

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

Haiying Shen* *Senior Member IEEE*, Guoxin Liu and Haoyu Wang, Student Member IEEE

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. We also propose several algorithms to enhance the cost efficient and SLO guarantee performance of ES^3 including i) dynamic request redirection, ii) grouped Gets for cost reduction, iii) lazy update for cost-efficient Puts, and iv) concurrent requests for rigid Get SLO guarantee. 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.

Keywords: Cloud storage, SLO, Data availability, Payment cost minimization.

1 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.

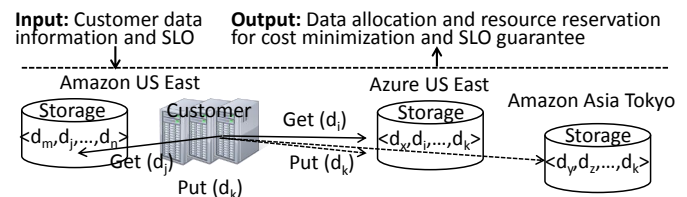


Fig. 1: An example of multi-cloud storage service.

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 pro-

• * Corresponding Author. Email: hs6ms@virginia.edu; Phone: (434) 924-8271; Fax: (434) 982-2214.

• Haiying Shen and Haoyu Wang are with the Department of Computer Science, University of Virginia, Charlottesville, VA, 22904. E-mail: {hs6ms, hw8c}@virginia.edu, guoxinl@clemson.edu

viding Service Level Objective (SLO, which is the deadline for GET/PUT requests) 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 know, 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_k s 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].

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_k s 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) Cost efficient and SLO guarantee enhancements.
 - Dynamic request redirection. By dynamic request redirection between storage datacenters considering their reserved Gets, the Get SLO guaranteed service is enhanced and the reserved Gets are more fully utilized.
 - Grouped Gets for cost reduction. By aggregating multiple Gets for objects that are often concurrently requested into one unit Get, the Get cost is further reduced.
 - Lazy update for cost-efficient Puts. By aggregating multiple sequential Puts into one unit Put, the Put cost is further reduced. Also, by deactivating the replicas not serving Gets, the Put and storage costs are saved during the period with low workload (number of Get requests).
 - Concurrent requests for rigid Get SLO guarantee. By sending concurrent requests to multiple storage datacenters, the Get SLO guaranteed service is enhanced.

(4) We conduct extensive trace-driven experiments on a supercomputing cluster 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.

Note that in addition to brokers, ES^3 can also be directly used by a cloud customer for the same objective. We also assume cloud providers notify their available computing resources, which is currently not available in this paper. The rest of this paper is organized as follows. Section 2 formulates the data allocation and reservation problem for cost minimization and SLO guarantee. Sections 3.2 and 3.3 present a data allocation and reservation method, and a genetic algorithm based data allocation adjustment to enhance the cost savings by reservation. Section 4 presents the methods for cost efficient and SLO guarantee enhancements in detail. In ES^3 , respectively. Section 5 presents the trace-driven experimental results on a supercomputing cluster and real-world clouds. Section 6 presents the related work. Section 7 concludes this work with remarks on our future work.

TABLE 1: Notations of inputs and outputs.

Input	Description	Input	Description
D_c	set of customer datacenters	dc_i	i^{th} customer datacenter
D_s	set of storage datacenters	dp_j	j^{th} storage datacenter
$c_{dp_j}^g$	Get capacity of dp_j	$c_{dp_j}^p$	Put capacity of dp_j
$p_{dp_j}^s(x)$	unit storage price of dp_j under x GB storage size	$p^t(dp_j)$	smallest unit transfer price to dp_j
$p_{dp_j}^g$	unit Get price of dp_j	$p_{dp_j}^p$	unit Put price of dp_j
$F^g(x)$	CDF of Get latency	$F^p(x)$	CDF of Put latency
α_{dp_j}	reservation price ratio	D	entire data set
d_l/s_{d_l}	data l and d_l 's size	$L^g(d_l)$	Get deadline to d_l
β	number of replicas	$L^p(d_l)$	Put deadline to d_l
$\epsilon^g(d_l)$	allowed % of Gets/Puts on d_l beyond deadlines	$v_{dc_i}^{d_l, t_k}$	Get/Put rates targeting d_l generated by dc_i in t_k
$\epsilon^p(d_l)$		$u_{dc_i}^{d_l, t_k}$	
T	reservation time	t_k	k^{th} billing period
Output	Description	Output	Description
C_t	total cost for storing D and serving requests	$X_{dp_j}^{d_l, t_k}$	existence of d_l 's replica in dp_j during t_k
$H_{dc_i, dp_j}^{d_l, t_k}$	whether dp_j serves requests on d_l from dc_i	$R_{dp_j}^g / R_{dp_j}^p$	optimal reserved number of Gets/Puts

2 PROBLEM STATEMENT

2.1 System Model

A customer deploys its application on one or multiple datacenters, which we call *customer datacenters*. The clients access the storage services directly and the broker is responsible for the price selection. We use D_c to denote the customer datacenters of all customers and use $dc_i \in D_c$ to denote the i^{th} customer datacenter. D_s denotes the set of the storage datacenters of all CSPs and $dp_j \in D_s$ 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 [13], 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 [14] 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. We also use table 1 to denote all the symbols in section 2, 3 and 4.

The capacity in the paper is the capacity of a data center or data centers in a region, which has the limit resources, especially the bandwidth. 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 [15].

2.2 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 $R_{dp_j}^g$ to denote the number of reserved Gets in dp_j and calculate the Get reservation benefit in dp_j ($f_{dp_j}^g(R_{dp_j}^g)$) by:

$$f_{dp_j}^g(R_{dp_j}^g) = \left(\sum_{t_k \in T} R_{dp_j}^g * (1 - \alpha_{dp_j}) - O_{dp_j}^g(R_{dp_j}^g) \right) * p_{dp_j}^g, \quad (1)$$

where $p_{dp_j}^g$ is the unit Get price, and $O_{dp_j}^g(R_{dp_j}^g)$ is the over reserved Get rates including the cost for over reservation and the over calculated saving and it is calculated by

$$O_{dp_j}^g(R_{dp_j}^g) = \sum_{t_k \in T} \text{Max}\{0, R_{dp_j}^g - \sum_{dc_i \in D_c} r_{dc_i, dp_j}^{t_k} * t_k\}, \quad (2)$$

where $r_{dc_i, dp_j}^{t_k}$ denotes the Get rate from dc_i to dp_j during t_k . We calculate the Put reservation benefit ($f_{dp_j}^p(R_{dp_j}^p)$) similarly.

