Minimum-cost Cloud Storage Service Across Multiple Cloud Providers

Guoxin Liu, Haiying Shen*, *Senior Member IEEE*

Abstract—Many Cloud Service Providers (CSPs) provide data storage services with datacenters distributed worldwide. These datacenters provide different Get/Put latencies and unit prices for resource utilization and reservation. Thus, when selecting different CSPs' datacenters, cloud customers of globally distributed applications (e.g., online social networks) face two challenges: i) how to allocate data to worldwide datacenters to satisfy application SLO (service level objective) requirements including both data retrieval latency and availability, and ii) how to allocate data and reserve resources in datacenters belonging to different CSPs to minimize the payment cost. To handle these challenges, we first model the cost minimization problem under SLO constraints using the integer programming. Due to its NP-hardness, we then introduce our heuristic solution, including a dominant-cost based data allocation algorithm and an optimal resource reservation algorithm. We further propose three enhancement methods to reduce the payment cost and service latency: i) coefficient based data reallocation, ii) multicast based data transferring, and iii) request redirection based congestion control. We finally introduce an infrastructure to enable the conduction of the algorithms. 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 effectiveness of our algorithms for SLO guaranteed services and customer cost minimization.

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

1 Introduction

Cloud storage, such as Amazon S3 [1], Microsoft Azure [2] and Google Cloud Storage [3], has become a popular commercial service. A cloud service provider (CSP) provides data storage service (including Get and Put functions) using its worldwide geographically distributed datacenters. Nowadays, more and more enterprisers shift their data workloads to the cloud storage in order to save capital expenditures to build and maintain the hardware infrastructures and avoid the complexity of managing the datacenters [4].

Cloud storage service is used by many web applications, such as online social networks and web portals, to provide services to clients all over the world. In the web applications, data access delay and availability are critical, which affect cloud customers' incomes. Experiments at the Amazon portal [5] demonstrated that a small increase of 100ms in webpage load time significantly reduces user satisfaction, and degrades sales by one percent. For a request of data retrieval in the web presentation process, the typical latency budget inside a storage system is only 50-100ms [6]. In order to reduce data access latency, the data requested by clients needs to be handled by datacenters near the clients, which requires worldwide distribution of data replicas. Also, data replication between datacenters enhances data availability since it avoids a high risk of service failures due to datacenter failure, which may be caused by disasters or power shortages.

However, a single CSP may not have datacenters in all locations needed by a worldwide web application. Besides, using a single CSP may introduce a data storage vendor lock-in problem [7], in which a customer may not

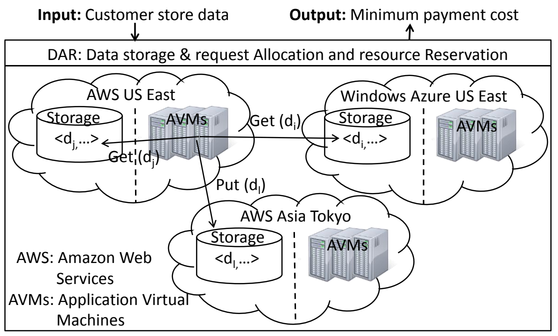


Fig. 1: An example of cloud storage across multiple providers.

be free to switch to the optimal vendor due to prohibitively high switching costs. Storage providers charge clients for bandwidth (Transfer), data requests (Get/Put), and Storage. Thus, a client moving from one CSP to another pays for Transfer cost twice, in addition to the Storage cost. The clients are vulnerable to price hikes by vendors, and will not be able to freely move to new and better options. The quickly evolving cloud storage marketplace may leave a customer trapped with an obsolete provider later. The vendor lock-in problem can be addressed by allocating data to datacenters belonging to different CSPs. Building such a geo-distributed cloud storage is faced with a challenge: *how to allocate data to worldwide datacenters to satisfy application SLO (service level objective) requirements including both data retrieval latency and availability?* The *data allocation* in this paper means the allocation of both data storage and Get requests to datacenters.

The payment cost of a cloud storage service consists of the costs for Storage, data Gets/Puts and Transfers [8]. Different datacenters of a CSP or different CSPs offer different prices for Storage, data Gets/Puts and Transfers. For example, in Figure 1, Amazon S3 provides cheaper data storage price ($0.01/GB and $0.005/1,000 requests), and Windows Azure in the US East region provides cheaper data Get/Put price ($0.024/GB and $0.005/100,000 requests). An application running on Amazon EC2 in the US East region might have data d_j with a large storage size and few Gets and data

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

- *Guoxin Liu is with the Department of Electrical and Computer Engineering, Clemson University, Clemson, SC, 29634. E-mail: guoxinl@clemson.edu Haiying Shen is with the Department of Computer Science, University of Virginia, Charlottesville, VA, 22904. E-mail: hs6ms@virginia.edu*

d_i which is read-intensive. Then, to reduce the total payment cost, the application should store data d_j in Amazon S3, and store data d_i into Windows Azure in the US East region. Thus, for a large-scale application with a large number of users, it is critical to have an optimal data allocation among datacenters to achieve the maximum cost saving. Besides the different prices, the pricing model is even more complicated due to two charging formats: pay-as-you-go and reservation. In the reservation manner, a customer specifies and prepays the number of Puts/Gets per reservation period T (e.g., one year) and is offered with cheaper unit price for the reserved usage than the unit price of the pay-as-you-go manner [8]. Then, the second challenge is introduced: *how to allocate data to datacenters belonging to different CSPs and make resource reservation to minimize the service payment cost?*

Since website visit frequency varies over time [9], unexpected events may introduce a burst of requests in a short time. It may affect the accuracy of predicting the visit frequency. Thus, we need to dynamically adjust the datacenter Get serving rate to save the payment cost.

In summary, to use a cloud storage service, a customer needs to determine data allocation and resource reservation among datacenters worldwide belonging to different CSPs to satisfy their application requirements on data retrieval latency and availability, minimize cost, and dynamically adjust the allocation to adapt to request variation.

Many previous works [10], [11], [12], [13] focus on finding the minimum amount of resources to support the application workload to reduce cloud storage cost in a single CSP. However, there have been only a few works that studied cloud storage cost minimization for a storage service across multiple CSPs with different prices. Within our knowledge, SPANStore [9] is the only work that handles this problem. It aims to minimize the cloud storage cost while satisfying the latency and failure requirement across multiple CSPs. However, it neglects both the resource reservation pricing model and the datacenter capacity limits for serving Get/Put requests. A datacenter's Get/Put capacity is represented by the Get/Put rate (i.e., the number of Gets/Puts in a unit time period) it can handle. Reserving resources in advance can save significant payment cost for customers and capacity limit is critical for guaranteeing SLOs. For example, Amazon DynamoDB [8] shows a capacity limitation of 360,000 reads per hour. Also, datacenter network overload occurs frequently [14], [15], which leads to packet loss. If the integer program used to create a data allocation schedule in [9] is modified to be capacity-limit aware, it becomes NP-hard, which cannot be easily resolved. Therefore, we need to consider the resource reservation pricing model and datacenter capacity limits when building a minimum-cost cloud storage service across multiple CSPs.

