, through extensive simulations, we will show in Sec. 6.2 that although intelligent initial estimates significantly accelerate ADP, as an iterative method, its convergence speed is still unsatisfactory to handle the large amount of demand data in our problem.

5 Approximation Algorithms

To overcome the prohibitive complexity of dynamic programming, in this section, we develop approximation algorithms to

Algorithm 1 Heuristic: Periodic Decisions

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

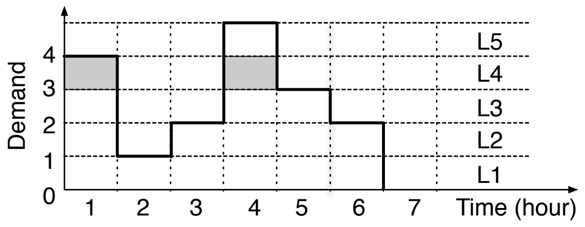


Fig. 5. The Periodic Decisions algorithm, with $\gamma = 2.5$, $p = 1$, $\tau = 6$, and $T = 6$.

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 unavailable.

5.1 A 2-Approximation Heuristic

We first present a simple heuristic that in the worst case, incurs twice the minimum cost. This heuristic serves as a basis to analyze algorithms proposed later in Sec. 5.2 and 5.3. We start off by dividing the demands into $\bar{d}$ *levels*, where $\bar{d}$ is the peak demand, i.e.,

$$\bar{d} := \max_t d_t \ . \tag{8}$$

For example, in Fig. 5, the total demands are divided into $\bar{d} = 5$ levels, with level 1 being the bottom (labeled as "L1" in Fig. 5) and level 5 being the top. Define d_t^l as the demand at time t in level l, such that $d_t^l = 1$ if $d_t \geq l$, and $d_t^l = 0$ otherwise. For example, in Fig. 5, level 4 has demands only at time 1 and 4 (i.e., $d_1^3 = d_4^3 = 1$).

We now consider a special case, when all given demands are within a single reservation period, i.e., $T \leq \tau$. 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 would be well justified if the reservation fee satisfies $\gamma \leq pu_1$; 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.,

$$u_l := \sum_{t=1}^T d_t^l, \quad l > 0 \ . \tag{9}$$

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

Fig. 5 shows an example with $\gamma = 2.5$, $p = 1$, and $\tau = 6$. 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., $u_4 = 2$, as shown by the shaded area. In this case, the optimal strategy is to reserve two instances in the bottom three levels, as $u_3 = 3 > 2.5 = \gamma/p$ while $u_4 = 2 < \gamma/p$.

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 interval separately, only at the beginning of that interval, by running the above algorithm. This leads to the Periodic Decisions described by Algorithm 1. It is easy to check that Algorithm 1 only requires $O(\bar{d}T)$ time and $O(T)$ space.

The following proposition shows that Algorithm 1 is more than a simple heuristic.

Proposition 2: Algorithm 1 is *2-approximation*, incurring no more than twice the minimum cost.

The proof is deferred to the appendix[1]. Below we briefly explain its main idea. We say a reserved instance is *interval-aligned* if it is reserved at the beginning of an interval, i.e., its reservation period overlaps exactly one interval. Now given an arbitrary instance reservation algorithm, the following construction will lead to an outcome with all reserved instances interval-aligned. Whenever an instance is reserved yet is not interval-aligned, its reservation period must overlap two consecutive intervals. We replace this reserved instance with two reservations aligned with these two intervals, respectively. We can show that such an interval-aligned construction incurs *at most* twice the cost of the original algorithm. Also note that Algorithm 1 is optimal among all algorithms making interval-aligned reservations. It hence incurs less cost than the interval-aligned construction of any algorithm, which implies that Algorithm 1 incurs at most twice the cost of any algorithm.

We now show by the following example that the 2-approximation analysis is *tight* for Algorithm 1. Consider a pricing setup with $p = 1$ and $\gamma = \tau/2 + \varepsilon$, where the reservation period τ is even. Let the demand curve be $d_1 = d_{\tau/2+2} = \cdots = d_{3\tau/2} = 1$, while $d_t = 0$ for all other t. It is easy to verify that Algorithm 1 will launch all instances on demand, incurring cost τ. On the other hand, the optimal strategy reserves one instance at time $\tau/2 + 2$, with the total cost $\tau/2 + 1 + \varepsilon$. By taking $\tau \gg 1$ and $\varepsilon \to 0$, the

1. The appendix is given in a supplementary document as per the TPDS submission guidelines.

approximation ratio could be arbitrarily close to 2.