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

$$C_{sum}^{c_n} = C_s * \gamma_s^{c_n} + C_t * \gamma_t^{c_n} + C_g * \gamma_g^{c_n} + C_p * \gamma_p^{c_n}, \quad (3)$$

where $\gamma_s^{c_n}$, $\gamma_t^{c_n}$, $\gamma_g^{c_n}$ and $\gamma_p^{c_n}$ 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 $F_{dc_i, dp_j}^g(x)$ and $F_{dc_i, dp_j}^p(x)$ to denote the cumulative distribution function (CDF) of Get and Put latency from dc_i to dp_j , respectively. Thus, $F_{dc_i, dp_j}^g(L^g(d_l))$ and $F_{dc_i, dp_j}^p(L^p(d_l))$ 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.,

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

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:

$S_{d_l}^p = \{dp_j | \forall dc_i \forall t_k, (u_{dc_i}^{d_l, t_k} > 0) \rightarrow (F_{dc_i, dp_j}^p(L^p(d_l)) \geq 1 - \epsilon^p(d_l))\}$ that satisfy Put SLO of d_l , where $u_{dc_i}^{d_l, t_k}$ denotes the Put rate targeting d_l from dc_i during t_k . The intersection of the two sets, $S_{d_l}^p \cap S_{dc_i, d_l}^g$, 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 $S_{d_l}^p \cap S_{dc_i, d_l}^g$ (**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:

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

A simple reduction from the generalized assignment problem [16] can be used to prove this problem is NP-hard.

3 THE DESIGN OF ES^3

3.1 Overview of ES^3

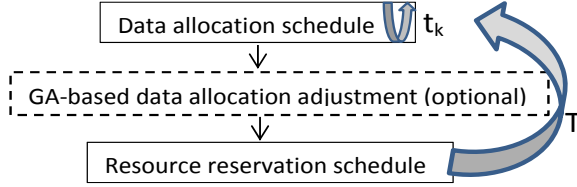


Fig. 2: Sequence of scheduling.

Due to the hardness of the above formulated problem, we propose a heuristic solution, called coordinated data allocation and reservation method (Section 3.2). 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 3.3) to improve the data allocation schedule before determining the resource reservation schedule.

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 4.1) is used whenever a Get request will be sent to a reservation-overutilized datacenter.

3.2 Coordinated Data Allocation and Resource Reservation

In Section 3.2.1, 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 3.2.2, we present how to schedule the data allocation by following this rule and leveraging all the pricing policies for cost minimization and SLO guarantee.

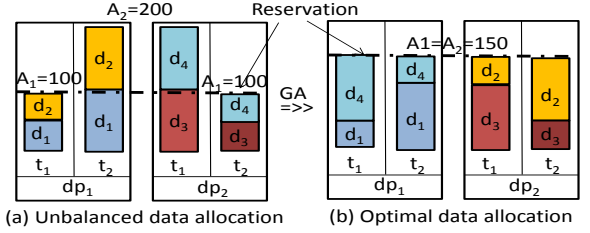


Fig. 3: Unbalanced and optimal data allocation.

3.2.1 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 $B_{dp_j} = \text{Max}\{f_{dp_j}^g(R_{dp_j}^g)\}_{R_{dp_j}^g \in \mathbb{N} \cup \{0\}}$ to denote the largest reservation benefit for dp_j given a specific data allocation. We use $A_{t_k} = \sum_{dc_i \in \mathcal{D}_c} r_{dc_i, dp_j}^{t_k} * t_k$ to denote the number of Gets served by dp_j during t_k , and define $\mathcal{A} = \{A_1, A_2, \dots, A_n\}$ 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.

We can prove that when $R_{dp_j}^g \in [A_i, A_{i+1}]$ ($i = 1, \dots, n-1$), reservation benefit $f_{dp_j}^g(R_{dp_j}^g)$ increases or decreases monotonically within $[A_i, A_{i+1}]$.

Theorem 1. For a datacenter dp_j , its reservation benefit function $f_{dp_j}(x)$ increases when $x \in [0, R_{dp_j}^g)$, and decreases when $x \in (R_{dp_j}^g, A_n]$, where $R_{dp_j}^g$ 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 $f_I(x) = f_{dp_j}(x) - f_{dp_j}(x-1) = (n * (1-\alpha) - O'(x)) * p_{dp_j}^g$, 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 $\sum_{dc_i \in \mathcal{D}_c} r_{dc_i, dp_j}^{t_k} < x$. 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, $f_{dp_j}^g(x)$ increases and then decreases. Since $f_{dp_j}^g(R_{dp_j}^g)$ reaches the largest $f(x)$, we can derive that $f_{dp_j}^g(x)$ increases when $x \in [0, R_{dp_j}^g)$, and decreases when $x \in (R_{dp_j}^g, A_n]$. \square

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 $A_{i+1} \leq R_{dp_j}^g$, then for $x \in [A_i, A_{i+1}]$, $f_{dp_j}(x)$ increases monotonically; otherwise $f_{dp_j}(x)$ decreases monotonically. This means that the reservation benefit reaches the maximum when $R_{dp_j}^g \in \mathcal{A}$. Thus, the optimal reservation is the A_i ($i \in [1, n-1]$) that generates the largest reservation benefit, i.e.,

$$B_{dp_j} = \text{Max}\{f_{dp_j}^g(A_i)\}_{A_i \in \mathcal{A}}. \quad (5)$$

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. For $x \in [0, A_1]$, we can transform Equation (1) to $f_{dp_j}(x) = \sum_{t_k \in T} x * (1 - \alpha) * p_{dp_j}^g$. Then, we can get that for $x \in [0, A_1]$, $f_{dp_j}(x)$ is positively proportional to x . Also, the maximum reservation benefit is no less than the reservation benefit of choosing $R_{dp_j}^g = \text{Min}\{A_i\}_{A_i \in \mathcal{A}} = A_1$. Therefore, in order to maximize reservation benefit on datacenter dp_j , we can enlarge its lower bound $f_{dp_j}^g(A_1)$, 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.

3.2.2 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 $C_s^{d_l}$, $C_g^{d_l}$, $C_t^{d_l}$ and $C_p^{d_l}$. 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 $C_s^{d_l}$ dominates the total cost (e.g., $C_s^{d_l} \gg C_g^{d_l} + C_p^{d_l}$), 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 [17]. 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.

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 2.2 indicates that datacenters in $(S_{d_l}^p \cap S_{dc_i, d_l}^g)$ 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 1. 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 3.2.1. 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} .