To handle the above-stated two challenges, we propose a geo-distributed cloud storage system for Data storage, request Allocation and resource Reservation across multiple CSPs (*DAR*). It transparently helps customers to minimize their payment cost while guaranteeing their SLOs. Building a geo-distributed cloud storage across multiple CSPs can avoid the vendor lock-in problem since a customer will not be constrained to an obsolete provider and can always choose the optimal CSPs for the cloud storage service. We summarize our contributions below:

- We have modeled the cost minimization problem under multiple constraints using the integer programming.
- We introduce a heuristic solution including:
  (1) A dominant-cost based data allocation algorithm, which finds the dominant cost (Storage, Get or Put) of each data item and allocates it to the datacenter with the minimum unit price of this dominant cost to reduce cost in the pay-as-you-go manner.
  (2) An optimal resource reservation algorithm, which maximizes the saved payment cost by reservation from the pay-as-you-go payment while avoiding over reservation.
- We further propose three enhancement methods to reduce the payment cost and service latency:
  (1) Coefficient based data reallocation, which aims to balance the workloads among all billing periods in order to minimize the payment cost by maximizing the reservation benefit.
  (2) Multicast based data transferring, which builds a minimum spanning tree to create new data replicas in order to minimize the Transfer cost for replica creation in a new data allocation deployment.
  (3) Request redirection based congestion control, which redirects Get requests from overloaded datacenters to underloaded datacenters that have received Gets more than (or less than) their expected number of Gets after data allocation to minimize the payment cost, respectively.
- 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 and efficiency of our system in cost minimization, SLO compliance and system overhead in comparison with previous systems.

Our dominant-cost based data allocation algorithm finds the dominant cost (Storage, Get or Put) of each data item. For example, if a data item is storage dominant, it means its storage cost is much higher than the sum of other costs. Therefore, *DAR* is suitable for the scenarios in which most customer data items have dominant cost. The rest of the paper is organized as follows. Section 2 depicts the cost minimization problem under the SLO for a geo-distributed cloud storage system over multiple CSPs. Sections 3 presents our heuristic solution for the cost minimization problem. Section 4 presents our enhancement methods to reduce the payment cost and service latency. Section 5 presents the infrastructure to realize the algorithms. Section 6 presents the trace-driven experimental results of *DAR* compared with previous systems. Section 7 presents the related work. Section 8 concludes our work with remarks on our future work.

2 Problem Statement

2.1 Background

We call a datacenter that operates a customer's application its *customer datacenter*. According to the operations of a customer's clients, the customer datacenter generates read/write requests to a storage datacenter storing the requested data. A customer may have multiple customer datacenters (denoted by D_c). We use $c_i \in D_c$ to denote the i^{th} customer datacenter of the customer. We use D_s to denote all datacenters provided by all cloud providers and use $p_j \in D_s$ to denote storage datacenter j. A client's Put/Get request is forwarded from a customer datacenter to the storage datacenter of the requested data. One type of SLO specifies the Get/Put bounded latency and the percentage of requests obeying the deadline [7]. Another type of SLO guarantees the data availability in the form of a service probability [16] by ensuring a certain number of replicas in different locations [1]. *DAR* considers both types to form its SLO and can adapt to either type easily. This SLO specifies

TABLE 1: Notations of inputs and outputs in scheduling.

Input	Description	Input	Description
$c_i \in D_c$	i^{th} customer datacenter	$p_j \in D_s$	j^{th} storage datacenter
$\zeta^g_{p_j}/\zeta^p_{p_j}$	Get/Put capacity of p_j	$A^{t_k}_{p_j}$	num. of Gets served by p_j
n	num. of billing periods in T	$p^s_{p_j}/p^t_{p_j}$	unit storage/transfer price
$p^g_{p_j}/p^p_{p_j}$	unit Get/Put price of p_j	α_{p_j}	reservation price ratio of p_j
L^g/L^p	Get/Put deadline	ϵ^g/ϵ^p	% of violated Gets/Puts
Q^g/Q^p	Get/Put SLO satisfaction	β	minimum num. of replicas
D	entire data set	d_l/s_{d_l}	data $l \in D/d_l$'s size
T	reservation time period	t_k	k^{th} billing period in T
$v^{d_l,t_k}_{c_i}$ $/u^{d_l,t_k}_{c_i}$	Get/Put rates on d_l from c_i	$S^g_{c_i}/S^p_{d_l}$	datacenter set satisfying Get/Put deadline
$\phi^g_{p_j}$ $/\phi^p_{p_j}$	available Get/Put capacities of p_j	$F^g_{c_i,p_j}(x)$ $/F^p_{c_i,p_j}(x)$	CDF of Get/Put latency from c_i to p_j

Output	Description	Output	Description
$R^g_{p_j}/R^p_{p_j}$	reserved num. of Gets/Puts	$X^{d_l,t_k}_{p_j}$	existence of d_l's replica in p_j
$H^{d_l,t_k}_{c_i,p_j}$	% of requests on d_l from c_i resolved by p_j in t_k	C_t	total payment cost

the deadlines for the Get/Put requests (L^g and L^p), the maximum allowed percentage of data Get/Put operations beyond the deadlines (ϵ^g and ϵ^p), and the minimum number of replicas (denoted by β) among storage datacenters [1]. All the data items from one customer have the same SLO requirement. If a customer has different sets of data with different SLOs, then we treat each data set separately in calculating data allocation schedule. A customer can also have an elastic deadline requirement (e.g., different SLOs in different time periods). We can easily accommodate this requirement by splitting the whole time period to different periods when specifying the constraints. For a customer datacenter's Get request, any storage datacenter holding the requested data (i.e., replica datacenter) can serve this request. A cloud storage system usually specifies the *request serving ratio* for each replica datacenter of a data item (denoted by d_l) during billing period t_k (e.g., one month).

The storage and data transfer are charged in the pay-as-you-go manner based on the unit price. The Get/Put operations are charged in the manners of both pay-as-you-go and reservation. In the reservation manner, the customer specifies and prepays the number of Puts/Gets per reservation period T (e.g., one year). The unit price for the reserved usage is much cheaper than the unit price of the pay-as-you-go manner (by a specific percentage) [8]. For simplicity, we assume all datacenters have comparable price discounts for reservation. That is, if a datacenter has a low unit price in the pay-as-you-go manner, it also has a relatively low price in the reservation manner. The amount of overhang of the reserved usage is charged by the pay-as-you-go manner. Therefore, the payment cost can be minimized by increasing the amount of Gets/Puts charged by reservation and reducing the amount of Gets/Puts for over reservation, which reserves more Gets/Puts than actual usage. For easy reference, we list the notations used in the paper in Table 1.

2.2 Problem Formulation

For a customer, *DAR* aims to find a schedule that allocates each data item to a number of selected datacenters, allocates request serving ratios to these datacenters and determines reservation in order to guarantee the SLO and minimize the payment cost of the customer. In the following, we formulate this problem using the integer programming. Our objective is to minimize the total payment cost for a customer. In the problem, we need to satisfy i) the deadline requirement or SLO guarantee, that is, the Get/Put requests (L^g and L^p) deadlines must be satisfied, and the maximum allowed percentage of data Get/Put operations beyond the deadlines is (ϵ^g and ϵ^p); ii) the data availability requirement, i.e., the minimum number of replicas (denoted by β) among storage datacenters [1]; and iii) the datacenter capacity constraint, i.e., each datacenter's load is no more than its capacity. Below, we introduce the objective and each constraint in detail.

Payment minimization objective. We aim to minimize the total payment cost for a customer (denoted as C_t). It is the sum of the total Storage, Transfer, Get and Put cost during entire reservation time T, denoted by C_s, C_c, C_g and C_p:

$$C_t = C_s + C_c + C_g + C_p. \tag{1}$$

The storage cost in a datacenter is the product of data size and unit storage price of each datacenter. Then, the total storage cost is calculated by:

$$C_s = \sum_{t_k \in T} \sum_{d_l \in D} \sum_{p_j \in D_s} X^{d_l,t_k}_{p_j} * p^s_{p_j} * s_{d_l}, \tag{2}$$

where s_{d_l} denotes the size of data d_l, $p^s_{p_j}$ denotes the unit storage price of datacenter p_j, and $X^{d_l,t_k}_{p_j}$ denotes a binary variable: it is 1 if d_l is stored in p_j during t_k; and 0 otherwise.

