An Economical and SLO-Guaranteed Cloud Storage Service across Multiple Cloud Service Providers

Guoxin Liu and Haiying Shen
Department of Electrical and Computer Engineering
Clemson University, Clemson, SC 29631, USA
{guoxinl, shenh}@clemson.edu

***Abstract*—It is important for cloud service brokers to provide a multi-cloud storage service to minimize their payment cost to cloud service providers (CSPs) while providing service level objective (SLO) guarantee to their customers. Many multi-cloud storage services have been proposed or payment cost minimization or SLO guarantee. However, no previous works fully leverage the current cloud pricing policies (such as resource reservation pricing) to reduce the payment cost. Also, few works achieve both cost minimization and SLO guarantee. In this paper, we propose a multi-cloud Economical and SLO-guaranteed Storage Service (ES^3), which determines data allocation and resource reservation schedules with payment cost minimization and SLO guarantee. ES^3 incorporates (1) a coordinated data allocation and resource reservation method, which allocates each data item to a datacenter and determines the resource reservation amount on datacenters by leveraging all the pricing policies; (2) a genetic algorithm based data allocation adjustment method, which reduce data Get/Put rate variance in each datacenter to maximize the reservation benefit; and (3) a dynamic request redirection method, which dynamically redirects a data request from a reservation-overutilized datacenter to a reservation-underutilized datacenter to further reduce the payment. Our trace-driven experiments on a supercomputing cluster and on real clouds (i.e., Amazon S3, Windows Azure Storage and Google Cloud Storage) show the superior performance of ES^3 in payment cost minimization and SLO guarantee in comparison with previous methods.**

I. Introduction

Cloud storage (e.g., Amazon S3 [1], Microsoft Azure [2] and Google Cloud Storage [3]), as an emerging commercial service, is becoming increasingly popular. This service is used by many current web applications, such as online social networks and web portals, to serve geographically distributed clients worldwide. In order to maximize profits, cloud customers must provide low data Get/Put latency and high availability to their clients while minimizing the total payment cost to the Cloud Service Providers (CSPs). Since different CSPs provide different storage service prices, customers tend to use services from different CSPs instead of a single CSP to minimize their payment cost (cost in short). However, the technical complexity of this task makes it non-trivial to customers, which calls for the assistance from a third-party organization. Under this circumstance, cloud service brokers [4] have emerged. A broker collects resource usage requirements from many customers, generates data allocation (including data storage and Get request allocation) over multiple clouds, and then makes resource requests to multiple clouds. It pays the CSPs for the actually consumed resources as a customer and charges its customers as a CSP. Cloud service brokers usually offer prices lower than CSPs' prices to attract more customers, which in turn helps reduce the brokers' cost by leveraging different pricing policies as explained below.

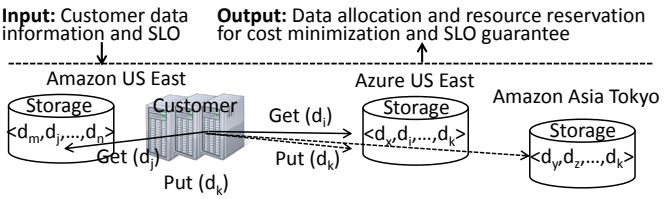


Fig. 1. An example of multi-cloud storage service.

First, datacenters in different areas of a CSP and datacenters of different CSPs in the same area offer different prices for resource usages including data Get/Put, Storage and Transfer. Second, the Storage/Transfer pricing follows a tiered model, which supplies a cheaper unit price for a larger size of data stored/transferred and vice versa. For example, in Amazon S3 US East, the unit price per GB decreases to $0.0275 when the data size is larger than 500TB. Third, the data transfer prices are different depending on whether the destination datacenter belongs to the same CSP or the same location as the source datacenter. Fourth, besides the pay-as-you-go pricing model, in which the consumer pays the CSPs based on resources actually used, CSPs also offer reservation pricing model [5], in which a consumer reserves its resource usage for a certain time in advance with much lower price (e.g., 53%-76% lower in Amazon DynamoDB [5]).

It is important for cloud service brokers to provide a multi-cloud storage service that leverages all these pricing policies to minimize their payment cost to CSPs while providing Service Level Agreement (SLO) guarantee to their customers. As shown in Figure 1, the cloud storage service determines the data allocation and resource reservation schedules among datacenters over clouds given customers' data information (i.e., data sizes and request rates) and their SLO requirements.

In spite of many previous research efforts devoted to minimizing the payment cost (or resource usage) or ensuring data retrieval SLOs in creating a cloud storage service [6], [7], [8], [9], [10], there are no previous works that fully utilize all the aforementioned pricing policies (such as resource reservation pricing and tiered pricing policies) or consider request rate variance for cost minimization and SLO guarantee. Also, most works aim to either minimize cost [6], [7] or provide SLO

guarantee [8], [9] but not both. To handle these problems, in this paper, we propose a multi-cloud Economical and SLO-guaranteed Storage Service (ES^3) for brokers to automatically generate data allocation and resource reservation schedules for cost minimization and SLO guarantee. As far as we known, this is the first work to build a multi-cloud storage service that fully leverages all aforementioned pricing policies (especially the resource reservation pricing policy) for cost minimization, and also simultaneously provides SLO-guaranteed service.

To minimize the payment cost, a broker needs to maximize reservation benefit (i.e., cost savings from reservation compared to the pay-as-you-go pricing), which however is a formidable challenge. A broker reserves a certain amount of Gets/Puts during a reservation time (denoted by T). For each billing period (denoted by t_k) in T, the amount of Gets/Puts under reservation is charged by the reservation price, and the amount of overhang of the reservations is charged by the pay-as-you-go price. Reserving the exact usage amount leads to the maximum reservation benefit while a reserved amount higher or lower than the exact usage amount leads to lower reservation benefit. However, the Get/Put rates on a datacenter may vary among different t_ks during T, which reduces the reservation benefit on the datacenter. For example, in Facebook, data is usually read heavily soon after its creation, and then is accessed rarely [11]. Also, static data allocation and resource reservation schedules based on the predicted data Get/Put rates in t_k may not always minimize the cost since the rates may vary greatly from the expected rates. Such request rate variance may make the resource consumption on some datacenters much higher or lower than their reserved amounts, which reduces the reservation benefit.

Therefore, ES^3 needs to handle three problems arisen in leveraging the reservation pricing policy to minimize cost: (1) how to make the resource reservation schedule so that the reservation benefit can be maximized? (2) how to further reduce the variance of the Get/Put rates in different t_ks over T in each datacenter to maximize its reservation benefit? (3) how to dynamically balance the Get/Put rates among datacenters to maximize the total reservation benefit?