Algorithm 1: Data allocation scheduling algorithm.

```

1 for each  $dc_i$  in  $\mathcal{D}_c$  do
2   for each  $d_l$  requested by  $dc_i$  do
3     while the number of replicas of  $d_l$  is less than  $\beta$  do
4       if first replica of  $d_l$  then
5         It is assigned to serve requests from  $dc_i$ 
          towards  $d_l$ ; All other replicas do not serve Gets;
6       if  $d_l$  is Storage intensive then
7          $L = \{(dp_j \text{ with the largest } \mathbb{S}_{dp_j} \text{ among all}$ 
          datacenters having the smallest Storage unit
          price)  $\wedge (dp_j \in S_{d_l}^p \cap S_{dc_i, d_l}^g) \wedge (dp_j \text{ with}$ 
          enough Get/Put capacity)  $\}$ ;
8       else if  $d_l$  is Get/Put intensive then
9          $L = \{(dp_j \text{ with the smallest Get/Put unit price}$ 
           $\vee$  with the lowest unit reservation price  $\vee$  with
          the largest increment of  $A_1$ )
           $\wedge (dp_j \in S_{d_l}^p \cap S_{dc_i, d_l}^g) \wedge (dp_j \text{ with enough}$ 
          Get/Put capacity)  $\}$ ;
10      else if  $d_l$  is non-intensive then
11         $L$  is the union of all the above  $L$  sets;
12       $d_l$  is allocated to  $dp_j$  in  $L$  with the smallest  $C_{sum}$ ;

```

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.

3.3 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 $R_{dp_j}^g = 100$ and $R_{dp_j}^p = 200$ 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).

As shown in Figure 4, this method regards each data allocation schedule, represented by $\langle d_l, \{dp_1, \dots, dp_\beta\} \rangle$ ($d_l \in \mathcal{D}$), as a genome string, where $\{dp_1, \dots, dp_\beta\}$ (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.



Fig. 4: GA-based data allocation adjustment.

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 $1-\theta$). To ensure the schedule validity, for each crossover, the genomes that do not meet all constraints in Section 2.2 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_1} (which serves Gets) and a randomly selected dp_k in \mathbb{G}_{d_i} 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, $N_g = \text{Min}\{N, \frac{N}{C_{sum}/C'_{sum}}\}$, 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]).

4 COST EFFICIENT AND SLO GUARANTEE ENHANCEMENTS

4.1 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 $T_{min} = R_{dp_j}^g/t_k$ to check whether a datacenter is reservation-underutilized. The master calculates the aggregate 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. In order to satisfy SLO, we store the data in datacenters close to the client datacenter, therefore, we do not expect a remote request will always happen, and the transfer cost is still not a dominance.

4.2 Grouped Gets for Cost Minimization

To fetch all data objects of a webpage, many Get requests are generated; each Get fetching one data object. In cloud storage, each Get has a size limitation (denoted by u_g) such as the 4kB specified in Amazon DynamoDB [5]. For a Get g_i from a user, the actual number of Gets considered by the cloud provider in cost calculation is equal to $\lceil s_{g_i}/u_g \rceil$, where s_{g_i} is the requested data size of Get g_i . That is, if the Get size is larger than the size limitation, the Get is considered as multiple Gets by the cloud provider when deciding the charging amount.

Usually, one single data object is much smaller than the size limitation of a Get. For example, one single object in Facebook [10] has less than 1kB. Therefore, reading multiple concurrently requested data objects together through one single Get can save the Get cost. This way, instead of reading a single data object by one Get, the grouped data objects within an aggregated data item are read entirely through one Get. In this section, we use o_i to represent a grouped data object and it may only include one single data object.

As indicated in [18], data dependency exists among the data objects requested to present a webpage. Object o_i depends on object o_j means o_j must be fetched before fetching o_i . For example, in Facebook, when a user logs in, his/her friend list is fetched first and then their recent posts are fetched in order to get the user's News Feed. A dependency tree can be used to show the dependency among data objects in a data item, where a parent's children depend on the parent, i.e., the parent data objects must be queried before the child data objects. Therefore, intuitively, we can group a data object and all of its predecessors together to save Get cost.

However, a parent data object's children may not need to be read together. Therefore, we still need to resolve how to cluster data objects in a data item into data groups so that each group has a high probability to be read together by a Get from customers. To handle this problem, we propose a coefficient-based data grouping method to aggregate the data objects together to reduce the number of Gets. We define the coefficient between two data objects as the probability that they are requested together by Gets. We use p_{o_i, o_j} to denote the coefficient of data objects o_i and o_j . o_i and o_j can be either a single data object or a grouped data object. We assume the probability that two data objects requested together is the same for Gets from different customer datacenters because they serve the same website. We also assume the data objects' coefficient does not change over time, unless the customer changes its website. In this case, all grouped data objects are broken into single data objects and they are grouped again by our grouping method.

Before we present our coefficient-based data grouping method, we first introduce how to calculate coefficient p_{o_i, o_j} . Each grouped data object o_i has a precedent data object. If o_i is a single data object, its precedent object is o_i itself; otherwise, its precedent object is the data object in o_i at the highest level of the dependency tree. To derive p_{o_i, o_j} , we first find the precedent data object in each grouped data object (o_i and o_j). When we form a grouped data object, we need to ensure that all data objects are directly or indirectly dependent on the precedent data object of the newly formed grouped data object. Due to the dependency, the read rate of a grouped data object is the read rate of its precedent data object. If the precedent object of one grouped data object o_i depends on the precedent object in the other data object o_j , then at each time when o_i is read, o_j must be already read. Hence, we can derive $p_{o_i, o_j} = v_{o_i} / (v_{o_i} + v_{o_j})$, where v_{o_i} and v_{o_j} is the read rate of grouped data object o_i and o_j from all customer datacenters during T . If the precedent objects of o_i and o_j have no dependency on each other, p_{o_i, o_j} is set to negative infinite.

If o_i and o_j forms a grouped data object, a Get request initially for o_i , o_j or both will be a Get request for this grouped data object. To decide whether o_i and o_j should be combined to a grouped data object, we need to check whether this combination saves the Get cost, that is, whether reading the combined object entirely has a smaller number of Gets than reading each single data object inside it individually based on the read rates of o_i and o_j . Thus, we first calculate the Get cost for reading individually as $C_{o_i, o_j}^{ind} = \lceil s_{o_i} / u_g \rceil * v_{o_i} + \lceil s_{o_j} / u_g \rceil * v_{o_j}$. We then calculate the cost of reading them together as $C_{o_i, o_j}^{grp} = \lceil (s_{o_i} + s_{o_j}) / u_g \rceil *$

$(v_{o_i} + v_{o_j}) * (1 - p_{o_i, o_j}) = \lceil (s_{o_i} + s_{o_j}) / u_g \rceil * v_{o_j}$ by timing the number of Gets of the entire grouped data object and the read rate together. We can then calculate the benefit of grouping two data objects as $B_{o_i, o_j} = C_{o_i, o_j}^{ind} - C_{o_i, o_j}^{grp}$. If $B_{o_i, o_j} > 0$, o_i and o_j can form a grouped data object.