With the above performance guarantee, it is worth mentioning that Algorithm 1 can be used to compute the initial estimates for the aforementioned ADP algorithm and speed up its convergence. Specifically, let $\text{Cost}_{A1}(t)$ be the cost incurred by Algorithm 1 for demands $d_1, \ldots, d_t$ up to time t. For each state $\mathbf{s}_t = (r_{t-1}, \ldots, r_{t-\tau+1})$, we set its initial estimate to be

$$\tilde{V}^{(0)}(\mathbf{s}_t) := \max\left\{ \frac{\text{Cost}_{A1}(t-1)}{2}, \sum_{i=t-\tau+1}^{t-1} \gamma r_i \right\} . \quad (10)$$

We have

Proposition 3: The initial estimate (10) is *optimistic* for all state $\mathbf{s}_t$, i.e.,

$$\tilde{V}^{(0)}(\mathbf{s}_t) \le V(\mathbf{s}_t), \quad \forall \mathbf{s}_t. \quad (11)$$

Proof: Because Algorithm 1 incurs no more than twice the minimum cost, we have

$$\text{Cost}_{A1}(t-1) \le 2V(\mathbf{s}_t) . \quad (12)$$

On the other hand, by definition, at state $\mathbf{s}_t$, at least $\sum_{i=t-\tau+1}^{t-1} r_i$ instances have been reserved, which implies

$$\sum_{i=t-\tau+1}^{t-1} \gamma r_i \le V(\mathbf{s}_t) . \quad (13)$$

Combining (12) and (13), we see the statement holds. □

Since the initial estimate is optimisitc, the ADP will converge to the optimality. We will show experimentally in Sec. 6.2 that the initial estimate (10) significantly accelerates ADP convergence.

5.2 An Improved Greedy Algorithm

Algorithm 1 divides problem (2) into reservation subproblems, each solved in a separate level. However, in each level, the reservations are made only at the beginnings of intervals. In this subsection, we consider an improvement of Algorithm 1 that optimally reserves instances in each level.

In particular, the algorithm starts to make optimal reservations in the top level $\bar{d}$. Note that there might exist some instance reserved in level $\bar{d}$ but unused at some time due to the lack of demand. For better utilization, these reservation slots are passed over to the lower level $\bar{d}-1$, with the hope that they could be used by demands there, if any. The algorithm then steps down to the second top level $\bar{d}-1$, where it makes optimal reservations, with the potential use of "leftover" reservation slots carried over from the upper level. Unused reservation slots are then passed over to the lower level $\bar{d}-2$. The algorithm proceeds *top-down* and stops when reaching level 0.

For each level, the optimal reservation can be efficiently computed via dynamic programming. Suppose before processing level l, at time t, there are m_t^l unused reservation slots carried over from upper levels. Let $V_l(t)$ be the minimum cost of serving demands $d_1^l, \ldots, d_t^l$ in level l up to time t. $V_l(t)$ can be recursively computed by the following Bellman equation:

$$V_l(t) = \min\{V_l(t-\tau) + \gamma,\ V_l(t-1) + c_l(t)\} , \quad (14)$$

Algorithm 2 Greedy Reservation Strategy

```
1. Initialization: $m_t^{\bar{d}} \leftarrow 0$ for all $t = 1, \ldots, T$.
2. for $l = \bar{d}$ down to 1 do
3.     Make optimal reservations in level $l$ via dynamic programming defined by (14), (15), and (16).
4.     Update $m_t^{l-1}$ for all $t$.
5. end for
```

where

$$c_l(t) = \begin{cases} p, & \text{if } d_t^l = 1 \text{ and } m_t^l = 0 , \\ 0, & \text{otherwise,} \end{cases} \quad (15)$$

To see the rationale behind (14), we note that there are two alternatives to serve demand d_t^l. The *first* is to use a reserved instance made in the current level l. Because $V_l(\cdot)$ is increasing, the best strategy is to optimally serve demands up to time $t-\tau$ and reserve an instance at the next time slot $t-\tau+1$, incurring the total cost $V_l(t-\tau)+\gamma$. The *second* alternative is to serve demand d_t^l using an on-demand instance, if there is no unused reservation slot from upper levels at time t, i.e., $m_t^l = 0$. Otherwise, serve demand d_t^l free with unused reservation slot. The costs incurred under these two conditions are exactly given by (15). We finally give the boundary conditions as follows:

$$V_l(t) = 0, \quad t \le 0 . \quad (16)$$

After the optimal reservations have been computed in level l, we update m_t^{l-1}, the number of unused reservation slots at time t in level $l-1$, 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 an unused reservation slot carried over from upper levels;
- $m_t^{l-1} = m_t^l$, otherwise.

Algorithm 2 summarizes the aforementioned greedy reservation strategy. The time and space complexity is $O(\bar{d}T)$ and $O(T)$, respectively, as solving the dynamic programming in each level requires $O(T)$ time and $O(T)$ space. Algorithm 2 is a "level-by-level" improvement of Algorithm 1, which leads to the following proposition:

Proposition 4: Algorithm 2 is 2-approximation.

While it remains open to see if the 2-approximation is a tight analysis for Algorithm 2, we can show by the following example that the competitive ratio is at least 1.5. Consider a pricing setup with $p = 1$, $\gamma = 1 + \varepsilon$, and $\tau = 3$. Let the demand curve be $d_1 = d_2 = d_4 = 1$, $d_3 = 2$, while $d_t = 0$ for all other t. It is easy to check that Algorithm 2 reserves only one instance at time 1, with the total cost $3 + \varepsilon$. On the other hand, the optimal strategy reserves two instances at time 1 and time 3, respectively, with the total cost $2 + 2\varepsilon$. Taking $\varepsilon \to 0$ leads to the factor 3/2 approximation.