The transfer cost for importing data to storage datacenters is one-time cost. The imported data is not stored in the datacenter during the previous period t_{k-1} ($X^{d_l,t_{k-1}}_{p_j} = 0$), but is stored in the datacenter in the current period t_k ($X^{d_l,t_k}_{p_j} = 1$). Therefore, we use $\theta_{d_l} = X^{d_l,t_k}_{p_j}(1 - X^{d_l,t_{k-1}}_{p_j})$ to denote whether data d_l is imported to datacenter. Thus, the data transfer cost is the product of the unit price and the size if $\theta_{d_l} = 1$.

$$C_c = \sum_{t_k \in T} \sum_{d_l \in D} \sum_{p_j \in D_s} \theta_{d_l} * p^t(p_j) * s_{d_l}, \tag{3}$$

where $p^t(p_j)$ is the cheapest unit transfer price of replicating d_l to p_j among all datacenters storing d_l. The Get/Put billings are based on the pay-as-you-go and reservation manners. The reserved number of Gets/Puts (denoted by $R^g_{p_j}$ and $R^p_{p_j}$) is decided at the beginning of each reservation time period T. The reservation prices for Gets and Puts are a specific percentage of their unit prices in the pay-as-you-go manner [8]. Then, we use α to denote the reservation price ratio, which means that the unit price for reserved Gets/Puts is $\alpha * p^g_{p_j}$ and $\alpha * p^p_{p_j}$, respectively. The amount of resource beyond the reserved amount that needs to pay for is $Max\{\sum_{c_i} r^{t_k}_{c_i,p_j} * t_k - R^g_{p_j}, 0\}$, where $\sum_{c_i} r^{t_k}_{c_i,p_j} * t_k$ is actually used resource. The Get/Put cost is the sum of this payment and the reservation payment.

Thus, the Get/Put cost is calculated by:

$$C_g = \sum_{t_k} \sum_{p_j} (Max\{\sum_{c_i} r^{t_k}_{c_i,p_j} * t_k - R^g_{p_j}, 0\} + \alpha R^g_{p_j}) * p^g_{p_j}, \tag{4}$$

$$C_p = \sum_{t_k} \sum_{p_j} (Max\{\sum_{c_i} w^{t_k}_{c_i,p_j} * t_k - R^p_{p_j}, 0\} + \alpha R^p_{p_j}) * p^p_{p_j}, \tag{5}$$

where $r^{t_k}_{c_i,p_j}$ and $w^{t_k}_{c_i,p_j}$ denote the average Get and Put rates from c_i to p_j per unit time in t_k, respectively.

SLO guarantee. To formulate the SLO objective, we first need to calculate the actual percentage of Gets/Puts satisfying the latency requirement within billing period t_k. To this end, we need to know the percentage of Gets and Puts from c_i to p_j within the deadlines L^g and L^p, denoted by $F^g_{c_i,p_j}(L_g)$ and $F^p_{c_i,p_j}(L_p)$. Thus, *DAR* records the Get/Put latency from c_i to p_j, and periodically calculates their cumulative distribution functions (CDFs) represented by $F^g_{c_i,p_j}(L_g)(x)$ and $F^p_{c_i,p_j}(L_p)(x)$. The average Get and Put rates from c_i to p_j per unit time in t_k (i.e., $r^{t_k}_{c_i,p_j}$ and $w^{t_k}_{c_i,p_j}$) equal the product of the average Get and Put rates on each data (denoted by d_l) from c_i per unit time in t_k (denoted by $v^{d_l,t_k}_{c_i}$ and $u^{d_l,t_k}_{c_i}$) and the ratio of requests for

d_l from c_i resolved by p_j during t_k (denoted by $H^{d_l,t_k}_{c_i,p_j}$ and $X^{d_l,t_k}_{p_j} \in [0,1]$). For easy Get/Put rate prediction, as in [17], [9], *DAR* conducts coarse-grained data division to achieve relatively stable request rates since a fine-grained data division makes the rates vary largely and hence difficult to predict. It divides all the data to relatively large data items, which of each is formed by a number of data blocks, such as aggregating data of users in one location [18]. The prediction accuracy depends on the stability of the Get/Put rate of the applications. High stability leads to more accurate prediction while low stability leads to less accurate prediction.

We can calculate the actual percentage of Gets/Puts satisfying the latency requirement within t_k for a customer (denoted as $q^{t_k}_g$ and $q^{t_k}_p$) (we omit t_k for simplicity below):

$$q_g = \frac{\sum_{c_i \in D_c} \sum_{p_j \in D_s} r_{c_i,p_j} * F^g_{c_i,p_j}(L_g)}{\sum_{c_i \in D_c} \sum_{p_j \in D_s} r_{c_i,p_j}}, \quad q_p = \frac{\sum_{c_i \in D_c} \sum_{p_j \in D_s} w_{c_i,p_j} * F^p_{c_i,p_j}(L_p)}{\sum_{c_i \in D_c} \sum_{p_j \in D_s} w_{c_i,p_j}}. \tag{6}$$

To judge whether the deadline SLO of a customer is satisfied during t_k, we define the Get/Put SLO satisfaction level of a customer, denoted by Q^g and Q^p.

$$\begin{aligned} Q^g &= Min\{Min\{q^{t_k}_g\}_{\forall t_k \in T}, (1-\epsilon^g)\}/(1-\epsilon^g) \\ Q^p &= Min\{Min\{q^{t_k}_p\}_{\forall t_k \in T}, (1-\epsilon^p)\}/(1-\epsilon^p). \end{aligned} \tag{7}$$

We see that if

$$Q^g * Q^p = 1 \tag{8}$$

i.e., $Q^g = Q^p = 1$, the customer's deadline SLO is satisfied.

Next, we formulate whether the data availability SLO is satisfied. To satisfy the data availability SLO, there must be at least β datacenters that store the requested data and satisfy the Get deadline SLO for each Get request of c_i during t_k. The set of all datacenters satisfying the Get deadline SLO for requests from c_i (denoted by $S^g_{c_i}$) is represented by:

$$S^g_{c_i} = \{p_j | F^g_{c_i,p_j}(L_g) \geq (1-\epsilon^g)\}. \tag{9}$$

The set of data items read by c_i during t_k is represented by: $G^{t_k}_{c_i} = \{d_l | v^{d_l,t_k}_{c_i} > 0 \wedge d_l \in D\}$. Then, the data availability constrain can be expressed as during any t_k, there exist at least β replicas of any $d_l \in G^{t_k}_{c_i}$ stored in $S^g_{c_i}$:

$$\forall c_i \forall t_k \forall d_l \in G^{t_k}_{c_i} \sum_{p_j \in S^g_{c_i}} X^{d_l,t_k}_{p_j} \geq \beta \tag{10}$$

Each customer datacenter maintains a table that maps each data item to its replica datacenters with assigned request serving ratios.