Algorithm 2: Coefficient-based data grouping algorithm.

Input: List L with all data objects in an aggregated data item
Output: List L' with all grouped data objects

- 1 Sort data objects in list L in descending order of their levels in the dependency tree of the data item;
- 2 **for each** o_i **in list** L **do**
- 3 Find $o_j \in L'$ with the largest grouping benefit with o_i , B_{o_i, o_j} ;
- 4 **if** $B_{o_i, o_j} > 0$ **then**
- 5 o_i is grouped into o_j ;
- 6 **else**
- 7 o_i is inserted into list L' ;

The detailed procedure of coefficient-based data grouping method is shown in Algorithm 2. To group data objects, we sort all single data objects in the descending order of their levels in the dependency tree into a list L (Line 1). We loop all data objects to combine each object into an existing grouped data object or form an individual grouped data object (Lines 2-7). For each data object $o_i \in L$ (Line 2), we loop each of all data objects inside another list L' , which initially is empty, and calculate the grouping benefit. For the data object o_j with the largest grouping benefit B_{o_i, o_j} (Line 3), if $B_{o_i, o_j} > 0$, we group o_i into grouped data object o_j (Lines 4-5); otherwise, we directly insert o_i into L' as a grouped data object with a single object (Lines 6-7). After looping all data objects inside L , L' includes the grouped data objects that can save Get cost. For newly added data objects, we first insert them into L and insert all current grouped data objects into L' , and then each new data object is grouped into an existing grouped data object or form a new grouped data object according to the procedure in Lines 2-7. This algorithm is conducted before the real data allocation conduction, so that the objects in a grouped data object are stored as a file unit for Get/Puts.

4.3 Lazy Update for Cost-Efficient Puts

4.3.1 Put Aggregation

Eventual consistency means that if no new updates are made to a given data item, eventually all accesses to that data item will return the last updated value. Each Put of a data item needs to be propagated to all of its replicas for consistency maintenance. We notice that for eventual consistency, the propagation of updates on rarely used replicas can be postponed, which can be leveraged to save Put cost. For example, adding an advertisement to a webpage only needs eventual consistency and it does not need an instant update. Similar to reading a grouped data object, we can aggregate sequential writes into one Put to propagate to all rarely used replicas. Recall that for data item d_l of customer datacenter dc_i , a storage datacenter dp_j storing d_l always serves Gets from dc_i targeting d_l and β replicas of d_l are stored in other storage datacenters for data availability. We

call d_l in datacenter dp_j the *master replica* of data d_l for customer datacenter dc_i and call other replicas *slave replicas* of data d_l for dc_i .

The write to data item d_l from customer datacenter dc_i always triggers a Put to its master replica, which serves Gets from dc_i targeting d_l , so that the customers served by dc_i can always see the updates. Since the slave replicas do not usually serve the Get requests, we can postpone their updates in order to save Put cost. Thus, customer datacenter dc_i caches the recent writes on d_l , and combines the writes before sending them to slave replicas later on. The TTL (time-to-live) timeout is the longest time the cached writes can be delayed. The combined writes of an object will be sent out to all slave replicas after a TTL timeout or whenever the customer datacenter's cache (used for the Put aggregation purpose) is full. In the case that the cache is full, we adopt a Least Recently Combined First Out (LRCFO) strategy to select the combined writes of top data items to send out to relieve cache space. This strategy assumes that for a data item, if there are no updates for a longer time, it has lower probability to be updated. Thus, for recently updated data items, we expect more writes to be combined to further save Put cost.

4.3.2 Replica Deactivation

Customers may need to maintain strong consistency for some data items, in which all Puts are seen in the same order (sequentially) by different distributed clients at the same time. For both strong and eventual consistencies, in order to save Put cost, we can dynamically deactivate (i.e., remove) the slave replicas of a data item if it receives few Get requests [19] for a certain time period, so that they do not need to receive updates. For example, for a customer datacenter serving online social networks, its service has a diurnal pattern [20], that is, its workload dramatically drops down at night. By deactivating some slave replicas of the data item serving few requests in the inactive time period, the Put cost can be further reduced. For some high valuable data, they can be marked by the end-user directly so that these data will not be deactivated in the period.

Recall that slave replicas of customer datacenter dc_i usually do not serve its Get requests and they are created mainly to increase the data availability. The slave replicas introduce Storage cost and Put cost. Only when the Get workload from dc_i is high and the storage datacenters of master replicas cannot provide SLO-guaranteed service, the Get requests will be forwarded to datacenters hosting the slave replicas of the requested data. Therefore, when the request rate of Gets from dc_i towards data item d_l drops below a threshold T_r (i.e., when the slave replicas are unlikely to be used), in order to save Put cost on the slave replicas, we can deactivate the slave replicas of d_l from storage datacenters. When the request rate of Gets towards d_l from dc_i increases beyond T_r , the slave replicas are dynamically created by transferring the updated replicas of d_l from the datacenters containing them.

Below, we introduce how to choose the slave replicas of d_l to deactivate in order to save total cost. Specifically, we predict the benefit of deactivating a slave replica to decide the necessity of its deactivation. For reactivation, the cost consumption includes the Transfer cost, which is

$C'_t = p^t(dp_j) * s_{d_l}$, where dp_j is a slave replica datacenter. The Storage and Put costs are saved during the deactivation. If the reserved Puts of slave replica datacenter dp_j has not been fully utilized, since the payment cost for Put reservation is already determined and cannot be further saved, the Put cost $C'_p = 0$; otherwise, it is calculated as $C'_p = w'_{dc_i,dp_j} * p^p_{dp_j} * t'$, where t' is the deactivation time period. The saved storage cost can be calculated as $C'_s = s_{d_l} * p^s_{dp_j} (S_{dp_j}) * t'$. Finally, we can derive the deactivation benefit of a slave replica as $C'_s + C'_p - C'_t$. If the deactivation benefit is larger than zero, this slave replica is deactivated; otherwise, it will not be deactivated.

The storage savings may be much smaller compared to transfer cost in the deactivation if the transferring happens between different CSPs, which has a higher unit Transfer price. For example, if a data item has 1GB and a data object that needs deactivation to save Put cost has 1kB, then transferring 1GB to reactivate the data item may generate cost higher than the sum of the costs for transferring only 1kB between different CSPs and storing (1GB-1kB). Thus, we do not need to remove the slave replica of the entire data item. Instead, we remove the data objects that receive Puts during the deactivation period. When a data item is identified to be deactivated, it is marked. Then, during the deactivation time, once there is a Put on an object, this object is removed. This way, we can further save the total cost by only transferring the updated data objects inside the data item in the activation to further reduce the transfer cost.