To handle problem (1), ES^3 smartly relies on the data allocation. Through analysis, we find that increasing the minimum resource usage in a t_k during T on a datacenter (denoted by A_1) can increase the reservation benefit on the datacenter. Thus, when selecting a datacenter to allocate each data item, ES^3 selects the datacenter that increases A_1 the most as an option. Then, based on the determined data allocation schedule, ES^3 determines the resource reservation schedule that maximizes the reservation benefit of each datacenter. To handle problem (2), ES^3 uses the Genetic Algorithm (GA) [12] that is routinely used to generate useful solutions to optimization problems by mimicking the process of natural selection. It conducts crossover between different data allocation schedules to find a schedule that generates the minimum payment cost. To handle problem (3), ES^3 uses data request redirection that forwards a data request from a reservation-overutilized datacenter to a reservation-underutilized datacenter. Accordingly, we summarize our contribution below:

(1) A coordinated data allocation and reservation method, which proactively helps to maximize reservation benefit in data allocation scheduling and then determines the resource reservation schedule. Moreover, this method leverages all the aforementioned pricing policies to reduce cost and also provides SLO guarantee.

(2) A GA-based data allocation adjustment method, which further adjusts the data allocation to reduce the variance of data Get/Put rates over time between different billing periods in each datacenter in order to maximize the reservation benefit.

(3) A dynamic request redirection method, which dynamically redirects a Get request from a reservation-overutilized datacenter to a reservation-underutilized datacenter with sufficient resource to serve the Get request to further reduce the cost.

(4) We conduct extensive trace-driven experiments on a supercomputing cluster (Palmetto [13]) and real clouds (i.e., Amazon S3, Windows Azure Storage and Google Cloud Storage) to show the effectiveness of ES^3 in cost minimization and SLO guarantee in comparison with previous methods.

In addition to achieving both cost minimization and SLO guarantee, ES^3 is novel in other three aspects for reducing cost: i) it leverages all the pricing policies, ii) it proactively maximizes reservation benefit in data allocation scheduling, and iii) it tries to fully utilize reserved resources. Note that in addition to brokers, ES^3 can also be directly used by a cloud customer for the same objective.

In the following, Section II formulates the data allocation and reservation problem for cost minimization and SLO guarantee. Sections III-B, III-C and III-D present the three methods in ES^3, respectively. Section IV presents experimental results. Section V presents the related work. Section VI concludes this work with remarks on our future work.

II. Problem Statement

A. System Model

A customer deploys its application on one or multiple datacenters, which we call *customer datacenters*. We use $\mathcal{D}_c$ to denote the customer datacenters of all customers and use $dc_i \in \mathcal{D}_c$ to denote the i^{th} customer datacenter. $\mathcal{D}_s$ denotes the set of the storage datacenters of all CSPs and $dp_j \in \mathcal{D}_s$ denotes the j^{th} storage datacenter. D denotes the set of all customers' data items, and $d_l \in D$ denotes the l^{th} data item. As in [14], the SLO indicates the maximum allowed percentages of Gets/Puts beyond their deadlines. We use $\epsilon^g(d_l)$ and $\epsilon^p(d_l)$ to denote the percentages and use $L^g(d_l)$ and $L^p(d_l)$ to denote the Get/Put deadlines in the SLO of the customer of d_l. In order to ensure data availability [15] in datacenter overloads or failures, like current storage systems (e.g., Google File System (GFS)) and Windows Azure), ES^3 creates a constant number (β) of replicas for each data item. The first of the β replicas serves the Get requests while the others ensure the data availability.

CSPs charge three different types of resources: the storage measured by the data size stored in a specific region, the data

transfer to other datacenters operated by the same or other CSPs, and the number of Get/Put operations [5]. We use α_{dp_j} to denote the reservation price ratio, which represents the ratio of the reservation price to the pay-as-you-go price for Get/Put operations. ES^3 needs to predict the size and Get/Put request rates of each data item (d_l) based on the past T periods to generate the data allocation schedule. For new data items, the information can be provided by customers if it is known in advance; otherwise, they can be randomly assigned to datacenters initially. Previous study [10] found that a group of data objects with requesters from the same location has a more stable request rate than each single item. Thus, in order to have relatively stable request rates for more accurate rate prediction, ES^3 groups data objects (the smallest unit of data) from the same location to one data item as in [16].

B. Problem Objective and Constraints

We formulate the problem to find the optimal data allocation and resource reservation schedules for cost minimization and SLO guarantee using an integer programming.

Payment minimization objective. We aim to minimize the total cost for a broker (denoted by C_{sum}), including Storage, Transfer, Get and Put costs during reservation time T, which are denoted by C_s, C_t, C_g and C_p, respectively. C_s equals the sum of the storage costs of all storage datacenters in all billing periods within T. The storage cost of a storage datacenter in a billing period equals the product of unit storage price and the size of stored data in the datacenter. C_t is calculated by the product of the unit price and the size of imported data. C_g and C_p can be calculated by deducting the reservation benefit from the pay-as-you-go cost, which is calculated by the product of total number of Gets/Puts and the pay-as-you-go unit price. We use $R^g_{dp_j}$ to denote the number of reserved Gets in dp_j and calculate the Get reservation benefit in dp_j ($f^g_{dp_j}(R^g_{dp_j})$) by:

$$f^g_{dp_j}(R^g_{dp_j}) = (\sum_{t_k \in T} R^g_{dp_j} * (1 - \alpha_{dp_j}) - O^g_{dp_j}(R^g_{dp_j})) * p^g_{dp_j}, \quad (1)$$

where $p^g_{dp_j}$ is the unit Get price, and $O^g_{dp_j}(R^g_{dp_j})$ is the over reserved Get rates including the cost for over reservation and the over calculated saving and it is calculated by

$$O^g_{dp_j}(R^g_{dp_j}) = \sum_{t_k \in T} Max\{0, R^g_{dp_j} - \sum_{dc_i \in \mathcal{D}_c} r^{t_k}_{dc_i,dp_j} * t_k\}, \quad (2)$$

where $r^{t_k}_{dc_i,dp_j}$ denotes the Get rate from dc_i to dp_j during t_k. We calculate the Put reservation benefit ($f^p_{dp_j}(R^p_{dp_j})$) similarly.

The payment cost of the broker ES^3 for its customer c_n is:

$$C^{c_n}_{sum} = C_s * \gamma^{c_n}_s + C_t * \gamma^{c_n}_c + C_g * \gamma^{c_n}_g + C_p * \gamma^{c_n}_p, \quad (3)$$

where $\gamma^{c_n}_s$, $\gamma^{c_n}_c$, $\gamma^{c_n}_g$ and $\gamma^{c_n}_p$ are the percentages of c_n's usages in all customers' usages of different resources.