5.3 An Online Reservation Strategy

Previous algorithms apply to the case where users submit their future demand predictions. In the case when no future information is available, we propose a simple *online strategy* that makes reservation decisions based only on history. We first introduce the Bahncard algorithm [12] and then use it as a building block to make reservation decisions in each demand level.

The Bahncard problem models online ticket purchasing on German Federal Railway. A customer can buy a ticket every time she travels, or she can purchase a Bahncard and will be free of charge for all trips in the following year. Without knowing the future travel plans, the customer needs an online strategy to choose between these two pricing options to save her travel cost. The Bahncard problem is exactly the instance reservation problem limited to one demand level, where a Bahncard corresponds to a reserved instance and a ticket corresponds to an on-demand instance. We can hence use the Bahncard algorithm [12] to make reservation decisions in one level as follows. At time t, in level l, we keep track of the overall cost incurred by the use of on-demand instances in the past reservation period, i.e., from time $t-\tau+1$ to t. If this on-demand cost turns out to be no less than the reservation fee γ, then reserve an instance at time t. Otherwise, use an on-demand instance to serve the current demand d_t^l. At every time t, we apply this Bahncard algorithm separately in each level. Algorithm 3 formalizes the detailed process.

Since Algorithm 3 makes reservation decisions only based on history, it is an online strategy, without any future information. Even so, the following proposition shows that Algorithm 3 offers worst-case cost guarantee.

Proposition 5: Algorithm 3 is 4-competitive, incurring at most 4 times the minimum cost.

The proof is deferred to the appendix. The main idea is to use the fact that the Bahncard algorithm incurs at most twice the minimum cost in each demand level [12]. This is because the most inefficient reservation is to reserve an instance at time t but will never use it in the following time slot due to the lack of demand. This reservation is made because the on-demand cost incurred in the past reservation period reaches γ. It hence costs 2γ to serve demands from time $t-\tau+1$ to t. On the other hand, the optimal strategy reserves one instance to serve the same demands, with cost γ. Also, we can show that optimally reserving instances separately in each level incurs at most twice the minimum cost. Combined with the 2-competitiveness achieved by the Bahncard algorithm in each level, we see that Algorithm 3 is 4-competitive.

It is worth mentioning that the 4-competitiveness is a loose analysis. A tight competitive ratio remains open for Algorithm 3. We shall show by simulations in the next section that the performance of the online strategy is comparable with the two approximation algorithms when demand predictions are available. We have proposed a conceptually more complicated online strategy for instance reservation that yields the optimal competitive ratio in [13], which we shall also compare with in the next section.

6 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, under an extensive range of scenarios.

Algorithm 3 Online reservation strategy at time t, upon the arrival of demand d_t

```
1. Initialization: $r_t = 0$
2. for $l$ = 1 to $d_t$ do
3.     Let $a_l$ be the accumulated cost incurred by the use of
       on-demand instances in the past reservation period, i.e.,
       from time $t-\tau+1$ to time $t$.
4.     if $a_l \geq \gamma$ then
5.         Reserve an instance in level $l$ at the current time $t$,
           i.e., $r_t \leftarrow r_t + 1$.
6.     else
7.         Use an on-demand instance to serve the current
           demand in level $l$.
8.     end if
9. end for
```

6.1 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 [10], [14] that were recently released 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 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.

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[2], 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 onto different instances. (For simplicity, we ignore other complicated task placement constraints such as on OS versions and machine types.) A new instance will be launched if none of the available instances can accommodate a submitted task. Note that tasks of one user cannot be scheduled onto another user's instances. In the end, we obtain a demand curve for each user, indicating how many instances the user requires in each hour. Fig. 6 illustrates the demand curves of three typical users in the first 200 hours. For the broker, it simply adds up

2. Most Google cluster machines are of the same computing capability, with 93% having the same CPU cycles.

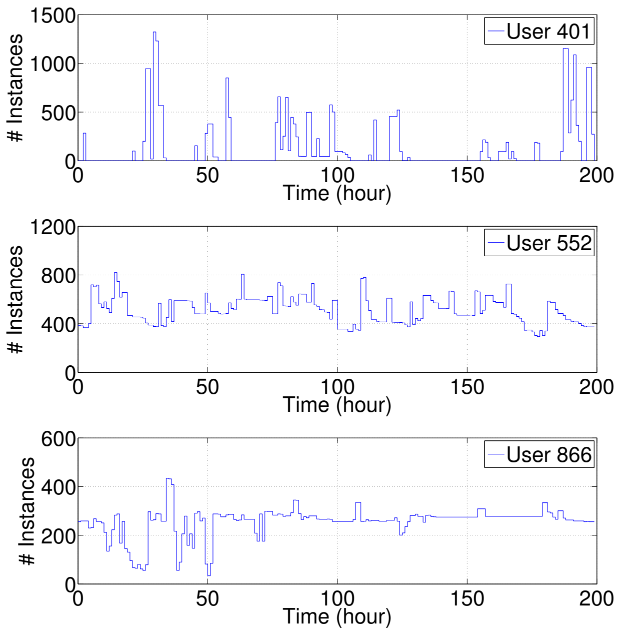


Fig. 6. The demand curves of three typical users.