4.4 Concurrent Requests for Rigid Get SLO Guarantee

Within each billing period, the data allocation of a customer is stable. However, the customer may require a more rigid Get SLO (low tail latency SLO) during this billing period with a smaller $\epsilon(d_l)$ or $L^g_{d_l}$. If the Get SLO of d_l is too rigid for the storage datacenter of the main replica to handle, we can concurrently submit multiple Get requests to different replicas including the master and slave replicas. This way, although some of the datacenters cannot supply a Get SLO guarantee service, there can be a datacenter among them responding the request with the rigid SLO guarantee. Though this method introduces additional Get cost due to more Get requests, it avoids the need to conduct data reallocation again, so that it saves the replica Transfer cost and does not waste the reserved Get/Put cost for datacenters currently storing d_l .

Intuitively, if we transmit a Get request targeting data d_l to all β datacenters with its replica, we can get a low response latency with a high probability. However, it may introduce unnecessary Get costs, since a combination of part of the datacenters may already supply a Get SLO guaranteed service. The problem to find such a combination with Get SLO guarantee and Get cost minimization can be easily reduced to the knapsack problem, which is NP-hard [16]. Since β is usually small, we can enumerate all combinations that satisfy the rigid Get SLO and find the one with the minimum cost. To efficiently find the combination, we introduce a greedy heuristic algorithm. Unlike the master replicas of d_l , the Gets towards its slave replicas are not considered in deciding the Get reservation of their datacenters. Then, the Get cost is calculated based on the pay-as-you-go policy.

Thus, to minimize the additional Get cost introduced by concurrent requests, we sort all slave replica datacenters of d_l in ascending order of the Get unit cost. Then, we sequentially check each slave replica datacenter dp_j in the list. If the additional Get workload on dp_j does not make its total Get workload exceed its Get capacity, we add dp_j into the combination \mathbb{C} . This process continues until the combination can satisfy the rigid Get SLO guarantee, that is,

$$\prod_{dp_j \in \mathbb{C}} (1 - F_{dc_i, dp_j}^g(L^g(d_l))) \leq \epsilon^g(d_l). \quad (6)$$

Therefore, the probability that all storage datacenters cannot respond a Get within the deadline, $\prod_{dp_j \in \mathbb{C}} (1 - F_{dc_i, dp_j}^g(L^g(d_l)))$, is no larger than the allowed percentage $\epsilon^g(d_l)$, so that the rigid SLO is satisfied.

5 PERFORMANCE EVALUATION

We conducted trace-driven experiments on Clemson University’s Palmetto Cluster [21], which has 771 8-core nodes, and on real-world clouds with a real deployment of ES^3 . We first introduce the experimental settings.

Simulated clouds. The services we are simulating is a broker serving multiple web application using key-value data storage model. 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 pricing model in the simulation refers to DynamoDB. 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 [22]. 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 [23] and we set the Get rate of each data item based on the 100:1 Get:Put ratio [18]. 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 [18]. 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.

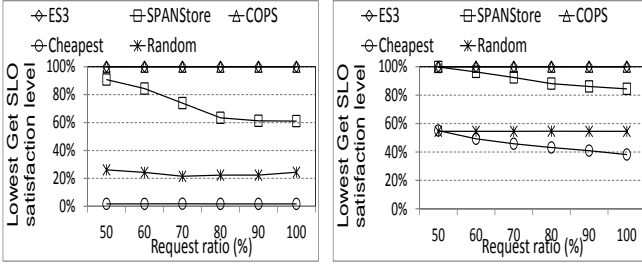
Comparison methods. We compared ES^3 with the following systems. i) *COPS* [9]. It allocates requested data into a datacenter with the shortest latency to each customer datacenter but does not consider payment cost minimization. ii) *Cheapest*. It selects the datacenters with the cheapest cost in the 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 key-value storage system over multiple CSPs’ datacenters to minimize cost and guarantee SLOs. It does not consider the data-center capacity limitation, which may lead to SLO violation. On the other hand, it does not fully leverage all pricing policies in cost minimization as indicated previously. Furthermore, it does not consider Get/Put rate variation during a billing period.

5.1 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 [23], with a step increase of 10%. The Get SLO satisfaction level of a customer is calculated by $\text{Min}\{\text{Min}\{n'_{t_k}/n_{t_k}\}_{\forall t_k \in T}, (1 - \epsilon^g)\}/(1 - \epsilon^g)$, 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. 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. *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.

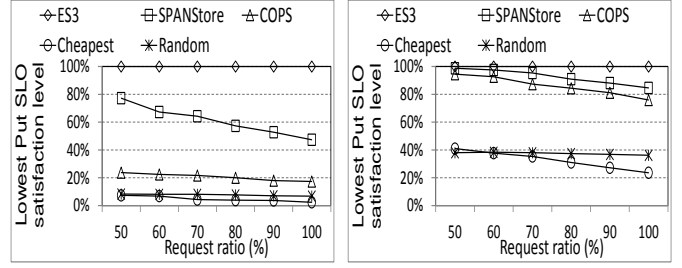
Figures 7(a) and 7(b) show the percentage of Gets received by overloaded datacenters in simulation and



(a) In simulation

(b) In real clouds

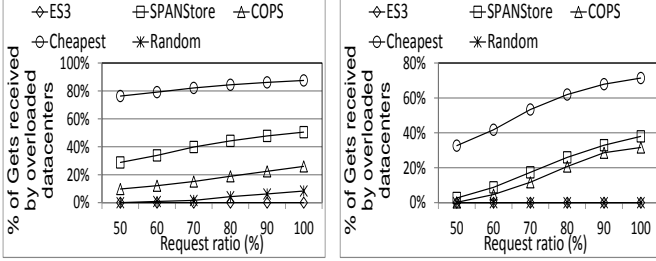
Fig. 5: Get SLO guaranteed performance.



(a) In simulation

(b) In real clouds

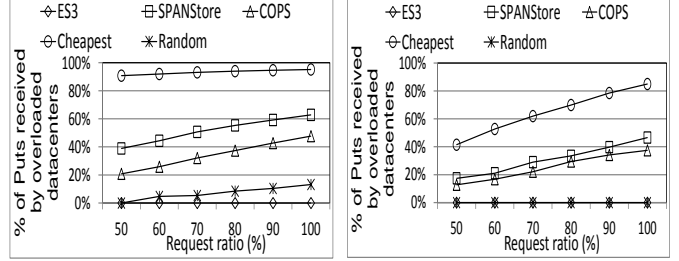
Fig. 6: Put SLO guaranteed performance.