Constraints. First, ES^3 needs to ensure that a request is served by a datacenter having a replica of its targeting data **(Constraint 1)**. ES^3 also needs to ensure that each Get/Put satisfies the Get/Put SLO. We use $F^g_{dc_i,dp_j}(x)$ and $F^p_{dc_i,dp_j}(x)$ to denote the cumulative distribution function (CDF) of Get and Put latency from dc_i to dp_j, respectively. Thus, $F^g_{dc_i,dp_j}(L^g(d_l))$ and $F^p_{dc_i,dp_j}(L^p(d_l))$ are the percentage of Gets and Puts from dc_i to dp_j within the latencies $L^g(d_l)$ and $L^p(d_l)$, respectively. Accordingly, for each customer datacenter dc_i, we can find a set of storage datacenters that satisfy the Get SLO for Gets from dc_i targeting d_l, i.e.,

$$S^g_{dc_i,d_l} = \{dp_j | F^g_{dc_i,dp_j}(L^g(d_l)) \geq (1 - \epsilon^g(d_l))\}.$$

We define G_{dc_i} as the whole set of Get/Put data requested by dc_i during T. For each data $d_l \in G_{dc_i}$, we can find another set of storage datacenters:

$$S^p_{d_l} = \{dp_j | \forall dc_i \forall t_k, (u^{d_l,t_k}_{dc_i} > 0) \rightarrow (F^g_{dc_i,dp_j}(L^p(d_l) \geq 1 - \epsilon^p(d_l))\}$$

that satisfy Put SLO of d_l, where $u^{d_l,t_k}_{dc_i}$ denotes the Put rate targeting d_l from dc_i during t_k. The intersection of the two sets, $S^p_{d_l} \cap S^g_{dc_i,d_l}$, includes the datacenters that can serve d_l's requests from dc_i with Get and Put SLO guarantees. Therefore, any storage datacenter that serves d_l's Get/Put requests from dc_i should belong to $S^p_{d_l} \cap S^g_{dc_i,d_l}$ **(Constraint 2)**.

ES^3 needs to maintain a constant number (β) of replicas for each data item requested by datacenter dc_i to ensure data availability **(Constraint 3)**. Finally, ES^3 needs to ensure that each datacenter's Get/Put capacity is not exceeded by the total amount of Gets/Puts from all customers **(Constraint 4)**.

Problem. The problem is to find data allocation schedule and resource reservation schedule that achieves:

$$\min \ C_{sum} = C_s + C_t + C_g + C_p \quad (4)$$
$$s.t. \quad \text{Constraints 1, 2, 3 and 4.}$$

A simple reduction from the generalized assignment problem [17] can be used to prove this problem is NP-hard.

III. The Design of ES^3

A. Overview of ES^3

Due to the hardness of the above formulated problem, we propose a heuristic solution, called coordinated data allocation and reservation method (Section III-B). It determines the data allocation first (that proactively increase the reservation benefit) and then determines the resource reservation schedule based upon the data allocation schedule. To maximize the reservation benefit, as shown in Figure 2, ES^3 can use its GA-based data allocation adjustment method (Section III-C) to improve the data allocation schedule before determining the resource reservation schedule.

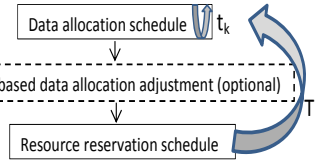


Fig. 2. Sequence of scheduling.

Using these methods, at the beginning of each reservation time T, the master server in ES^3 determines the two schedules based on its predicted data size and Get/Put rates of each data item in the next billing period t_k. To facilitate the prediction, each customer datacenter dc_i measures and reports this information and Get/Put latency distribution to storage datacenters to the master after each t_k. The resource reservation in each datacenter will not be changed during the entire reservation time T. Since the Get/Put latency and rates vary over time, the data allocation schedule under the fixed reservation schedule needs to update after each t_k in order to reduce the cost. The dynamic request redirection method (Section III-D) is used whenever a Get request will be sent to a reservation-overutilized datacenter.

B. Coordinated Data Allocation and Resource Reservation

In Section III-B1, we present how to schedule resource reservation given a data allocation schedule, and a rule that needs to follow in data allocation scheduling to increase reservation benefit. In Section III-B2, we present how to schedule the data allocation by following this rule and leveraging all the pricing policies for cost minimization and SLO guarantee.

1) Resource Reservation: First, we introduce how to find the optimal reservation amount on each datacenter that maximizes the reservation benefit given a data allocation schedule. We take the Get reservation for datacenter dp_j as an example to explain the method. The determination of the Put reservation is the same as the Get reservation. We use $B_{dp_j} = Max\{f^g_{dp_j}(R^g_{dp_j})\}_{R^g_{dp_j}\in\mathbb{N}\cup\{0\}}$ to denote the largest reservation benefit for dp_j given a specific data allocation. We use $A^{t_k} = \sum_{dc_i\in\mathcal{D}_c} r^{t_k}_{dc_i,dp_j} * t_k$ to denote the number of Gets served by dp_j during t_k, and define $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ as a list of all A^{t_k}s of different $t_k \in T$ sorted in an increasing order. As shown in Figure 3(a), for datacenter dp_1, if the reservation is the amount of Gets in billing period t_1, since the usage in t_2 is much higher than the reserved amount, the payment in t_2 is high. If the reservation is the amount of Gets in t_2, then since the real usage in t_1 is much lower, the reserved amount is wasted. It is a challenge to determine the optimal reservation.

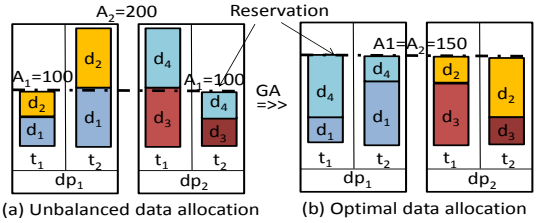


Fig. 3. Unbalanced and optimal data allocation.

We can prove that when $R^g_{dp_j} \in [A_i, A_{i+1}]$ $(i = 1, .., n-1)$, reservation benefit $f^g_{dp_j}(R^g_{dp_j})$ increases or decreases monotonically within $[A_i, A_{i+1}]$ (as shown in Appendix A). This means that the reservation benefit reaches the maximum when $R^g_{dp_j} \in \mathcal{A}$. Thus, the optimal reservation is the A_i $(i \in [1, n-1])$ that generates the largest reservation benefit, i.e.,

$$B_{dp_j} = Max\{f^g_{dp_j}(A_i)\}_{A_i\in\mathcal{A}}. \quad (5)$$

Then, based on the determined data allocation, we use Equation (5) to determine the reserved amount for each datacenter.