Datacenter capacity constraint. Each datacenter has limited capacity to supply Get and Put service, respectively [19]. Therefore, the cumulative Get rate and Put data rate of all data in a datacenter p_j should not exceed its Get capacity and Put capacity (denoted by $\zeta^g_{p_j}$ and $\zeta^p_{p_j}$), respectively. Since storage is relatively cheap and easy to be increased, we do not consider the storage capacity as a constraint. This constraint can be easily added to our model, if necessary. Then, we can calculate the available Get and Put capacities, denoted by $\phi^g_{p_j}$ and $\phi^p_{p_j}$:

$$\begin{aligned} \phi^g_{p_j} &= Min\{\zeta^g_{p_j} - \textstyle\sum_{c_i \in D_c} r^{t_k}_{c_i,p_j}\}_{\forall t_k \in T} \\ \phi^p_{p_j} &= Min\{\zeta^p_{p_j} - \textstyle\sum_{c_i \in D_c} w^{t_k}_{c_i,p_j}\}_{\forall t_k \in T} \end{aligned}$$

If both $\phi^g_{p_j}$ and $\phi^p_{p_j}$ are no less than 0, the datacenter capacity constraint is satisfied. Then, we can express the datacenter capacity constraint by:

$$\forall p_j \; Min\{\phi^g_{p_j}, \phi^p_{p_j}\} \geq 0 \tag{11}$$

Problem statement. Finally, we formulate the problem that minimizes the payment cost under the aforementioned constraints using the integer programming.

$$\min \; C_t \; (calculated\ by\ Formulas\ (2), (3), (4)\ and\ (5)) \tag{12}$$

$$s.t. \quad Q^g * Q^p = 1 \tag{8}$$

$$\forall c_i \forall t_k \forall d_l \in G^{t_k}_{c_i} \sum_{p_j \in S^g_{c_i}} X^{d_l,t_k}_{p_j} \geq \beta \tag{10}$$

$$\forall p_j \; Min\{\phi^g_{p_j}, \phi^p_{p_j}\} \geq 0 \tag{11}$$

$$\forall c_i \forall p_j \forall t_k \forall d_l \; H^{d_l,t_k}_{c_l,p_j} \leq X^{d_l,t_k}_{p_j} \leq 1 \tag{13}$$

$$\forall c_i \forall t_k \forall d_l \sum_{p_j} H^{d_l,t_k}_{c_i,p_j} = 1 \tag{14}$$

Constraints (8), (10) and (11) satisfy the deadline requirement and data availability requirement in the SLO and the datacenter capacity constraint, as explained previously. Constraints (13) and (14) together indicate that any request should be served by a replica of the targeted data.

Operation. Table 1 indicates the input and output parameters in this integer programming. The unit cost of Gets/Puts/Storage/Transfer usage is provided or negotiated with the CSPs. During each billing period t_k, *DAR* needs to measure the latency CDF of Get/Put ($F^g_{c_i,p_j}(L_g)(x)$ and $F^p_{c_i,p_j}(L_p)(x)$), the size of new data items d_l (s_{d_l}), and the data Get/Put rate from each c_i ($v^{d_l,t_k}_{c_i}$ and $u^{d_l,t_k}_{c_i}$). The output is the data storage allocation ($X^{d_l,t_k}_{p_j}$), request servicing ratio allocation ($H^{d_l,t_k}_{c_i,p_j}$) and the total cost C_t. The optimal Get/Put reservation in each storage datacenter ($R^g_{p_j}/R^p_{p_j}$) is an output at the beginning of reservation time period T and is an input at each billing period t_k in T. After each t_k, T is updated as the remaining time after t_k. We use $T \backslash \{t_k\}$ to denote the update T. *DAR* adjusts the data storage and request distribution among datacenters under the determined reservation using the same procedure. This procedure ensures the maximum payment cost saving in request rate variation. The billing period and reservation period are determined by the cloud customer and cloud provider. Shorter periodicity (more frequent schedule calculation) generates higher overhead, while longer periodicity (less frequent schedule calculation) generates lower overhead.

This integer programming problem is NP-hard. A simple reduction from the generalized assignment problem [20] can be used to prove this. We skip detailed formal proof due to limited space. The NP-hard feature makes the solution calculation very time consuming. We then propose a heuristic solution to this cost minimization problem in the next section.

3 Data Allocation and Resource Reservation

DAR has two steps. First, its dominant-cost based data allocation algorithm (Section 3.1) conducts storage and request allocation scheduling that leads to the lowest total payment only in the pay-as-you-go manner. Second, its optimal resource reservation algorithm (Section 3.2) makes a reservation in each used storage datacenter to maximally reduce the total payment.

- Dominant-cost based data allocation algorithm. To reduce the total payment in the pay-as-you-go manner as much as possible, *DAR* tries to reduce the payment for each data item. Specifically, it finds the dominant cost (Storage, Get or Put) of each data item and allocates it to the datacenter with the minimum unit price of this dominant cost.
- Optimal resource reservation algorithm. It is a challenge to maximize the saved payment cost by reservation from the pay-as-you-go payment while avoiding over reservation. To handle this challenge, through theoretical analysis, we find the optimal reservation amount, which avoids both over reservation and under reservation as much as possible.

3.1 Dominant-Cost based Data Allocation

A valid data allocation schedule must satisfy Constraints (8), (10), (11), (13) and (14). To this end, *DAR* first identifies the datacenter candidates that satisfy Constraint (8). Then, *DAR* selects datacenters from the candidates to store each data item requested by c_i to satisfy other constraints and achieve Objective (12). We introduce these two steps below.

Datacenter candidate identification. Constraint (8) guarantees that the deadline SLO is satisfied. That is, the percentage of data Get and Put operations of a customer beyond the specified deadlines is no more than ϵ^g and ϵ^p, respectively. To satisfy this constraint, some Get/Put response datacenters can have service latency beyond the deadlines with probability larger than ϵ^g and ϵ^p, while others have the probability less than ϵ^g and ϵ^p. Since finding a combination of these two types of datacenters to satisfy the SLO is complex, *DAR* simply finds the datacenters that have the probability less than ϵ^g and ϵ^p. That is, if p_j serves Get/Put from c_i, p_j has $F^g_{c_i,p_j}(L_g) \geq 1-\epsilon^g$ and $F^p_{c_i,p_j}(L_p) \geq 1-\epsilon^p$:

$$\forall t_k \in T \, \forall c_i \in D_c, r^{t_k}_{c_i,p_j} > 0 \rightarrow F^g_{c_i,p_j}(L_g) \geq 1-\epsilon^g \quad (15)$$

$$\forall t_k \in T \, \forall c_i \in D_c, w^{t_k}_{c_i,p_j} > 0 \rightarrow F^p_{c_i,p_j}(L_p) \geq 1-\epsilon^p \quad (16)$$

Then, by replacing $F^g_{c_i,p_j}(L_g)$ with $1-\epsilon^g$ in Equation (6), we can have $q^{t_k}_g \geq 1-\epsilon^g$ which ensures the Get SLO. The same applies to the Put SLO. Therefore, the new Constraints (15) and (16) satisfy Constraint (8).

Accordingly, for each customer's datacenter c_i, we can find $S^g_{c_i}$ using Equation (9), a set of storage datacenters that satisfy Get SLO for Gets from c_i. For each data d_l, we can find another set of storage datacenters $S^p_{d_l} = \{p_j | \forall c_i \forall t_k, (u^{d_l,t_k}_{c_i} > 0) \rightarrow (F^g_{c_i,p_j}(L_g)(L^p) \geq 1-\epsilon^p)\}$ that consists of datacenters satisfying Put SLO of d_l. To allocate d_l requested by c_i, in order to satisfy both Get and Put delay SLOs, we can allocate d_l to any $p_j \in S^g_{c_i} \cap S^p_{d_l}$.