(a) In simulation

(b) In real clouds

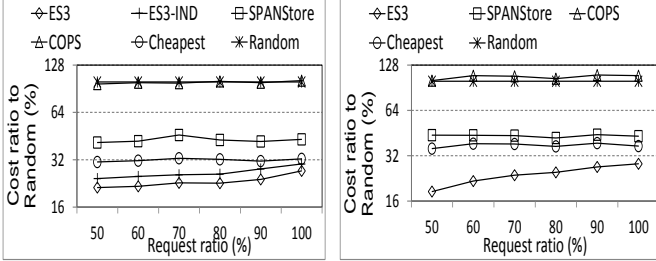
Fig. 7: Percent of Gets received by overloaded datacenters.



(a) In simulation

(b) In real clouds

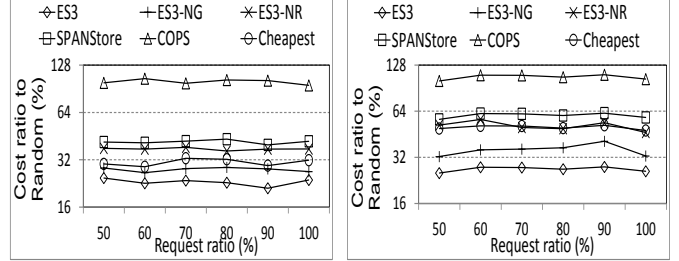
Fig. 8: Percent of Puts received by overloaded datacenters.



(a) In simulation

(b) In real clouds

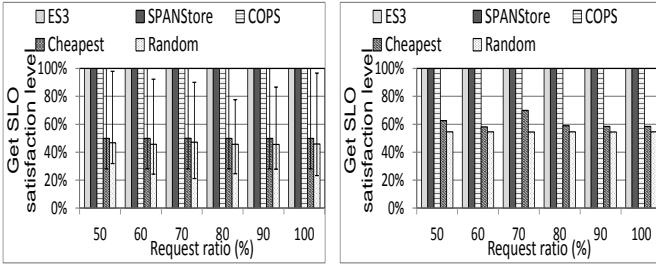
Fig. 9: Payment cost minimization with normal Get/Put workload.



(a) In simulation

(b) In real clouds

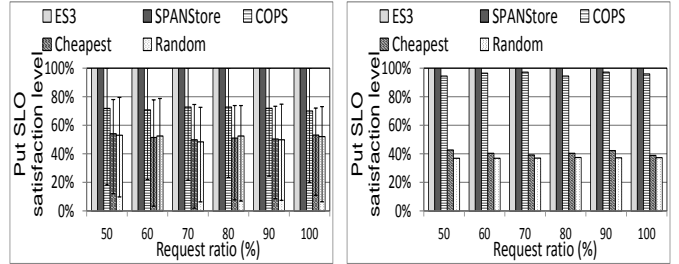
Fig. 10: Payment cost minimization with light Get/Put workload.



(a) In simulation

(b) In real clouds

Fig. 11: Get SLO guaranteed performance with light Get/Put workload.



(a) In simulation

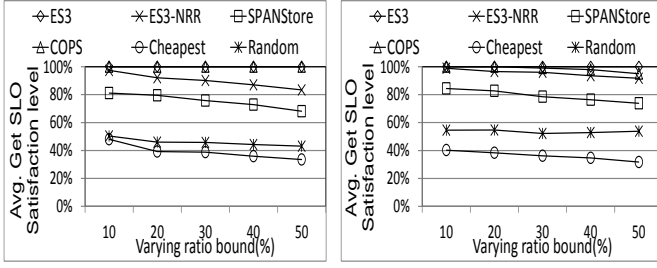
(b) In real clouds

Fig. 12: Put SLO guaranteed performance with light Get/Put workload.

real-world experiment, respectively. Due to the capacity-awareness, ES^3 can avoid the datacenter overloads. *Random* allocates data items over all storage datacenters randomly, so it has a smaller probability of overloading storage datacenters. The other methods make datacenters overloaded, and show opposite orders from Figure 5(a) due to the same reasons. Figures 8(a) and 8(b) show the percentage of Puts received by overloaded datacenters. Figures 7 and 8 indicate that ES^3 outperforms other systems in that it can effectively avoid overloading datacenters by capacity-aware data allocation, which helps ensure the Get/Put SLOs.

Since *Random* does not consider SLO guarantee or payment cost minimization, we measure the cost improvement of the other systems compared to *Random*.

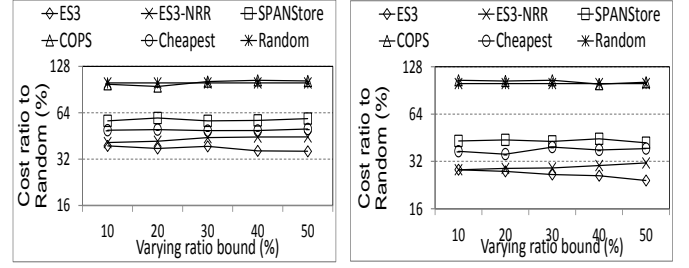
Figures 9(a) and 9(b) show the ratio of each system's cost to *Random*'s cost in simulation and real-world experiment, respectively. In order to show the effect of considering the tiered pricing model, in simulation, we also tested a variant of ES^3 , denoted by ES^3-IND , in which each customer individually uses ES^3 to allocate its data without aggregating their workload together through the broker. 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



(a) In simulation

(b) In real clouds

Fig. 13: SLO guarantee of Gets with varying Get rate.



(a) In simulation

(b) In real clouds

Fig. 14: Cost minimization with varying Get rate.

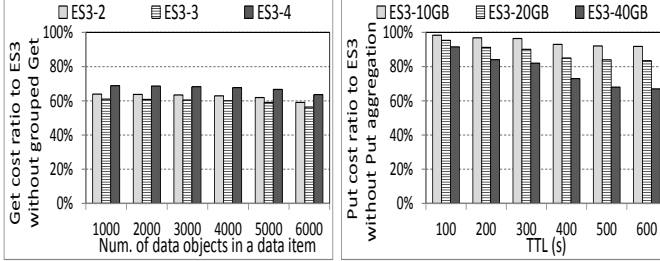


Fig. 15: Cost reduction by grouped Gets.

Fig. 16: Cost reduction by Put aggregation.