Next, we show how the data allocation can proactively help increase the reservation benefit when selecting a datacenter to allocate a data item. As shown in Appendix A, we can also prove that for $R^g_{dp_j} \in [0, A_1]$, $f^g_{dp_j}(R^g_{dp_j})$ is positively proportional to $R^g_{dp_j}$. Also, the maximum reservation benefit is no less than the reservation benefit of choosing $R^g_{dp_j} = Min\{A_i\}_{A_i\in\mathcal{A}} = A_1$. Therefore, in order to maximize reservation benefit on datacenter dp_j, we can enlarge its lower bound $f^g_{dp_j}(A_1)$, which needs to enlarge A_1 in data allocation. Thus, in data allocation, we should follow the following rule:

Rule 1: Among several datacenter candidates to allocate a data item, we need to choose the datacenter that leads to the largest A_1 increment after being allocated with the data item.

2) Data Allocation: Before we explain the datacenter selection for a data item, we first introduce a concept of Storage/Get/Put-intensive data item. A data item d_l's payment cost consists of Get, Put, Transfer and Storage cost denoted by $C^{d_l}_s$, $C^{d_l}_g$, $C^{d_l}_t$ and $C^{d_l}_p$. Transfer conducts one-time data import to clouds and is unlikely to become the dominant cost. We consider data item d_l as Storage-intensive if $C^{d_l}_s$ dominates the total cost (e.g, $C^{d_l}_s \gg C^{d_l}_g + C^{d_l}_p$), and the Get/Put-intensive data items are defined similarly. Many data items have certain operation patterns and accordingly become Get-, Put- or Storage-intensive. For example, the instant messages in Facebook are Put-intensive [18]. In the web applications such as Facebook, the old data items with rare Gets/Puts [11] become Storage-intensive. In addition, recall that only one copy of the β replicas of each data item is responsible for the Get requests, the remaining $\beta - 1$ replicas then become either Put or Storage intensive. In order to reduce cost, a Get, Put or Storage-intensive replica is allocated to a datacenter with the cheapest unit price for Get, Put or Storage, respectively.

Algorithm 1: Data allocation scheduling algorithm.

```
for each dc_i in D_c do
    for each d_l requested by dc_i do
        while the number of replicas of d_l is less than β do
            if first replica of d_l then
                It is assigned to serve requests from
                dc_i towards d_l; All other replicas do not serve Gets;
            if d_l is Storage intensive then
                L = {(dp_j with the largest S_{dp_j} among all
                datacenters having the smallest Storage unit price)
                ∧(dp_j ∈ S^p_{d_l} ∩ S^g_{dc_i,d_l})∧(dp_j with enough Ge/Put
                capacity) };
            else if d_l is Get/Put intensive then
                L = {(dp_j with the smallest Get/Put unit price ∨
                with the lowest unit reservation price ∨ with the
                largest increment of A_1)
                ∧(dp_j ∈ S^p_{d_l} ∩ S^g_{dc_i,d_l})∧(dp_j with enough Get/Put
                capacity)};
            else if d_l is non-intensive then
                L is the union of all the above L sets;
            d_l is allocated to dp_j in L with the smallest C_sum;
```

Next, we introduce how to identify the datacenter to store a given data item. For each data item, the first replica handles all Get requests (Constraint 1), and all other replicas do not handle the Get requests. Section II-B indicates that datacenters in $(S^p_{d_l} \cap S^g_{dc_i,d_l})$ satisfy the SLO of data item d_l (Constraint 2) and Constraint 4 must be satisfied to ensure that the allocated datacenters have enough Get/Put capacity for d_l. Among these qualified datacenters, we need to choose β (Constraint 3) datacenters that can reduce the cost as much as possible (Objective (4)). In the datacenter selection, we consider all current pricing policies as presented in Section I. First, storing the data in the datacenter that has the cheapest unit price for its dominant cost (e.g., Get, Put or Storage) can reduce the cost greatly. Second, if the data is Storage-intensive, based on the tiered pricing policy, storing the data in the datacenter that results in the largest aggregate storage size $\mathbb{S}_{dp_j}$ can reduce the cost greatly. Third, if the data is Get/Put-intensive,

in order to minimize the reservation cost, we should choose the datacenters with the lowest unit reservation price and the datacenters selected following Rule 1 in Section III-B1. Based on these three considerations, the datacenter candidates to store the data are selected. Among these selected datacenters, the one with the smallest C_{sum} is finally identified to store the data. Algorithm 1 shows the pseudocode for the data allocation algorithm. After each billing period, using Algorithm 1, ES^3 finds a new data allocation schedule and calculates its C_{sum}. It compares the new C_{sum} with previous C_{sum}, and chooses the data allocation schedule with smaller C_{sum}.

After determining the data allocation schedule, ES^3 needs to transfer a data replica from a source datacenter with the replica to the assigned datacenter. To reduce cost (Objective (4)), ES^3 takes advantage of the tiered pricing model of Transfer to reduce the Transfer cost. It assigns priorities to the datacenters with the replica for selection in order to have a lower unit price of Transfer. Specifically, for the datacenters belonging to the same CSP of assigned datacenter dp_j, those in the same location as dp_j have the highest priority, and those in different locations from dp_j have a lower priority. The datacenters that do not belong to dp_j's CSP have the lowest priority, and are ordered by their current unit transfer prices (under the aggregate transfer data size) in an ascending order to assign priorities. Finally, the datacenter with the highest priority is chosen as the source datacenter to transfer data.

C. GA-based Data Allocation Adjustment

If the allocated Get/Put rates vary over time largely (i.e., the rates exceed and drop below the reserved rates frequently), then the reservation saving is small according to Equation (1). For example, Figure 3(a) shows a data allocation schedule. Then, both $R^g_{dp_j} = 100$ and $R^g_{dp_j} = 200$ reduce reservation benefit at a billing period. We propose the GA-based data allocation adjustment method to make the reserved amount approximately equal to the actual usage as shown in Figure 3(b).

Fig. 4. GA-based data allocation adjustment.

As shown in Figure 4, this method regards each data allocation schedule, represented by $<d_l, \{dp_1, ..., dp_\beta\}>$ ($d_l \in D$), as a genome string, where $\{dp_1, ..., dp_\beta\}$ (denoted by $\mathbb{G}_{d_l}$) is the set of datacenters that store d_l. Using Algorithm 1, it generates the data allocation schedule with the lowest total cost (named as global optimal schedule). It also generates the data allocation schedules with the lowest Storage cost, lowest Get cost and lowest Put cost (named as local optimal schedules) by assuming all data items as Storage-, Get- and Put-intensive, respectively.