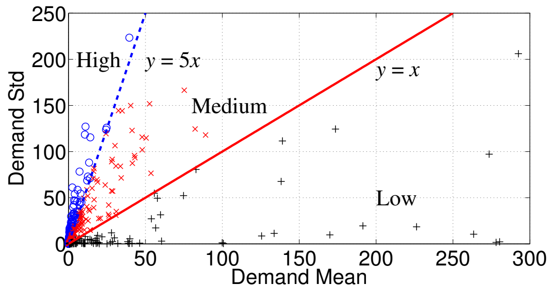


Fig. 7. Demand statistics and the division of users into 3 groups according to demand fluctuation level.

all users' demands for instances as the aggregate demand. This preserves the instance isolations among users as no user shares instances with one another.

Pricing: Unless explicitly mentioned, we set the on-demand hourly rate to \$0.044, the same as Amazon EC2 small instances[3]. Since the Google traces 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 [3], [5], [6], [9].

Group Division: To further understand the demand statistics of users, we compute the demand mean and standard deviation for each user and illustrate the results in Fig. 7. As has been mentioned, to what extent a user can benefit from reservations critically depends on its demand pattern: the more fluctuating the demand is, the less is the benefit from using reserved instances. We hence classify all 933 users into the following three groups based on 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. 6. There are 271 users in this group, represented by "o" in Fig. 7. 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. 6. There are 286 users in this group, represented by "x" in Fig. 7. These users require 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. 7. A typical user's demand is shown in the bottom graph of Fig. 6. Almost all high-demand users with the demand mean 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. 8 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. 8a and 8b that aggregating bursty users (i.e., users in Group 1 and 2) results in a steadier demand 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. 8c). In addition, Fig. 8d shows the result of aggregating all the users. In all cases, the aggregated demand is stabler and more suitable for service via reserved instances.

Another benefit of demand aggregation is to reduce the wasted instance-hours incurred by partial usage. To see this, for each user, we count the wasted instance-hours billed but not used to 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. 9 shows 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 fluctuating users. This is due to the relatively small number of users in Group 1 — we do not have a large amount of high-fluctuating demands to aggregate.

6.2 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. We use two methods to speed up the convergence of ADP. *First,* following (10), 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

3. The price is for Standard On-Demand Instances, Linux, US East, as of April 10, 2014.

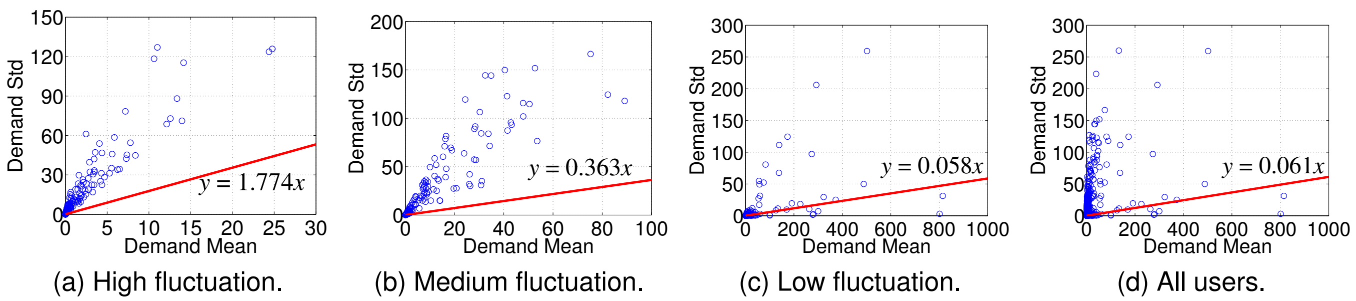


Fig. 8. Aggregation suppresses the demand fluctuation of individual users. Each circle represents a user. The line indicates the demand fluctuation level (the ratio between the demand standard deviation and mean) in the aggregate demand curve.

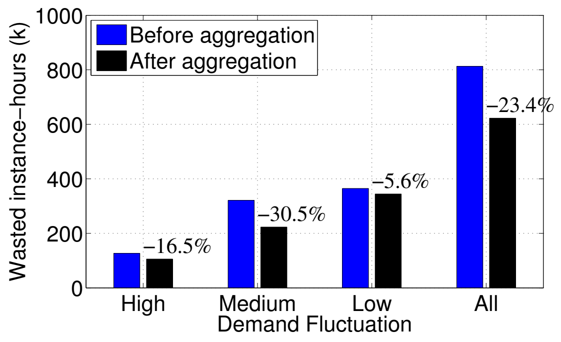


Fig. 9. Aggregation reduces the wasted instance-hours due to partial usage.

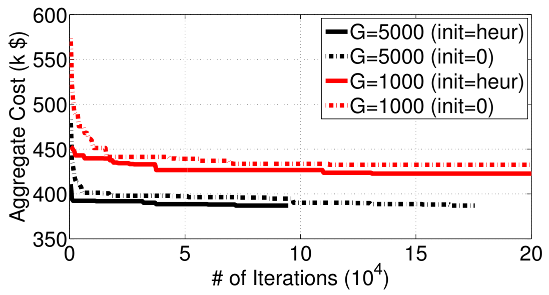


Fig. 10. The convergence speed of ADP accelerated by initial estimate and coarse-grained reservations.