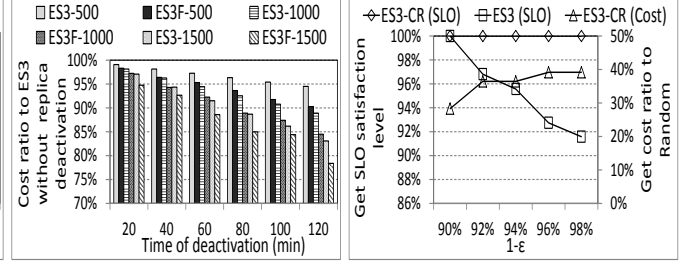


Fig. 17: Cost reduction by replica deactivation.

Fig. 18: Performance of concurrent requests.

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.

5.2 Effectiveness of Individual Methods in ES^3

We are interested to see whether the data intensiveness change will affect the performance of different systems. 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 10(a) and 10(b) show the ratio of each system's cost to *Random*'s cost in simulation and real-world experiment, respectively. 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.

Figure 11(a) shows the median, 5th and 95th percentile of all customers' Get SLO satisfaction levels of each system with each request ratio in simulation. Figure 11(b) shows the Get SLO satisfaction level of the customer of each system

in real-world experiment. They show that ES^3 and *COPS* can supply a Get SLO ensured service due to the same reasons as in Figure 5(a). *SPANStore* also supplies a Get SLO guaranteed service, due to its SLO awareness and the light workload that does not overload datacenters. *Random* and *Cheapest* do not consider the SLO, thus their Get SLO satisfaction levels are much lower.

Figure 12(a) shows the median, 5th and 95th percentile of all customers' Put SLO satisfaction levels of each system with each request ratio. Figure 12(b) shows the Put SLO satisfaction level of the customer of each system with each request ratio in real-world experiment. They show a similar order of all systems as in Figure 6(a) due to the same reasons. Different from Figure 6(a), in Figure 12(a), *SPANStore* can supply an SLO guaranteed service, and *Random* and *Cheapest* achieve similar performances due to the same reasons as in Figure 11(a).

5.3 Performance of Enhancements

In this section, we present the performance of the cost efficient and SLO guarantee performance of the enhancement methods in Section 4 in real clouds.

5.3.1 Dynamic Request Redirection

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 13(a) and 13(b) show the average Get 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 14(a) and 14(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 9(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.

5.3.2 Grouped Gets for Cost Reduction

Next, we measured the performance of the effectiveness of the grouped Get method for Get cost saving. We set the size limitation of each unit Get as 400kB, since the size of Get unit is 4 times as large as the size of average data object in Facebook [5], [18], and the default size of a data object was set to 100kB. We varied the number of data objects in a data item from 1000 to 6000 with a step size as 1000. All data objects form a N -ary dependency tree [18], with N increasing from 2 to 4 with a step size as 1. The dependency between data objects in one data item is randomly generated. We used the default setting of Get rates for the data objects that are leaf nodes of the N -ary tree, and then the Get rate of a parent node is the sum of the Get rates of all of its children. We used ES^3 - N to denote ES^3 with the grouped Get method.

Figure 15 shows the Get cost ratio of ES^3 - N calculated by the Get cost of ES^3 with the grouped Get method over the Get cost of ES^3 without this method. It shows that the Get cost ratio of ES^3 - N is from 56.4% to 68.9%. This is because by aggregating Gets for data objects concurrently requested into one unit Get, it reduces the number of Gets, so that the Get cost is reduced. The figure also shows that with more data objects within a data item, the Get cost ratio does not increase, which means the performance of the grouped Get method is scalable when the number of the requested data objects increases. We can also see that ES^3 -3 achieves the smallest cost ratio. Due to the 100kB data object size and 400kB Get limitation, one Get can fetch four objects together. Then, according to Algorithm 2, in ES^3 -3, a data object will be grouped with all of its child data objects, so that the Get cost for all the children can be saved by requesting the parent data object with them together. The figure indicates that the group Get method can effectively save the Get cost.

5.3.3 Lazy Update for Cost-Efficient Puts

We then measured the performance of cost reduction by the put aggregation method. We use ES^3 - x GB to denote ES^3 with the Put aggregation method with a cache size as x GB. Figure 16 shows the Put cost ratio calculated by the Put cost of ES^3 with the Put aggregation method over the Put cost of ES^3 without this method. This is because the Put aggregation method combines consecutive Puts together to save the number of Puts, leading to lower Put cost. It also shows that the cost ratio decreases when the TTL or the cache size increases. A longer TTL allows data objects with lower Put rates to aggregate multiple Puts together to be propagated in order to save Put cost. Similarly, a larger cache size also allows more data objects to aggregate their Puts, leading to lower cost. The result indicates that the Put aggregation can effectively save the Put cost. Since a larger TTL leads to a longer inconsistency period but a smaller payment cost, in reality, we need to set the TTL according to both the consistency and the payment cost minimization requirement to break the tie. Besides TTL, a larger cache also leads to a lower payment cost. However, it brings a larger capital investigation cost of the customer datacenter. Therefore, in reality, a customer of ES^3 can choose an optimal cache size according to its payment cost to CSPs and its own capital investment (or payment to CSP) on cache to break the tie.

Figure 17 shows the cost ratio of ES^3 with the replica deactivation (or the fine granularity replica deactivation) compared to ES^3 without it. Since the cost does not involve Get cost, we only measure the cost including Put, Storage and Transfer. It shows that the cost ratio of ES^3 is varied from 83% to 99%. This is because during the deactivation, the Put and Storage cost can be saved by removing the slave replicas temporarily. It also shows that the ES^3 - F can reduce more cost than ES^3 by removing the data objects that received updates instead of removing a whole data item in order to save the Transfer cost. Compared to ES^3 , ES^3 - F further reduced up to 4.7% total cost. From the figure we can also see that the cost ratio decreases when the deactivation time or T_r increases. With a longer deactivation time, more Puts can be skipped since there is shorter time for storing the data item, so that the Storage and Put costs are reduced. Similarly, a larger threshold leads to more replicas to be deactivated, which saves more cost. The figure indicates that the replica deactivation can save the payment cost, and the fine granularity replica deactivation can save more payment cost. A larger T_r may lead to more data replica deactivations, and the master replica itself needs to serve a larger workload. However, due to the deactivation of slave replicas, the data availability becomes more important. In reality, ES^3 can choose an optimal T_r considering both the data availability and cost minimization requirements.

5.3.4 Concurrent Requests for Rigid Get SLO Guarantee

We finally measured the SLO guarantee performance of the concurrent request method. We use ES^3 - CR (SLO) and ES^3 (SLO) to denote the Get SLO performance of ES^3 with and without the concurrent request method, respectively. In order to measure the performance under a highly rigid Get SLO, we need more replicas to send out concurrent

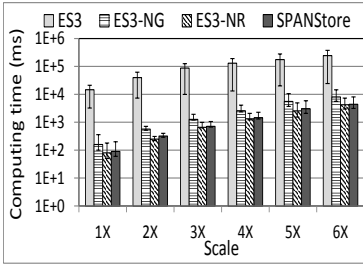


Fig. 19: Schedule computing time.