Min-cost storage datacenter selection. After the datacenter candidates $S^g_{c_i} \cap S^p_{d_l}$ are identified, *DAR* needs to further select datacenters that lead to the minimum payment cost. For this purpose, we can use a greedy method, in which the cost of storing data item d_l in each $p_j \in S^g_{c_i} \cap S^p_{d_l}$ (denoted as $C^{d_l}_{c_i,p_j}$) is calculated based on Equation (1) and the p_j with the lowest cost is selected. However, such a greedy method is time consuming. Our dominant-cost based data allocation algorithm can speed up the datacenter selection process. Its basic idea is to find the dominant cost among the different costs in Equation (1) for each data item d_l requested by each c_i and stores d_l in the datacenter that minimizes the dominant cost.

If one cost based on its minimum unit price among datacenters is larger than the sum of the other costs based on their maximum unit prices among datacenters, we consider this cost as the dominant cost. We do not consider the transfer cost for importing data when determining the dominant cost of a data item since it is one-time cost and comparatively small compared to other three costs, and then is less likely to be dominant in the total cost of a data item.

We classify each d_l requested by c_i into four different sets: Put dominant, Get dominant, Storage dominant and balanced. Data blocks in the balanced set do not have an obvious dominant cost. A data item should be stored in the datacenter in the candidates $S^g_{c_i} \cap S^p_{d_l}$ that has the lowest unit price in its dominant resource in order to reduce its cost as much as possible. Finding such a datacenter for each data item d_l requested by a given c_i is also time-consuming. Note that $S^g_{c_i}$ is common for all data items requested by c_i. Then, to reduce time complexity, we can calculate $S^g_{c_i}$ only one time. From $S^g_{c_i}$, we select the datacenter that belongs to $S^p_{d_l}$ to allocate each d_l requested by a given c_i.

Algorithm 1: Dominant-cost based data allocation.

```
1  for each $c_i$ in $D_c$ do
2      $L^s_{c_i}$, $L^g_{c_i}$ and $L^g_{c_i}$ are $S^g_{c_i}$ sorted in an increasing order of unit
       Storage/Get/Put price, respectively.
3      for each $d_l$ with $\exists\, t_k\; d_l \in G^{t_k}_{c_i}$ do
4          $H = 100\%$;
5          switch $d_l$ with $H^{d_l}_{c_i} = H$ do
6              case dominant
7                  $L = L^s_{c_i}$ or $L^g_{c_i}$ or $L^p_{c_i}$ according to the dominant
                   cost is Storage or Get or Put
8              case balanced
9                  Find $p_j \in S^g_{c_i} \cap S^p_{d_l}$ with the smallest $C^{d_l}_{c_i,p_j}$ and
                   satisfies all constraints
10         for each $p_j$ with $p_j \in L \cap S^p_{d_l}$ do
11             if $(X^{d_l}_{p_j} = 1 \rightarrow \phi^p_{p_j} < 0) \vee (\phi^g_{p_j} = 0)$ then
12                 Continue;
13             Find the largest $H^{d_l}_{c_i,p_j}$ satisfying $\phi^g_{p_j} \geq 0 \wedge H \geq H^{d_l}_{c_i,p_j}$;
14             if $C^{d_l}_{c_i,p_j} \leq C^{d_l}_{c_i,p_{k\,(k=j+1,\ldots,j+c)}}$ when
               $H_{c_i,p_k} = H_{c_i,p_j}$ then
15                 $X^{d_l}_{p_j} = 1$; $H = H - H^{d_l}_{c_i,p_j}$;
16             else
17                 $H^{d_l}_{c_i,p_j} = 0$;
18             if $\sum_{p_j \in S^g_{c_i}} X^{d_l}_{p_j} \geq \beta \wedge H = 0$ then
19                 break;
```

The pseudocode of this algorithm is shown in Algorithm 1, in which all symbols without t_k denote all remaining billing periods in T. For each c_i, we sort $S^g_{c_i}$ by increasing order of unit Put cost, unit Get cost and unit Storage cost, respectively, which results in three sorted lists called Put, Get and Storage sorted datacenter lists, respectively. We use Max_g/Min_g, $Max_s\;/Min_s$ and Max_p/Min_p to denote the maximum/minimum Get unit prices, Storage unit prices and Put unit prices among the datacenters belonging to $S^g_{c_i}$.

For each data d_l requested by a given c_i, we calculate its maximum/minimum Storage cost, Get cost and Put cost:

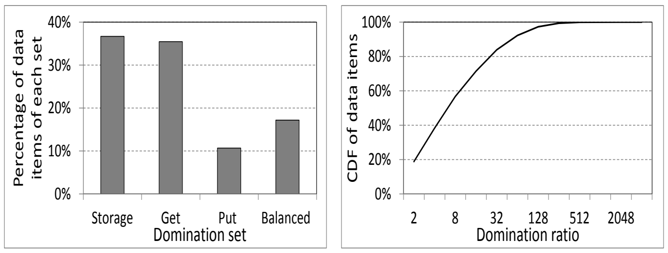
$Max^{d_l}_s = \sum_{t_k \in T} Max_s * s_{d_l} * t_k$,

$Max^{d_l}_g = \sum_{t_k \in T} Max_g * v^{d_l,t_k}_{c_i} * t_k$,

$Max^{d_l}_p = \sum_{t_k \in T} Max_p * u^{d_l,t_k}_{c_i} * t_k$,

$Min^{d_l}_s$, $Min^{d_l}_g$ and $Min^{d_l}_p$ are calculated similarly. If $Min^{d_l}_s >> Max^{d_l}_g + Max^{d_l}_p$, we regard that the data belongs to the Storage dominant set. Similarly, we can decide whether d_l belongs to the Get or Put dominant set. If d_l does not belong to any dominant set, it is classified into the balanced set. The datacenter allocation for data items in each dominant set is conducted in the same manner, so we use the Get dominant set as an example to explain the process.

For each data d_l in the Get dominant set, we try each datacenter from the top in the Get sorted datacenter list. We find a datacenter satisfying Get/Put capacity constraints (Constraint (11)) (Line 11) and Get/Put latency SLO constraints (Constraint (8)) (Lines 9-10), and determine the largest possible request serving ratio of this replica. The subsequent datacenters in the list may have a similar unit price for Gets but have different unit prices for Puts and Storage, which may lead to lower total cost for this data allocation. Therefore, we choose a number of subsequent datacenters, calculate $C^{d_l}_{c_i,p_k}$ for d_l, where $k \in [j+1, j+c]$, and choose p_j to create a replica and assign requests to (Constraint (13)) (Lines 15-17) if $C^{d_l}_{c_i,p_j}$ is smaller than all



(a) Data distribution (b) CDF of data items

Fig. 2: Efficiency and the validity of the dominant-cost based data allocation algorithm .

$C^{d_l}_{c_i,p_k}$. When several datacenters satisfy the SLO constrains and also provide the same total cost, we choose the datacenter that is geographically closer to the location of the most requesters of data item d_l in order to further reduce the data request latency. If there are no less than β replicas (Constraint (10)) (Line 18), and the remaining request ratio to assign is equal to 0 (Constraint (14)) (Lines 4 and 18), the data allocation for d_l is completed. For any data in the balanced set, we choose the datacenter in $S^g_{c_i} \cap S^p_{d_l}$ that generates the lowest total cost for d_l. In the datacenter selection process, the constraints in Section 2.2 are checked to ensure the selected datacenter satisfying the conditions. After allocating all data items, we get a valid data allocation schedule with sub-optimal cost minimization.