TABLE 1
Comparisons in terms of cost and convergence.

Algorithm	Cost ($)	Converged	Run Time[4] (s)
ADP ($G = 8000$)	396,147	Yes	47
ADP ($G = 5000$)	390,344	Yes	65
ADP ($G = 3000$)	395,166	No	388
ADP ($G = 2000$)	399,019	No	1645
ADP ($G = 1000$)	422,680	No	2732
Heuristic ($G = 1$)	386,268	N/A	1
Greedy ($G = 1$)	385,552	N/A	6

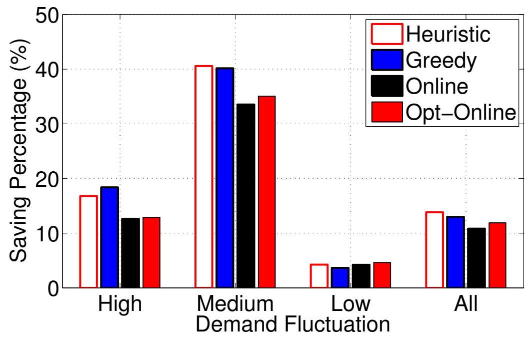


Fig. 11. Aggregate cost savings in different user groups due to the brokerage service.

reservation strategy leads to a sub-optimal solution when $G > 1$, it can accelerate the convergence, as the strategy space is exponentially reduced. 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. 10, although a good initial estimate reduces the convergence iterations by an order compared with naively setting the initial estimate to 0, it still takes over 90K iterations to converge even for an extremely coarse-grained reservation with $G = 5000$. As shown in Table 1, for more fine-grained reservations (G=1000, 2000, or 3000), ADP shows no sign of convergence even after 200K iterations, where the achieved aggregate cost remains higher than a more coarse-grained strategy with $G = 5000$. In fact, we find that $G = 5000$ is around the sweet spot that balances both the optimality and the convergence speed: setting a larger G, though converging faster, incurs higher cost due to the coarser reservation granularity. Table 1 further compares ADP with the proposed Heuristic and Greedy strategies. We see that the 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 approximation algorithms.

6.3 Aggregate Cost Savings

We now evaluate the aggregate cost savings offered by the broker under different reservation strategies. In particular, when demand predictions are reliable, both Heuristic (Algorithm 1) and Greedy (Algorithm 2) strategies can be applied. In this case, we simply set the demand information available to both

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

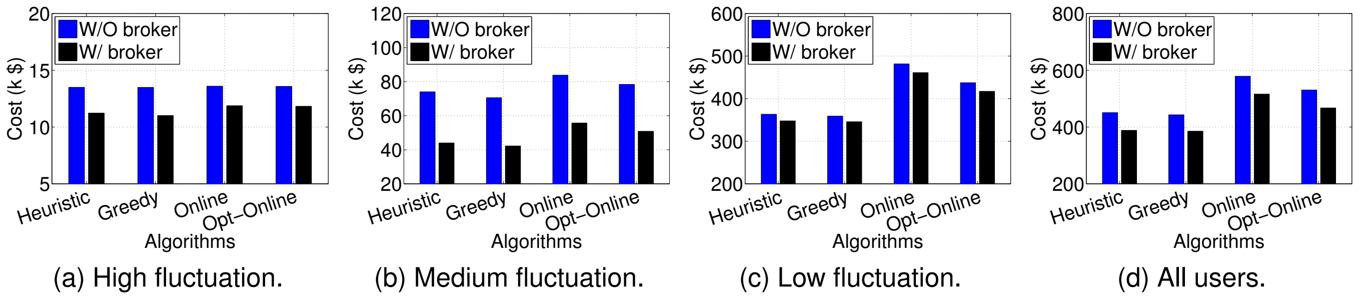
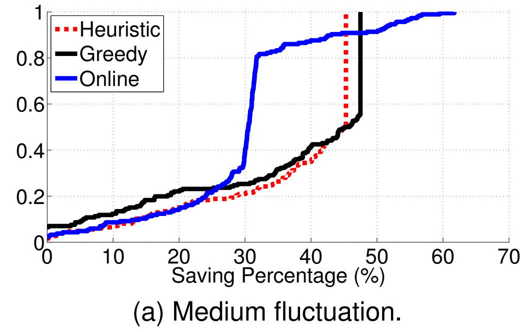


Fig. 12. Aggregate service costs with and without broker in different user groups.

algorithms. We take this simple approach because we have observed only a slight difference on the cost savings offered by the broker even when there are some prediction errors (up to 10%). When predictions are unavailable, we evaluate two online strategies, i.e., Algorithm 3 (Online) and a conceptually more complicated strategy we proposed in [13]. We refer to the latter algorithm as "Opt-Online" as it gives the optimal competitive ratio [13]. In either case, 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. 12 shows such comparisons in each user group, while Fig. 11 shows the percentage of cost savings due to the use of a broker.