Gets. Therefore, in this experiment, we changed the default setting for the number of replicas per data item $\beta = 2$ to $\beta = 3$. We varied the minimum percentage of requests responded within deadlines $(1 - \epsilon)$ from 90% to 98% with a step size of 2%. Figure 18 shows the Get SLO satisfaction level of ES^3 with and without the concurrent request method versus $1 - \epsilon$. It shows that when $1 - \epsilon$ is larger than the default setting 90%, ES^3 -CR can still guarantee the SLO but ES^3 cannot. This is because by concurrently sending multiple requests to different replicas, it produces a higher probability that the fastest response is within the specified deadline in the SLO.

Recall that our concurrent request method uses the minimum Get unit cost first in selecting slave replicas to send requests until Formula (6) is satisfied. We measured the Get cost ratio calculated by the Get cost of our method over the Get cost of another concurrent request method that randomly selects slave replicas to send requests until Formula (6) is satisfied (denoted by *Random*). The figure shows that the minimum Get unit cost first selection can save at least 60% of the cost generated by *Random* due to the Get unit cost aware datacenter selection. A more rigid SLO needs more concurrent Gets. The additional selected datacenters have higher Get unit prices than those with lower $(1 - \epsilon)$ in our method, but have comparable Get unit prices as those with lower $(1 - \epsilon)$ in *Random*, which leads to lower Get cost savings compared to *Random*. Therefore, more concurrent Gets lead to a higher Get cost ratio to *Random*. The figure indicates that the concurrent request method can effectively guarantee a more rigid temporal Get SLO and meanwhile supply a cost-efficient service compared to random replica datacenter selection.

5.4 System Overhead

In this experiment, we measure the overhead of all systems with SLO guaranteed service. We use ES^3 -NG to denote ES^3 without the GA based data allocation adjustment approach. We enlarge the number of datacenters and the number of customers by x times, which was varied from 1 to 5, with a step size as 1. Figure 19 shows the median, 5th and 95th percentile of computing time of the data allocation schedule calculation in different systems. ES^3 depends on the GA based algorithm to enhance the cost minimization which is time consuming. Thus, it generates the largest computing time. ES^3 -NG does not have this algorithm, but compared to *SPANStore*, it needs to calculate the reservation, which leads to a slightly higher computing time than *SPANStore*. Even though GA based algorithm leads to the largest computing time, according to the discussion in Section 3.3, it needs to be conducted only at the initial of T to determine the reservation.

6 RELATED WORK

Storage services over multiple clouds. RACS [24] and DepSky [25] 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 [26], [6], a cluster storage configuration automation method is proposed to use the minimum resource to support the desired workload. We also share the similarities with [27] by using redundant requests to reduce tail latency. But they focus in one datacenter, we focus on across datacenters.

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.

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.

7 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 also incorporates several methods to enhance its cost efficient and SLO guarantee performance. 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. The Transfer cost has a tiered pricing model and becomes more complex, and different CSP provide different unit prices from a source storage datacenter to other datacenters belonging to different CSPs or at different locations. In our future work, we will study the cost minimization problem of transferring replicas of data items to different storage datacenters whenever a new data allocation schedule is generated.

ACKNOWLEDGEMENTS

This research was supported in part by U.S. NSF grants IIS-1354123, CNS-1254006, CNS-1249603, CNS-1049947, CNS-0917056 and CNS-1025652, Microsoft Research Faculty Fellowship 8300751.

REFERENCES

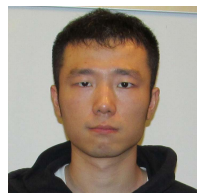
- [1] Amazon S3. <http://aws.amazon.com/s3/>.
- [2] Microsoft Azure. <http://www.windowsazure.com/>.
- [3] Google Cloud storage. <https://cloud.google.com/storage/>.
- [4] D. Niu, C. Feng, and B. Li. A Theory of Cloud Bandwidth Pricing for Video-on-Demand Providers. In *Proc. of INFOCOM*, 2012.
- [5] 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] S. Alan, K. Srikanth, G. Albert G, K. Changhoon, and S. Bikas. Sharing the data center network. In *Proc. of NSDI*, 2011.
- [14] 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.
- [15] G. Liu, H. Shen, and H. Chandler. Selective Data Replication for Online Social Networks with Distributed Datacenters. In *Proc. of ICNP*, 2013.
- [16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- [17] 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.
- [18] 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.
- [19] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proc. of EuroSys*, 2011.
- [20] S. Traverso, K. Huguenin, I. Trestian, V. Erramilli, N. Laoutaris, and K. Papagiannaki. TailGate: Handling Long-Tail Content with a Little Help from Friends. 2012.
- [21] Palmetto Cluster. <http://citi.clemson.edu/palmetto/>.
- [22] 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>.
- [23] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the Evolution of User Interaction in Facebook. In *Proc. of WOSN*, 2009.
- [24] A. Hussam, P. Lonnie, and W. Hakim. RACS: A Case for Cloud Storage Diversity. In *Proc. of SoCC*, 2010.
- [25] A. N. Bessani, M. Correia, B. Quaresma, F. Andr, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *TOS*, 2013.
- [26] 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.
- [27] W. Zhe, Y. Curtis, and M. Harsha V. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proc. of NSDI*, 2015.
- [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.



Haiying Shen received the BS degree in Computer Science and Engineering from Tongji University, China in 2000, and the MS and Ph.D. degrees in Computer Engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Associate Professor in Department of Computer Science at University of Virginia. Her research interests include distributed computer systems and computer networks with an emphasis on P2P and content delivery networks, mobile computing, wireless sensor networks, and grid and cloud computing. She was the Program Co-Chair for a number of international conferences and member of the Program Committees of many leading conferences. She is a Microsoft Faculty Fellow of 2010, a senior member of the IEEE and a member of the ACM.



Guoxin Liu received the BS degree in BeiHang University 2006, and the MS degree in Institute of Software, Chinese Academy of Sciences 2009. He is currently a Ph.D. student in the Department of Electrical and Computer Engineering of Clemson University. His research interests include distributed networks, with an emphasis on Peer-to-Peer, data center and online social networks. He is a student member of IEEE.



Haoyu Wang received the BS degree in University of Science & Technology of China, and the MS degree in Columbia University in the city of New York. He is currently a Ph.D student in the Department of Computer Science in University of Virginia. His research interests include data center, cloud and distributed networks.