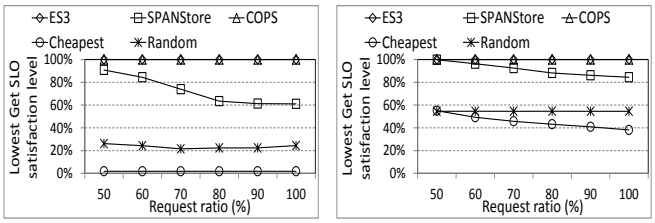
To generate the children of the next generation, this method conducts crossover between the global optimal schedule with each local optimal schedule with crossover probability θ (Figure 4). Each genome in a child's genome string is from either the global optimal schedule (with probability θ) or the local optimal schedule (with probability 1-θ). To ensure the schedule validity, for each crossover, the genomes that do not meet all constraints in Section II-B are discarded. In order not to be trapped into a sub-optimal result, the genome mutation occurs in each genome string after the crossover with a certain probability. In the mutation of a genome, for each data item, dp_1 in $\mathbb{G}_{d_l}$ (which serves Gets) and a randomly selected dp_k in $\mathbb{G}_{d_l}$ are replaced with qualified datacenters.

After a crossover and mutation, the global optimal schedule and the local optimal schedules are updated accordingly. Among the child schedules and the global optimal schedule, the one with the smallest C_{sum} (based on Equation (4)) is selected as the new global optimal schedule. Similarly, we evaluate each schedule's Storage/Get/Put cost exclusively to generate the new Storage/Get/Put local optimal schedules, respectively. In order to speed up the convergence to the optimal solution, the number of children in the next generation (N_g) is inversely proportional to the improvement of the global optimal schedule in the next generation. That is, $N_g = Min\{N, \frac{N}{C_{sum}/C'_{sum}}\}$, where N is a constant integer as the base population, C_{sum} and C'_{sum} are the total cost of current and new global optimal schedules, respectively. Creating generation is terminated when the maximum number of consecutive generations without cost improvement or the largest number of generations is reached. Though this method is time consuming, it is only executed once at the beginning of reservation time period T (e.g., one year in Amazon DynamoDB [5]).

D. Dynamic Request Redirection

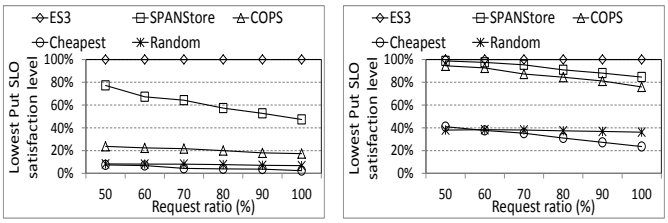
ES^3 master predicts the Get load of each storage datacenter dp_j at the initial time of t_k (A^{t_k}), which is used to calculate the data allocation schedule. If the actual number of Gets is larger or smaller than A^{t_k}, then the schedule may not reach the goal of SLO guarantee and minimum cost. There may be a request burst due to a big event, which leads to an expensive resource usage under current request allocation among storage datacenters. Sudden request silence may lead to a waste of reserved usage. The Get operation only needs to be resolved by one of β replicas. Therefore, we can redirect the burst Gets on a datacenter that uses up its reservation to a replica in a datacenter whose reservation is underutilized in order to save cost. This redirection can also be conducted whenever a datacenter overload or failure is detected.

We consider a datacenter *reservation-overutilized* if its Get load is higher than its reserved number of Gets and use threshold $T_{max} = A^{t_k}/t_k$ to check whether a datacenter is reservation-overutilized. We consider a datacenter *reservation-underutilized* if its reserved Gets are not fully used and use threshold $T_{min} = R^g_{dp_j}/t_k$ to check whether a datacenter is reservation-underutilized. The master calculates the aggregate



(a) In simulation (b) In real clouds

Fig. 5. Get SLO guaranteed performance.



(a) In simulation (b) In real clouds

Fig. 6. Put SLO guaranteed performance.

number of Gets for each datacenter during t_k, denoted by g_{dp_j}. We used t to denote the elapsed time interval during t_k. Datacenters with $g_{dp_j}/t < T_{min}$ are reservation underutilized, datacenters with $g_{dp_j}/t \geq T_{max}$ are reservation-overutilized, and datacenters with $T_{min} < g_{dp_j}/t < T_{max}$ are called reservation-normalutilized datacenters. We aim to release the load from reservation-overutilized datacenters to reservation-underutilized datacenters in order to fully utilize the reservation. Specifically, ES^3 master sends out the three different groups to all the customer datacenters. If a customer datacenter notices that the target datacenter to serve a request is a reservation-overutilized datacenter, it selects another replica among β replicas in a reservation-underutilized datacenter with sufficient resource to serve the request and the lowest unit Get price. The consideration of the unit Get price is to reduce the cost if the redirected request uses up the reservation of the datacenter. If there are no reservation-underutilized datacenters, the reservation-normalutilized datacenter with sufficient resource to serve the request and the lowest unit Get price is selected. This way, the dynamic request redirection algorithm further reduces the cost by fully utilizing the reserved resource.

IV. Performance Evaluation

We conducted trace-driven experiments on Palmetto [13], a super computing cluster with 771 8-core nodes, and deployed ES^3 on real-world clouds. We first introduce the experimental settings.

Simulated clouds. We simulated two datacenters in each of all 25 cloud storage regions in Amazon S3, Microsoft Azure and Google cloud storage [1], [2], [3]. The distribution of the inter-datacenter Get/Put latency follows the real latency distribution as in [10]. The unit prices for Storage, Get, Put and Transfer and the reservation price ratio in each region follow the real prices listed online. We simulated ten times of the number of all customers listed in [1], [2], [3] for each cloud storage provider. As in [10], in the SLOs for all customers, the Get deadline is 100ms [10], the percentage of latency guaranteed Gets and Puts is 90%, and the Put deadline for a customer's datacenters in the same continent is 250ms and is 400ms for an over-continent customer. Also, the size of each data item of a customer was randomly chosen from $[0.1TB, 1TB, 10TB]$ [10]. The number of data items of a customer follows a bounded Pareto distribution with a lower bound, upper bound and shape as 1, 30000 and 2 [19]. We set the mutation rate, crossover rate, and the maximum number of generations in the GA-based data allocation adjustment method to 0.2, 0.8, and 200 respectively. In simulation, we set the billing period to 1 month, and we computed the cost and evaluated the SLO performance in 12 months. We run each experiment for 10 times and reported the average performance.

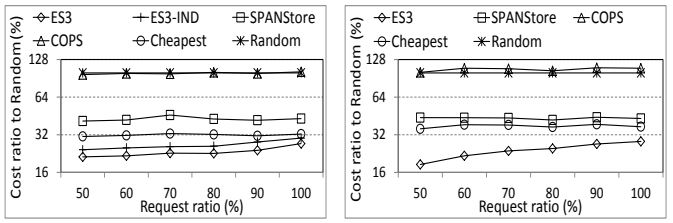
Get/put operations. The percentage of the data items visited (Get/Put) follows a bounded Pareto distribution with a upper bound, lower bound and shape as 20%, 80% and 2. The size of each requested data object was set to 100KB [10]. The Put rate follows the publicly available wall post trace from Facebook [20] and we set the Get rate of each data item based on the 100:1 Get:Put ratio [21]. We set the Get and Put capacities of each datacenter to 1E8 and 1E6 Gets/Puts per second, respectively, based on real Facebook Get/Put capacities [21]. When a datacenter is overloaded, the Get/Put operation on it was repeated once.