From Fig. 11, 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 nearly \$100K, as shown in Fig. 12d. 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. 12c. In contrast, for fluctuating demands, as shown in Fig. 12b, the broker can smooth out the demand curve through aggregation, better exploiting discounts of reserved instances. However, when users are *highly* fluctuating with bursty demands, as shown in Fig. 12a, 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 $15\% \sim 20\%$ cost saving, partly due to aggregation and the reduction of partial usage.

We now compare the cost performance of different reservation strategies. We see from Fig. 12 that both Heuristic and Greedy algorithms outperform the two online strategies, due to the availability of demand prediction. On the other hand, despite the lack of future knowledge, the costs incurred by the two online algorithms are very close to those of Heuristic and Greedy when demands are fluctuating, as shown in Figs. 12a and 12b. However, for users with stable demand curves, the cost difference between the online algorithms and Heuristic and Greedy is more prominent. Fortunately, when users have stable demands, it would be easy to accurately predict their future demands, so that the online strategies will not be needed anyway. We hence view the online algorithms and Heuristic



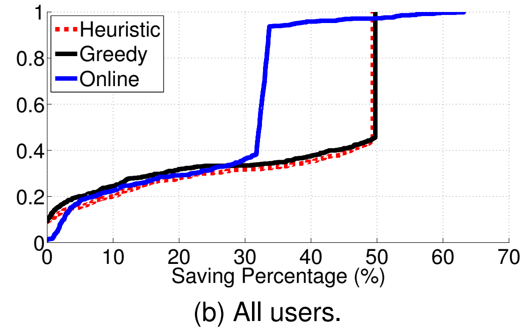
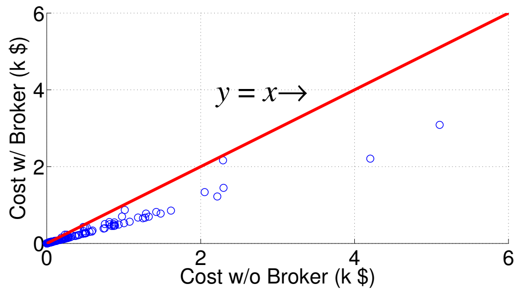


Fig. 13. CDF of price discounts for individual users due to the brokerage service, under different algorithms.

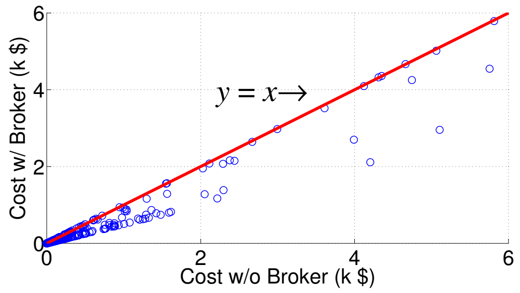
(Greedy) as *complementary* approaches applied in different scenarios. Also, Fig. 12 shows that the cost performance of the simple Online strategy can be further improved by Opt-Online, yet at a cost of more complicated design, implementation and analysis [13]. In terms of the relative cost savings offered by the broker, both online algorithms achieve similar performance gains, as shown in Fig. 11, although Opt-Online is a little better. However, due to its conceptual simplicity and ease of understanding, the Online strategy has its own merit that may appeal to fast adoption by brokerage service operators who prefer a lightweight implementation in reality. Therefore, we focus only on the Online strategy in the following evaluations.

6.4 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



(a) Medium fluctuation.



(b) All users.

Fig. 14. Cost without the broker vs. with the broker for individual users, using Greedy strategy. Each circle is a user.

to their instance-hours. In Fig. 13, we plot the CDF of price discounts of individual users due to using the broker. In Fig. 14, 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 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. 13a, we see that over 70% of users in Group 2 save more than 30%, while in Fig. 13b, we see that the broker can bring more than 25% price discounts to 70% of users if all users are aggregated. Several interesting observations are noted from Fig. 13 and Fig. 14. To begin with, there is an upper limit on the price discount a user can get under Greedy, which is about 50%. Moreover, with Online, a majority (around $40 - 50\%$) of users receive a discount of around 30%. Furthermore, 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. 14). 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.

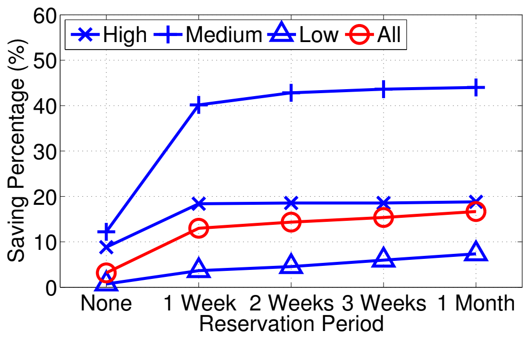
It is worth noting that the above usage-based billing is only one of many possible pricing policies that the broker can use. We adopt it here because it is easy to implement and understand. Although it may cause the problem of compensating overcharged users as mentioned above, it is not typically an issue in our simulations. We note that more complicated pricing polices, such as charging based on users' Shapley value [15], can resolve this problem with guaranteed discounts for every user. The discussion of these policies is orthogonal to 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 [16]).