Efficiency and validity of the algorithm. The efficiency of the dominant-cost based data allocation algorithm depends on the percentage of data items belonging to the three dominant sets, since it allocates data in each dominant set much more efficiently than data in the balanced set. We then measure the percentage of data items in each data set from a real trace in order to measure the efficiency of the algorithm. We get the Put rates of each data from the publicly available wall post trace from Facebook New Orleans networks [21], which covers inter-posts between 188,892 distinct pairs of 46,674 users. We regard each user's wall post as a data item. The data size is typically smaller than 1 KB. The Get:Put ratio is typically 100:1 in Facebook's workload [22], from which we set the Get rate of each data item accordingly. We uses the unit prices for Storage, Get and Put in all regions in Amazon S3, Microsoft Azure and Google cloud storage [1], [2], [3]. For each data item d_l, we calculated its dominant ratio of Storage as $Min^{d_l}_s/(Max^{d_l}_g + Max^{d_l}_p)$, and if it is no less than 2, we consider d_l as storage dominant. Similarly, we can get a dominant ratio of Get and Put. Figure 2(a) shows the percentage of data items belonging to each dominant set. We can see that most of the data items belong to the Storage dominant set and Get dominant set, and only 17.2% of data items belong to the balanced set. That is because in the trace, most data items are either rarely or frequently requested with majority costs as either Storage or Get cost. The figure indicates that the dominant-cost based data allocation algorithm is efficient since most of the data belongs to the three dominant sets rather than the balanced set. Figure 2(b) shows the CDF of data items over the dominant ratio in the Get dominant set as an example. It shows that most of the data items in the Get dominant set have a dominant ratio no less than 8, and the largest dominant ratio reaches 3054. Thus, the cost of these data items quickly decreases when the Get unit price decreases, and then we can allocate them to the datacenter with the minimum Get unit price. These results support the algorithm design of finding appropriate datacenter p_j in the sorted datacenter list of the dominant resource of a data item. Note that *DAR* is most effective not in the balanced set and it is effective for the scenarios, in which most customer data items have dominant cost, though it also chooses the datacenters with the minimum cost for the data items in the balanced set which do not have dominant cost.

3.2 Optimal Resource Reservation

After the dominant-cost based allocation, we need to determine reserved Get/Put rates for each datacenter in order to further reduce the cost as much as possible given a set of allocated data items and their Get/Put rate over T. Since the method to determine the reserved Get and Put rates is the same, we use Get as an example to present this method. Before we introduce how to find reservation amount to achieve the maximum reservation benefit, we first introduce the benefit function of the reservation, denoted as $F_{p_j}(x)$, where x is the reserved number of Gets/Puts in any billing period t_k. The benefit is the difference between the saved cost by using reservation instead of pay-as-you-go manner and the cost for over-reservation. The over reservation cost includes the cost for over reservation and the over calculated saving. Combining these two parts, we can calculate the benefit by

$$F_{p_j}(x) = (\sum_{t_k \in T} x * (1-\alpha) * p^g_{p_j}) - O_{p_j}(x) * p^g_{p_j}, \qquad (17)$$

where $O_{p_j}(x)$ is the over reserved number of Gets. It is calculated by

$$O_{p_j}(x) = \sum_{t_k \in T} Max\{0, x - \sum_{d_l \in D} \sum_{c_i \in D_c} r^{t_k}_{c_i,p_j} * t_k\}, \qquad (18)$$

where x is the rserved amount and $\sum_{d_l \in D} \sum_{c_i \in D_c} r^{t_k}_{c_i,p_j} * t_k$ is actually used amount. Recall that $R^g_{p_j}$ is the optimal number of reserved Gets for each billing period during T in a schedule. That is, when $x = R^g_{p_j}$, $F_{p_j}(x)$ reaches the maximum value, represented by $B_{p_j} = F_{p_j}(R^g_{p_j}) = Max\{F_{p_j}(x)\}_{x \in N^+}$.

In the following, we first prove Corollary 3.1, which supports the rationale that allocating as much data as possible to the minimum-cost datacenter in the dominant-cost based data allocation algorithm is useful in getting a sub-optimal result of reservation benefit. Then, we present Corollary 3.2 that helps find reservation x to achieve the maximum reservation benefit. Finally, we present Theorem 3.1 that shows how to find this reservation x.

Corollary 3.1. *Given a datacenter p_j that stores a set of data items, allocating a new data item d_l and its requests to this datacenter, its maximum reservation benefit B_{p_j} is non-decreasing.*

Proof. After allocating d_l to p_j, we use $F'_{p_j}(x)$ to denote the new reservation benefit function since $r^{t_k}_{c_i,p_j}$ in Equation (18) is changed. Then, we can get $F'_{p_j}(R^g_{p_j}) \geq F_{p_j}(R^g_{p_j})$ since $r^{t_k}_{c_i,p_j}$ is not decreasing. Since the new reserved benefit $B'_{p_j} = Max\{F'_{p_j}(x)\}_{x \in N^+}$, thus $B'_{p_j} \geq F'_{p_j}(R^g_{p_j}) \geq F_{p_j}(R^g_{p_j}) = B_{p_j}$ after d_l is allocated. $\square$

We define the number of Gets in t_k as $m = Max\{\sum_{d_l \in D} \sum_{c_i \in D_c} r^{t_k}_{c_i,p_j} * t_k\}_{t_k \in T}$. Then, according to Equation (17), we can get the optimal reservation Gets $R^g_{p_j} \in [0, m]$. Thus, by looping all integers within $[0, m]$, we can get the optimal reservation that results in maximum F_{p_j}. This greedy method, however, is time consuming. In order to reduce the time complexity, we first prove Corollary 3.2, based on which we introduce a binary search based optimal reservation method.

Corollary 3.2. *For a datacenter p_j, its benefit function $F_{p_j}(x)$ is increasing when $x \in [0, R^g_{p_j})$ and decreasing when $x \in (R^g_{p_j}, m]$.*

Proof. According to Equation (17), we define $F_I(x) = F_{p_j}(x) - F_{p_j}(x-1) = (n*(1-\alpha) - O_I(x)) * p^g_{p_j}$, where n is the number of billing periods in T. The extra over reserved number of Gets of $O_{p_j}(x)$ compared to $O_{p_j}(x-1)$, represented by $O_I(x) = O_{p_j}(x) - O_{p_j}(x-1)$, equals the number of billing periods during T that have the number of Gets smaller than x, i.e., $\sum_{c_i \in D_c} r^{t_k}_{c_i,p_j} * t_k < x$. Therefore, $O_I(x)$ is increasing. At first $O_I(0) = 0$, and when $O_I(x) < n*(1-\alpha)$, then $F_I(x) > 0$, which means $F_{p_j}(x)$ is increasing; when $O_I(x) > n*(1-\alpha)$, then $F_I(x) < 0$, which means $F_{p_j}(x)$ is decreasing. Therefore, $F_{p_j}(x)$ is increasing and then decreasing. Since $F_{p_j}(R^g_{p_j})$ reaches the largest $F_{p_j}(x)$, we can derive that $F_{p_j}(x)$ is increasing when $x \in [0, R^g_{p_j})$, and decreasing when $x \in (R^g_{p_j}, m]$. □

We use $A^{t_k}_{p_j} = \sum_{c_i \in D_c} r^{t_k}_{c_i,p_j} * t_k$ to denote the total number of Gets served by p_j during t_k, and define $\mathbf{A} = \{A^{t_1}_{p_j}, A^{t_2}_{p_j}, ..., A^{t_n}_{p_j}\}$.