Real clouds. We also conducted a small scale trace-driven experiment on real-world clouds. We implemented ES^3's master in Amazon EC2's US West (Oregon) Region. We simulated one customer that has customer datacenters in Amazon EC2's US West (Oregon) Region and US East Region. Unless otherwise indicated, the settings are the same as before. Due to the small scale, the number of data items was set to 1000, the size of each item was set to 100MB, and β was set to 2. We set the Put deadline to 200ms. We set the capacity of a datacenter in each region of all CSPs as 30% of total expected Get/Put rates. Since it is impractical to conduct experiments lasting a real contract year, we set the billing period to 4 hours, and set the reservation period to 2 days.

Comparison methods. We compared ES^3 with the following systems. i) *COPS* [9]. It allocates requested data into a datacenter with the shortest latency to each customer datacenter but does not consider payment cost minimization. ii) *Cheapest*. It selects the datacenters with the cheapest cost in the pay-as-you-go manner to store each data item. It neither provides SLO guarantee nor attempts to minimize the cost with the consideration of reservations. iii) *Random*. It randomly selects datacenters to allocate each data item without considering cost minimization or SLO guarantee. iv) *SPANStore* [10]. It is a storage system over multiple CSPs' datacenters to minimize cost and support SLOs. It neither considers datacenter capacity limitations to guarantee SLOs nor considers reservation, tiered pricing model, or the Transfer price differences to minimize cost.

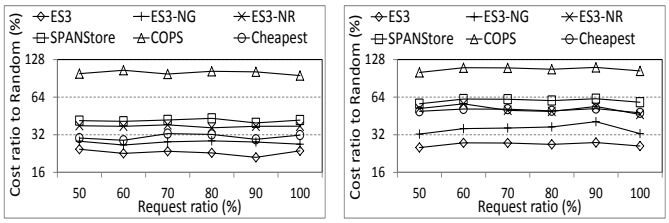
A. Comparison Performance Evaluation

In this section, we varied each data item's Get/Put rate from 50% to 100% (named as request ratio) of its actual Get/Put rate in the Facebook trace [20], with a step increase of 10%. The Get SLO satisfaction level of a customer is calculated by



(a) In simulation (b) In real clouds

Fig. 7. Payment cost minimization with normal Get/Put workload.



(a) In simulation (b) In real clouds

Fig. 8. Payment cost minimization with light Get/Put workload.

$Min\{Min\{n'_{t_k}/n_{t_k}\}_{\forall t_k \in T}, (1-\epsilon^g)\}/(1-\epsilon^g)$, where n'_{t_k} and n_{t_k} are the number of Gets within L^g and the total number of Gets of this customer, respectively. Similarly, we can get the Put SLO satisfaction level.

Figures 5(a) and 5(b) show the lowest Get SLO satisfaction level of each system in simulation and real-world experiment, respectively. We see that the result follows $100\% = ES^3$=*COPS*>*SPANStore*>*Random*>*Cheapest*. ES^3 considers both the Get SLO and capacity constraints, thus it can supply a Get SLO guaranteed service. *COPS* always chooses the provider datacenter with the smallest latency. *SPANStore* always chooses the provider datacenter with the Get SLO consideration. However, since it does not consider datacenter capacity, a datacenter may become overloaded and cannot meet the Get SLO deadline. *Random* randomly selects datacenter so it generates a lower Get SLO guaranteed performance than *SPANStore*. *Cheapest* does not consider SLOs, so it generates the worst SLO satisfaction level. Figure 6(a) and 6(b) show the lowest Put SLO satisfaction level of each system in simulation and real-world experiment, respectively. It shows the same order and trends of all systems as in Figure 5(a) due to the same reasons except for *COPS*. *COPS* allocates data without considering the Put latency minimization, and the Puts to far-away datacenters may introduce a long delay. Thus, *COPS* generates a lower Put SLO satisfaction level than *SPANStore*. Figures 5 and 6 indicate that only ES^3 can supply a both Get/Put SLO guaranteed service.

Since *Random* does not consider SLO guarantee or payment cost minimization, we measure the cost improvement of the other systems compared to *Random*. Figures 7(a) and 7(b) show the ratio of each system's cost to *Random*'s cost in simulation and real-world experiment, respectively. In order to show the effect of considering the tiered pricing model, in simulation, we also tested a variant of ES^3, denoted by ES^3-*IND*, in which each customer individually uses ES^3 to allocate its data without aggregating their workload together through the broker. The figures show that the cost follows *COPS*≈*Random*>*SPANStore*>*Cheapest*>ES^3-*IND*>ES^3. Since both *COPS* and *Random* do not consider cost, they produce the largest cost. *SPANStore* selects the cheapest datacenter in pay-as-you-go manner with SLO constraints, thus it generates a smaller cost. However, it produces a larger cost than *Cheapest*, which always chooses the cheapest datacenter. ES^3-*IND* generates a smaller cost than these methods, because it chooses the datacenter under SLO constraints that minimizes each customer's cost by considering all pricing policies. ES^3 generates the smallest cost because it further aggregates workloads from all customers to get a cheaper Storage and Transfer unit price based on the tiered pricing model. The figures confirm that ES^3 generates the smallest payment cost in all systems and the effectiveness of considering tiered pricing model.

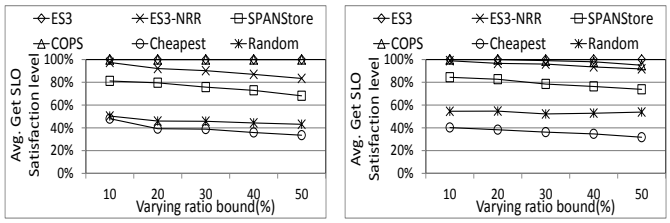
B. Effectiveness of Individual Methods in ES^3

We are interested to see whether the data intensiveness change will affect the performance of different systems. Thus, in this test, we repeated the experiments in Section IV-A with the Get/Put rates of data objects reduced by 1/10 times, which makes a larger percentage of data items Storage-intensive. In order to measure the effectiveness of GA-based data allocation on cost minimization, we varied the Get/Put rate of each data item in a billing period. Specifically, the Get/Put rate was set to $x\%$ of the rate in the previous billing period, where x was randomly chosen from $[50, 200]$ according to [10]. We use ES^3-*NG* to denote ES^3 without GA-based data allocation adjustment. In order to show the effect of considering the reservation on cost minimization, we also tested ES^3 without any reservation or GA-based method, denoted by ES^3-*NR*.