6.5 Reservation Period, Discount, and Billing Cycle

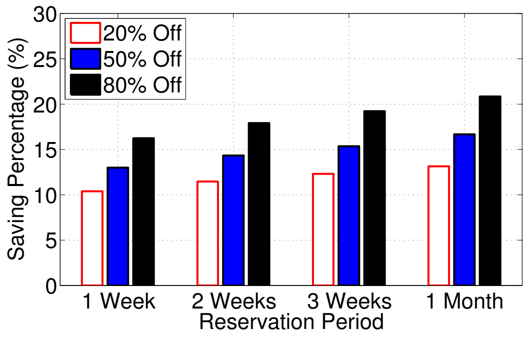
We now quantify the impact of other factors on the performance of the broker. The first factor we consider is the length of the reservation period. In practice, different reservation periods are adopted in different IaaS clouds, ranging from a month to years. To see how this affects the cost saving benefits, 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 given in Fig. 15a. 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.

Besides reservation period, another important parameter is the reservation discount offered by a reserved instance. Usually, the longer the reservation period, the heavier the reservation discount. To quantify how both parameters may affect the cost benefit of the brokerage service, we combine different reservation periods, varied from 1 to 4 weeks, with different reservation discounts, varied from 20%, 50%, to 80%. Fig. 15b presents the cost savings offered by the brokerage service under all 12 combinations, using the Greedy strategy. We observe a general trend that the heavier the reservation discount, the more cost savings could be achieved. This is because reserved instances can be more efficiently utilized via the brokerage service, leading to more cost benefits under a heavier discount.

The third 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 [9]. We set the daily on-demand rate to 24 times the original hourly rate (i.e., $24 \times \$0.044 = \$1.056$). The full-usage reservation discount remains 50% (VPS.NET offers 41% full-usage reservation discount, though). Fig. 16a and Fig. 16b present the simulation results using the Greedy strategy. As compared to the case of hourly billing cycle (Fig. 11 and 13b), 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.

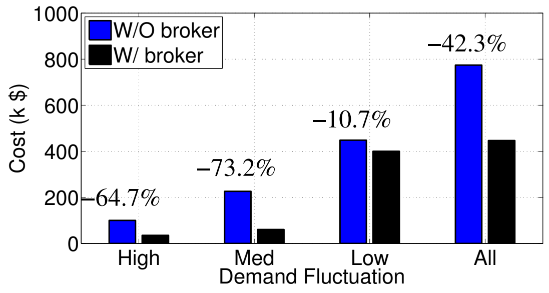


(a) Cost savings in different user groups with different reservation periods (50% reservation discount).

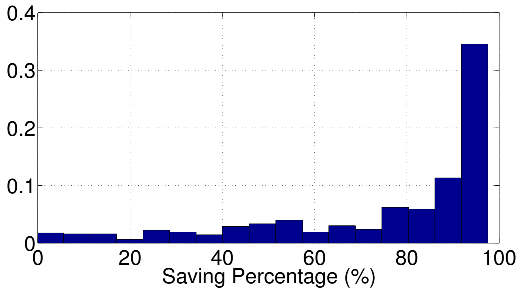


(b) Cost savings with different reservation discounts and reservation periods.

Fig. 15. Cost savings achieved by the Greedy strategy with different reservation periods and discounts.



(a) Aggregate cost savings.



(b) Histogram of individual cost savings.

Fig. 16. Cost savings with a daily billing cycle under the Greedy strategy.

7 Discussion and Future Work

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 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 [3]. 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 [5] 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. 15a (where non-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. 2, 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 [3]. 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 [13]. 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. In that case, our algorithms still apply, and the experimental observations will be similar.

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 their prices are generally *sub-additive* [3], the cost of provisioning aggregated resources is much cheaper than the total cost of purchasing them individually from the cloud.

There are several interesting problems worth further investigation in the future. To begin with, in some occasions, especially when demand is high, a cloud provider (e.g., EC2) may reject requests of creating on-demand instances due to a lack of resources. Our current formulation does not take into account the risk of unavailable on-demand instances. However, we note that such a risk is not introduced by the broker and is intrinsic to all cloud users. Even purchasing directly from the cloud, as long as the aggregate demand exceeds some supply threshold, a user's on-demand request may be declined anyway. The only difference when using the broker is that the risk must be shared by all users. Note that reserving additional instances when on-demand instances are unavailable eliminates this risk, yet at a higher price. A risk-sharing mechanism is therefore needed to allow each user to share a fair portion of the incurred penalty. We believe discussions based on the rich literature on cost-sharing mechanisms (e.g., Ch. 15 in [16]) will lead to an interesting future direction. Also, it has been shown in many cases that the use of Spot Instances [3] can further reduce the instance acquisition costs, which we have not considered in the current formulation. This serves as another interesting direction for further investigation.