Theorem 3.1. *To achieve the maximum reservation benefit, the reservation amount x is the N^{th} smallest value in $\mathbf{A} = \{A^{t_1}_{p_j}, A^{t_2}_{p_j}, ..., A^{t_n}_{p_j}\}$, where N equals $\lceil n*(1-\alpha)\rceil + 1$ or $\lfloor n*(1-\alpha)\rfloor + 1$.*

Proof. The proof of Corollary 3.2 indicates that when $O_I(x) = \lceil n*(1-\alpha)\rceil$ or $\lfloor n*(1-\alpha)\rfloor$, $F_{p_j}(x)$ can reach B_{p_j}. As indicated above, $O_I(x)$ represents the number of billing periods during T with $A^{t_k}_{p_j} = \sum_{c_i \in D_c} r^{t_k}_{c_i,p_j} * t_k < x$. Therefore, when x is the N^{th} smallest value in $\mathbf{A}$, where N equals $\lceil n*(1-\alpha)\rceil + 1$ or $\lfloor n*(1-\alpha)\rfloor + 1$, $F_{p_j}(x)$ reaches B_{p_j}. □

Algorithm 2: Binary search based resource reservation.

```
1 Sort $\mathbf{A} = \{A^{t_1}_{p_j}, A^{t_2}_{p_j}, ..., A^{t_n}_{p_j}\}$ in ascending order;
2 $N_1 = \lfloor n*(1-\alpha)\rfloor + 1$; $N_2 = \lceil n*(1-\alpha)\rceil + 1$;
3 $x_1 =$ the ${N_1}^{th}$ smallest value of $\mathbf{A}$;
4 $x_2 =$ the ${N_2}^{th}$ smallest value of $\mathbf{A}$;
5 if $F_{p_j}(x_1) \geq F_{p_j}(x_2)$ then
6     $R^g_{p_j} = x_1$;
7 else
8     $R^g_{p_j} = x_2$;
```

We then use the binary search algorithm to find the optimal reservation number of Gets. Its pseudocode is shown in Algorithm 2.

4 Cost and Latency Efficiency Enhancement

4.1 Coefficient based Data Reallocation

The Get/Put rates of data items may vary over time largely. Therefore, the total Get/Put rate of all data items in a storage datacenter may also vary greatly. In this case, the previous decision on reservation schedule may become obsolete. According to Algorithm 2, the ${N_1}^{th}$ and ${N_2}^{th}$ smallest values of $\mathbf{A}$ may be much smaller than the other values behind them in $\mathbf{A}$ and much larger than the other values before them in $\mathbf{A}$. The total Gets/Puts beyond reservation is $\sum_{A^{t_i}_{p_j} \in \mathbf{A}} Max\{0, A^{t_i}_{p_j} - R_{p_j}\}$, where R_{p_j} is the optimal reservation number of Gets/Puts. Recall that the Gets/Puts beyond reservation are paid in a pay-as-you-go manner, which has a much higher unit price than the reservation price. Therefore, we should reduce the total Gets/Puts beyond reservation in order to minimize the payment cost under Get/Put rate variance. Also, the total number of over reserved Gets/Puts should be reduced, since the over reserved Gets/Puts during billing period t_i, i.e., $R_{p_j} - A^{t_i}_{p_j}$, are not utilized and their reservation cost is wasted. Therefore, to achieve both goals and make the reservation schedule adaptive to the current workload distribution, we need to balance the workloads among all billing periods within the reservation period T, so that the ${N_1}^{th}$ and ${N_2}^{th}$ smallest values (hence R_{p_j}) are close to the other values in $\mathbf{A}$.

In order to balance the workloads between billing periods to get the highest reservation benefit, we propose an algorithm to reallocate the data items in data allocation scheduling. It is conducted after the dominant-cost based data allocation by Algorithm 1 before the real data allocation conduction. It has two strategies i) transferring (i.e., reallocating) a data item between storage datacenters in order to balance both Gets and Puts, and ii) redirecting the Gets among replicas of a data item to balance the Get workloads.

We use Get workload balance as a showcase, and the Put workload balance is handled in the similar way without the Get redirection strategy above. This algorithm needs to select the data items to transfer or to redirect their Gets. In a storage datacenter, it measures the coefficient between the total workload (Gets or Puts) towards this datacenter and the workload towards each data item in it. The coefficient represents the contribution from a data item to the datacenter on the imbalanced Get/Put workloads.

We use $r^{t_k}_{p_j}$ and $v^{d_l,t_k}_{p_j}$ to denote the Get rate towards storage datacenter p_j and data item d_l in p_j during time t_k, respectively. Then, $(r^{t_k}_{p_j} - \frac{R_{p_j}}{t_k})$ represents the over-use or under-user of the reserved Gets per unit time. We then calculate their coefficient (denoted by $\tau^{d_l}_{p_j}$) as

$$\tau^{d_l}_{p_j} = \sum_{t_k \in T} (r^{t_k}_{p_j} - \frac{R_{p_j}}{t_k}) * v^{d_l,t_k}_{p_j}. \quad (19)$$

The coefficient calculates the total workload contribution of d_l while the Gets towards p_j is under or over reservation during all billing periods. A positive coefficient means under reservation, while a negative coefficient means over reservation. Recall that during a billing period t_k, a larger amount of an under (or over) reservation leads to more over pay. Therefore, the d_l with a higher workload during this t_k is more important to be transferred in under reservation or to be maintained in over reservation. Maintaining a data item means keeping the current data allocation for this data item and no further action is needed. Below, we introduce how to reallocate data items to avoid under reservation.

For a storage datacenter p_j, Get cost dominant data items should contribute the majority of the Get workloads. Therefore, we calculate the coefficient for all Get cost dominant data items of p_j and sort them in the descending order of their coefficients. For each data item d_l with a positive coefficient (that contributes to under reservation), d_l needs to be transferred or its Get requests need to be redirected. Recall that the datacenters which can satisfy the SLO Get requirement of d_l requested by c_i are $S^g_{c_i} \cap S^p_{d_l}$. Then, we sort all datacenters in $S^g_{c_i} \cap S^p_{d_l}$ in ascending order of their Get unit price. Similar to Algorithm 1 at Line 14, we select the best datacenter from the top c datacenters with enough Get capacity to transfer Get workloads from p_j. For each datacenter p_k in the list, we measure the total cost (C_t) savings if we transfer d_l's workload in p_j to p_k, which is calculated by $(C^{p_j}_t + C^{p_k}_t) - ({C_t}'^{p_j} + {C_t}'^{p_k})$. $C^{p_j}_t$ and ${C_t}'^{p_j}$ is the total cost of p_j before and after transferring d_l's workload from p_j to p_k. The cost is calculated according to Equation (1) by regarding $D_s = \{p_j, p_k\}$ and D only contains all data items stored in p_j. Similarly, we can calculate $C^{p_k}_t$ and ${C_t}'^{p_k}$, which are the total cost of p_k before and after transferring d_l's workload from p_j to p_k. Here, there is a Transfer cost for the data transfer though it is small. To make the saving cost calculation more accurate, we can include this data Transfer cost into the equation. If several datacenters lead to the same total cost savings, we can choose one further considering locality.

Recall that each data item has a fixed number of β replicas. The storage datacenter p_k with the largest positive savings will be selected to the new host of b_l if p_k does not have a replica of d_l; otherwise, we redirect Get requests from p_j to p_k by reassigning the request serving ratios of p_k and p_j for Gets towards d_l from c_i, that is, $H^{d_l}_{c_i,p_k} = H^{d_l}_{c_i,p_j} + H^{d_l}_{c_i,p_k}$ and $H^{d_l}_{c_i,p_j} = 0$.

For all storage datacenters, if there is no workload transfer, current data allocation has the minimized cost after workload balancing among billing periods; otherwise, the whole process from data item selection to data transferring or Get request redirection is repeated again. This is because the data allocation schedule changed may lead to the change of the data items with positive coefficient, and hence another transfer or redirection may further reduce the payment cost. Algorithm 3 shows the pseudocode of coefficient based data reallocation for Get cost minimization. We then conduct the process of coefficient based data reallocation for Put cost minimization, which has a similar process as Algorithm 3 without Get redirection (Lines 12-15) with all Put cost dominant data items.