Figures 8(a) and 8(b) show the ratio of each system's cost to *Random*'s cost in simulation and real-world experiment, respectively. The figures show the same order between all systems as Figure 7(a) due to the same reasons, which indicates that the data intensiveness does not affect the performance differences between the systems. Since ES^3-*NR* also chooses the cheapest datacenters by considering different pricing policies except reservation, it produces a cheaper cost than *SPANStore*. However, by choosing datacenters with SLOs constraints that may offer a higher price than the cheapest price, ES^3-*NR* generates a larger cost than *Cheapest*, which generates a larger cost than ES^3. This result shows the effectiveness of considering reservation in cost minimization. ES^3-*NG* produces a higher cost than ES^3, which shows the effectiveness of the GA-based data allocation adjustment method in cost minimization.

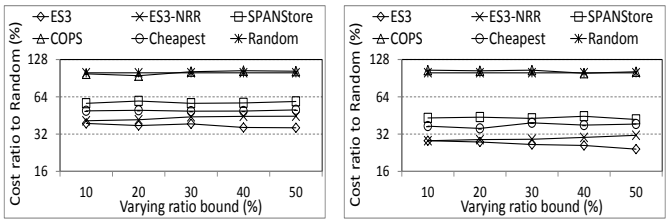
C. Performance under Dynamic Request Rates

This section measures the performance in providing Get SLO guarantee and cost minimization under dynamic request rates. We denote ES^3 without the Request Redirection method by ES^3-*NRR*. The Get rate of each data item was randomly chosen from $[(1-x)v, (1+x)v]$, where v is the Get rate, and x is called varying ratio bound and was varied from 10% to 50% in experiments. Figures 9(a) and 9(b) show the average Get



(a) In simulation (b) In real clouds

Fig. 9. SLO guarantee of Gets with varying Get rate.



(a) In simulation (b) In real clouds

Fig. 10. Cost minimization with varying Get rate.

SLO satisfaction level of all customers in simulation and real-world experiment, respectively. They show the same trends and orders of all systems as in Figures 5(a) and 5(b) due to the same reasons. The figure also shows that ES^3-*NRR* generates a lower Get SLO satisfaction level than ES^3 and *COPS* but a higher level than the others. This is because ES^3-*NRR* generates long latency on overloaded datacenters when some data items have larger request rates than expected, so it cannot supply an SLO guaranteed service in the case of varying request rates. However, due to its Get/Put SLO guarantee and capacity awareness, it generates a higher SLO satisfaction level than others. The figures indicate the high effectiveness of ES^3's dynamic request redirection method to handle the Get rate variance in ensuring Get SLO.

Figures 10(a) and 10(b) show the ratio of each system's cost to *Random*'s cost. The figures show the same order between all systems as in Figure 7(a) due to the same reasons. It also shows that ES^3-*NRR* generates a higher cost than ES^3 but a lower cost than others. Without dynamic request redirection, ES^3-*NRR* cannot fully utilize reserved resources like ES^3 and pays more for the over-utilized resources beyond the reservation. However, by leveraging all pricing policies, ES^3-*NRR* generates a lower payment cost than other systems. The figures indicate the high effectiveness of ES^3's dynamic request redirection method to reduce the payment cost in varying request rates and the superior performance of ES^3 in handling dynamic request rates for cost minimization.

V. Related Work

Storage services over multiple clouds. RACS [14] and DepSky [22] are storage systems that transparently spread the storage load over many cloud storage providers with replication in order to better tolerate provider outages or failures. *COPS* [9] allocates requested data into a datacenter with the shortest latency. Unlike these systems, ES^3 considers both SLO guarantee and payment cost minimization.

Cloud/datacenter storage payment cost minimization. In [23], [6], a cluster storage configuration automation method is proposed to use the minimum resource to support the desired workload. These works are focused on one cloud rather than a geographical distributed cloud storage service over multiple CSPs, so they do not consider the price differences from different CSPs. Puttaswamy *et al.* [7] proposed a multi-cloud file system called FCFS. FCFS considers data size, Get/Put rates, capacities and service price differences to adaptively assign data with different sizes to different storage services to minimize the cost for storage. However, it cannot guarantee the SLOs without deadline awareness. SPANStore [10] is a key-value storage system over multiple CSPs' datacenters to minimize payment cost and guarantee SLOs. However, it does not consider the datacenter capacity limitation, which may lead to SLO violation, and also does not fully leverage all pricing policies in cost minimization as indicated previously. Also, SPANStore does not consider Get/Put rate variation during a billing period, which may cause datacenter overload and violate the SLOs. ES^3 is advantageous in that it overcomes these problems in achieving SLO guarantee and cost minimization.

Pricing models on clouds. There are a number of works studying resource pricing problem for CSPs and customers. In [24], [25] and [26], dynamic pricing models including adaptive leasing or auctions for cloud computing resources are studied to maximize the benefits of cloud service customers. Roh *et al.* [27] formulated the pricing competition of CSPs and resource request competition of cloud service customers as a concave game. The solution enables the customers to reduce their payments while receiving a satisfied service. Different from all these studies, ES^3 focuses on the cost optimization for a customer deploying geographically distributed cloud storage over multiple cloud storage providers with SLO constraints.

Cloud service SLO guarantee. Spillane *et al.* [28] used advanced caching algorithms, data structures and Bloom filters to reduce data read/write latencies in a cloud storage system. Wang *et al.* [8] proposed Cake to guarantee service latency SLO and achieve high throughput using a two-level scheduling scheme of data requests within a datacenter. Wilson *et al.* [29] proposed D_3 with explicit rate control to apportion bandwidth according to flow deadlines to guarantee the SLOs. Hong *et al.* [30] adopted a flow prioritization method by all intermediate switches based on a range of scheduling principles to ensure low latencies. Zats *et al.* [31] proposed a cross-layer network stack to reduce the long tail of flow completion times. Wu *et al.* [32] adjusted TCP receive window proactively before packet drops occur to avoid incast congestions to reduce the incast delay. Unlike these works, ES^3 focuses on building a geographically distributed cloud storage service over multiple clouds with SLO guarantee and cost minimization.

VI. Conclusion

In this paper, we propose a multi-cloud Economical and SLO-guaranteed cloud Storage Service (ES^3) for a cloud broker over multiple CSPs that provides SLO guarantee and cost minimization even under the Get rate variation. ES^3 is more advantageous than previous methods in that it fully utilizes

different pricing policies and considers request rate variance in minimizing the payment cost. ES^3 has a data allocation and reservation method and a GA-based data allocation adjustment method to guarantee the SLO and minimize the payment cost. ES^3 further has a dynamic request redirection method to select a replica in a datacenter with sufficient reservation to serve the request on a reservation-overutilized datacenter in order to reduce the cost when the request rates vary greatly from the expected rates. Our trace-driven experiments on a supercomputing cluster and real different CSPs show the superior performance of ES^3 in providing SLO guarantee and cost minimization in comparison with previous systems and the effectiveness of individual methods in ES^3. In our future work, we will study how to dynamically create and delete data replicas in datacenters to fully utilize the Put reservation. We will also consider the dependency between data items for data allocation in order to expedite the data retrieval.