8 Related Work

Three types of pricing options are currently adopted in IaaS clouds. Besides the on-demand and reserved instances introduced in Sec. 2, we note that some cloud providers charge dynamic prices that fluctuate over time, e.g., the Spot Instances in Amazon EC2 [3]. Some existing works discuss how to leverage these pricing options to reduce instance running costs for an individual user. For example, Chohan et al. [17] investigate the use of Spot Instances as accelerators of the MapReduce process to speed up the overall MapReduce time while significantly reducing monetary costs. Zhao et al. [18] propose resource rental planning with EC2 spot price predictions to reduce the operational cost of cloud applications. Hong et al. [19] design an instance purchasing strategy to reduce the "margin cost" of over-provisioning. [19] 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. Chaisiri et al. [20] investigate a similar problem and propose an algorithm by solving a stochastic integer programming problem. Their algorithm limits the reservation decisions to be made at some specific time phases. The recent work of [13] proposes optimal online strategies to reserve instances without any *a priori* knowledge of future demands. Vermeersch [21] implements a prototype software that dynamically retrieves instances from Amazon EC2 based on the user workload. All these works offer a *consulting service*, e.g., [22], [23], [24], that helps an *individual user* make instance purchasing decisions.

IaaS cloud brokers have recently emerged as *intermediators* connecting buyers and sellers of computing resources. For example, SpotCloud [25] offers a "clearinghouse" in which companies can buy and sell unused cloud computing capacity. Buyya et al. [26] discuss the engineering aspects of using brokerage to interconnect clouds into a global cloud market. Song et al. [27], 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, [28] 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 [29], the anti-correlation between the demands of different cloud tenants is exploited to save bandwidth reservation cost in the cloud. [30] empirically evaluates the idea of statistical multiplexing and resource overbooking 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.

9 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] W. Wang, D. Niu, B. Li, and B. Liang, "Dynamic cloud resource reservation via cloud brokerage," in *Proc. IEEE ICDCS*, 2013.

[2] 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.

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

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

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

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

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

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

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

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

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

[12] R. Fleischer, "On the bahncard problem," *Theoretical Computer Science*, vol. 268, no. 1, pp. 161–174, 2001.

[13] W. Wang, B. Li, and B. Liang, "To reserve or not to reserve: Optimal online multi-instance acquisition in iaas clouds," in *Proc. USENIX Intl. Conf. Autonomic Computing (ICAC)*, 2013.

[14] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM SoCC*, 2012.

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

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

[17] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz, "See spot run: Using spot instances for mapreduce workflows," in *Proc. USENIX HotCloud*, 2010.

[18] 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.

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

[20] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of resource provisioning cost in cloud computing," *IEEE Trans. Services Comput.*, vol. 5, no. 2, pp. 164–177, 2012.

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

[22] "Cloudability," http://cloudability.com.

[23] "Cloudyn," http://www.cloudyn.com.

[24] "Cloud Express," https://www.cloudexpress.com.

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

[26] 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 Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–619, 2009.

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

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

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

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

PLACE PHOTO HERE

Wei Wang received the B.Engr. and M.A.Sc degrees from the Department of Electrical Engineering, Shanghai Jiao Tong University, in 2007 and 2010. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at the University of Toronto. His general research interests cover the broad area of computer networking, with special emphasis on resource management and scheduling in cloud computing systems. He is also interested in problems at the intersection of computer networking and economics.

PLACE PHOTO HERE

Di Niu received the B.Engr. degree from the Department of Electronics and Communications Engineering, Sun Yat-sen University, China, in 2005 and the M.A.Sc. and Ph.D. degrees from the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, in 2009 and 2013. Since September, 2012, he has been with the Department of Electrical and Computer Engineering at the University of Alberta, where he is currently an Assistant Professor.

He was a recipient of the NSERC Postgraduate Scholarship 2010-2012 and a recipient of the NSERC Alexander Graham Bell Canada Graduate Scholarship 2006-2008. His research interests span the areas of multimedia delivery systems, cloud computing and storage, data mining and statistical machine learning for social and economic computing, distributed and parallel computing, and network coding.

PLACE PHOTO HERE

Ben Liang received honors-simultaneous B.Sc. (valedictorian) and M.Sc. degrees in Electrical Engineering from Polytechnic University in Brooklyn, New York, in 1997 and the Ph.D. degree in Electrical Engineering with Computer Science minor from Cornell University in Ithaca, New York, in 2001. In the 2001 - 2002 academic year, he was a visiting lecturer and post-doctoral research associate at Cornell University. He joined the Department of Electrical and Computer Engineering at the University of Toronto in 2002, where he is now a Professor. His current research interests are in mobile communications and networked systems. He is an editor for the IEEE Transactions on Wireless Communications and an associate editor for the Wiley Security and Communication Networks journal, in addition to regularly serving on the organizational or technical committee of a number of conferences. He is a senior member of IEEE and a member of ACM and Tau Beta Pi.

PLACE PHOTO HERE

Baochun Li received the B.Engr. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995 and the M.S. and Ph.D. degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000. Since 2000, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently a Professor. He holds the Nortel Networks Junior Chair in Network Architecture and Services from October 2003 to June 2005, and the Bell Canada Endowed Chair in Computer Engineering since August 2005. His research interests include large-scale multimedia systems, cloud computing, peer-to-peer networks, applications of network coding, and wireless networks. Dr. Li was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems in 2000. In 2009, he was a recipient of the Multimedia Communications Best Paper Award from the IEEE Communications Society, and a recipient of the University of Toronto McLean Award. He is a member of ACM and a senior member of IEEE.