ACKNOWLEDGEMENTS

This research was supported in part by U.S. NSF grants NSF-1404981, IIS-1354123, CNS-1254006, IBM Faculty Award 5501145 and Microsoft Research Faculty Fellowship 8300751.

REFERENCES

[1] Amazon S3. http://aws.amazon.com/s3/.
[2] Microsoft Azure. http://www.windowsazure.com/.
[3] Goolge Cloud storage. https://cloud.google.com/storage/.
[4] D. Niu, C. Feng, and B. Li. A Theory of Cloud Bandwidth Pricing for Video-on-Demand Providers. In *Proc. of INFOCOM*, 2012.
[5] Amazon DynamoDB. http://aws.amazon.com/dynamodb/.
[6] H. V. Madhyastha, J. C. McCullough, G. Porter, R. Kapoor, S. Savage, A. C. Snoeren, and A. Vahdat. SCC: Cluster Storage Provisioning Informed by Application Characteristics and SLAs. In *Proc. of FAST*, 2012.
[7] K. P. N. Puttaswamy, T. Nandagopal, and M. S. Kodialam. Frugal Storage for Cloud File Systems. In *Proc. of EuroSys*, 2012.
[8] A. Wang, S. Venkataraman, S. Alspaugh, R. H. Katz, and I. Stoica. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *Proc. of SoCC*, 2012.
[9] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Dont Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proc. of SOSP*, 2011.
[10] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services. In *SOSP*, 2013.
[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebooks Distributed Data Store for the Social Graph. In *Proc. of ATC*, 2013.
[12] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
[13] Palmetto Cluster. http://citi.clemson.edu/palmetto/.
[14] A. Hussam, P. Lonnie, and W. Hakim. RACS: A Case for Cloud Storage Diversity. In *Proc. of SoCC*, 2010.
[15] N. Bonvin, T. G. Papaioannou, and K. Aberer. A Self-Organized, Fault-Tolerant and Scalable Replication Scheme for Cloud Storage. In *Proc. of SoCC*, 2010.
[16] G. Liu, H. Shen, and H. Chandler. Selective Data Replication for Online Social Networks with Distributed Datacenters. In *Proc. of ICNP*, 2013.
[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
[18] D. Borthakur, J. S. Sarma, J. Gray, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proc. of SIGMOD*, 2011.
[19] P. Yang. Moving an Elephant: Large Scale Hadoop Data Migration at Facebook. https://www.facebook.com/notes/paul-yang/moving-an-elephant-large-scale-hadoop-data-migration-at-facebook/10150246275318920.
[20] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the Evolution of User Interaction in Facebook. In *Proc. of WOSN*, 2009.
[21] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. of Usenix NSDI*, 2013.
[22] A. N. Bessani, M. Correia, B. Quaresma, F. Andr, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *TOS*, 2013.
[23] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. C. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proc. of FAST*, 2002.
[24] F. Wang, J. Liu, and M. Chen. CALMS: Cloud-assisted Live Media Streaming for Globalized Demands with Time/region Diversities. In *Proc. of INFOCOM*, 2012.
[25] Y. Song, M. Zafer, and K.-W. Lee. Optimal Bidding in Spot Instance Market. In *Proc. of INFOCOM*, 2012.
[26] D. Niu, B. Li, and S. Zhao. Quality-assured Cloud Bandwidth Auto-scaling for Video-on-Demand Applications. In *Proc. of INFOCOM*, 2012.
[27] H. Roh, C. Jung, W. Lee, and D. Du. Resource Pricing Game in Geo-Distributed Clouds. In *Proc. of INFOCOM*, 2013.
[28] R. P. Spillane, P. Shetty, E. Zadok, S. Dixit, and S. Archak. An Efficient Multi-Tier Tablet Server Storage Architecture. In *Proc. of SoCC*, 2011.
[29] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proc. of SIGCOMM*, 2011.
[30] C. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. of SIGCOMM*, 2012.
[31] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proc. of SIGCOMM*, 2012.
[32] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *Proc. of CoNEXT*, 2010.

APPENDIX
RESERVATION BENEFIT MONOTONICITY

Theorem 1. *For a datacenter dp_j, its reservation benefit function $f_{dp_j}(x)$ increases when $x \in [0, R'^g_{dp_j})$, and decreases when $x \in (R'^g_{dp_j}, A_n]$, where $R'^g_{dp_j}$ is the optimal reserved number of Gets that leads to the maximal reservation benefit.*

Proof. According to Equation (1), we define the increasing benefit of increased reservation as $f_I(x) = f_{dp_j}(x) - f_{dp_j}(x-1) = (n*(1-\alpha) - O'(x))*p^g_{dp_j}$, where n is number of billing periods in T. $O'(x) = O_{dp_j}(x) - O_{dp_j}(x-1)$ represents the number of billing periods during T with $\sum_{dc_i \in \mathcal{D}_c} r^{t_k}_{dc_i,dp_j} < x$. Thus, $O'(x)$ increases. At first, when $O'(x) < n*(1-\alpha)$, then $f_I(x) > 0$, which means $f_{dp_j}(x)$ increases; when $O'(x)$ is larger than $n*(1-\alpha)$, then $f_I(x) < 0$, which means $f_{dp_j}(x)$ decreases. Therefore, $f^g_{s^t_j}(x)$ increases and then decreases. Since $f^g_{s^t_j}(R'^g_{dp_j})$ reaches the largest $f(x)$, we can derive that $f^g_{s^t_j}(x)$ increases when $x \in [0, R'^g_{dp_j})$, and decreases when $x \in (R'^g_{dp_j}, A_n]$. □

Based on Theorem 1, we can derive that when $x \in [A_i, A_{i+1}]$, $f_{dp_j}(x)$ increases or decreases monotonically. This is because if $A_{i+1} \leq R'^g_{dp_j}$, then for $x \in [A_i, A_{i+1}]$, $f_{dp_j}(x)$ increases monotonically; otherwise $f_{dp_j}(x)$ decreases monotonically. For $x \in [0, A_1]$, we can transform Equation (1) to $f_{dp_j}(x) = \sum_{t_k \in T} x*(1-\alpha)*p^g_{dp_j}$. Then, we can get that for $x \in [0, A_1]$, $f_{dp_j}(x)$ is positively proportional to x.