Algorithm 3: Coefficient based data reallocation.

```
1  Has_Transferring=True;
2  while Has_Transferring do
3      Has_Transferring=False;
4      for each $p_j \in D_s$ do
5          Create list $L$ including Get dominant items in $p_j$;
6          Sort data items in $L$ in descending order of coefficient;
7          for each $d_l$ in $L$ with a positive coefficient do
8              Sort all storage datacenters in $S^g_{c_i} \cap S^p_{d_l}$ in ascending
               order of Get unit price;
9              Select top $c$ storage datacenters with enough available Get
               capacity to serve Gets from $c_i$ towards $d_l$, i.e.,
               $Max\{v^{d_l,t_k}_{c_i}\}_{t_k \in T} * H^{d_l}_{c_i,p_j}$;
10             if exist another storage datacenter $p_k$ with the largest positive
               cost saving then
11                 Has_Transferring=True;
12                 if $p_k$ has $d_l$'s replica then
13                     $H^{d_l}_{c_i,p_k} = H^{d_l}_{c_i,p_j} + H^{d_l}_{c_i,p_k}$; $H^{d_l}_{c_i,p_j} = 0$;
14                 else
15                     $H^{d_l}_{c_i,p_k} = H_{c_i,p_j}$;
16                     $X^{d_l}_{p_k} = 1$; $X^{d_l}_{p_j} = 0$;
```

4.2 Multicast based Data Transferring

At the beginning of each billing period, there may be a new data allocation schedule and then the data allocation should be deployed. When deploying a new data allocation, for a data item d_l, there may exist multiple data replicas. We can minimize the Transfer cost by creating a new replica from an existing replica of d_l in the previous billing period with minimum Transfer unit price. As shown in Figure 3, there are two replicas of data item d_l in the system, and three new replicas are needed to create in three datacenters. The weight on each edge represents the payment cost to transfer (i.e., replicate) the data replica from a source storage datacenter to a destination storage datacenter. Then, as shown in Figure 3 (a), the minimized payment cost is 18 by transferring replica d^1_l to datacenter p_1 and p_3, and transferring replica d^2_l to p_2. To further save the Transfer cost in replica creation, we propose a multicast based data transferring method. Instead of creating replicas concurrently, our method conducts sequential data transferring. As shown in Figure 3 (b), it creates the new replica in p_2 from d^2_l first, and then creates the two remaining new replicas in p_1 and p_3 from the new replica. Then, the payment is reduced to 8.

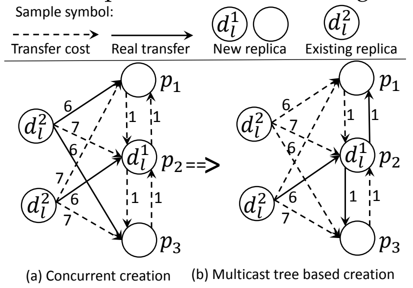


Fig. 3: Multicast based data transferring.

To minimize the Transfer cost in replica creation, we create a multicast based data transferring schedule using a minimum spanning tree [23]. In the tree, the direct edge e denotes a data transfer from the source node to destination, and the weight represents the Transfer cost. The minimum spanning tree is acyclic with the minimum sum of the path weights when the replicas are transferred from the parent to its children in the top-down manner in order.

However, the minimum spanning tree only has one tree root while we have β initially existing data replicas that can be used as the tree root. One method to create a tree with the minimum Transfer cost is to use each existing replica as a tree root to build a minimum spanning tree that embraces all replica nodes. Then, we choose the tree that generates the minimum total edge weights, i.e., minimum total Transfer cost. However, this method may not maximally reduce the Transfer cost because these existing replicas actually can cooperatively create new replicas to further minimize the Transfer cost. For example, in Figure 3, another new replica only costs 1 by replicating from d^1_l while costs 6 by replicating from d^2_l, then it should be replicated from d^1_l, while the replica in p_2 is replicated from d^2_l. Thus, to leverage the minimum spanning tree, we create a virtual tree root (i.e., source replica node) that represents all initially existing data replicas. To create the tree, we first build a graph that connects all replica nodes using edges with edge weight representing Transfer cost. The weight from the virtual source replica node to another replica node p_j in the graph is the minimum weight among the edges from each initially existing data replicas to node p_j. For example, in Figure 3 (a), the weight between the newly created replica in p_2 and existing replica nodes is 6, and the actual transferring to the p_2 is from d^2_l. After the graph is built, we gradually add nodes to the minimum spanning tree. We first add the virtual source replica node to the tree. We then find the edge with the minimum weight connecting one of the nodes in the tree to one of the nodes outside of the tree, and add the edge and the node outside to the tree. The process continues until all nodes are in this minimum spanning tree. Based on the constructed minimum spanning tree, we create all new replicas in a top-down sequence so that the Transfer cost is minimized. For example, the replica in p_2 is created from d^2_l, and then p_2 transfers the replica to p_1 and p_3. Note that the cost saved here is Transfer monetary cost, which is part of Transfer cost dealt with previously. It is not related to the calculation overhead to figure out the multicast based data transferring schedule.

4.3 Request Redirection based Congestion Control

The Get rates of a customer's web application may vary over time. The target storage datacenter may be overloaded due to a burst of Get workloads. For example, the visit frequency of the Clemson football game data in the Clemson area suddenly increases when Clemson football team entered the championship game. To avoid congestions in order to guarantee the Get SLO, we can redirect the Get workloads to other datacenters. Recall that there are β replicas for each data item in order to guarantee a promised data availability. In the redirection method, when storage datacenter p_j is overloaded (i.e., total Get workload exceeds its Get capacity) at a certain time during a billing period t_k, the customer datacenter redirects the data requests towards other datacenters that store the replicas of the requested data. Recall that $H^{d_l}_{c_i,p_j}$ represents the ratio of requests targeting d_l's replica stored in datacenter p_j. Therefore, by adjusting the request ratio, we can redirect the requests among replicas to avoid congestions. Recall that $A^{t_k}_{p_j}$ denotes the total number of expected Gets towards storage datacenter p_j during billing period t_k (which is the data allocation schedule that minimizes the payment cost). We use ${A'}^{t_k}_{p_j}$ to denote the number of Gets that have been resolved by storage datacenter p_j during t_k. In order to fully utilize the expected usage of each datacenter and reduce the Get cost, we redirect the request with a probability to another datacenter p_m having d_l's replica according to its remaining expected usage calculated by $A^{t_k}_{p_m} - {A'}^{t_k}_{p_m}$. Specifically, we distribute the excess workload on p_j among the replica datacenters based on the number of their unused expected Gets. Since all Gets in $A^{t_k}_{p_m}$ are within the expected payment as C_t, more unused Gets indicate a larger expected payment that has not been fully used, so that the datacenter needs to be redirected more Gets (i.e., reassigned larger Get request serving ratio).

We use $D_s(d_l, p_j^-)$ to denote the set of all storage datacenters except p_j that store d_l. We use $v^{d_l,t_k}_{c_i}$ and ${v'}^{d_l,t_k}_{c_i}$ to denote the expected Get rate and the real burst Get rate of d_l. Then, the excess workload towards p_j that needs to be redirected equals $H^{d_l}_{c_i,p_j} * (1 - \frac{v^{d_l,t_k}_{c_i}}{{v'}^{d_l,t_k}_{c_i}})$. Therefore, we can derive the new request ratio of storage datacenter $p_m \in D_s(d_l, p_j^-)$, denoted by ${H'}^{d_l}_{c_i,p_m}$:

$${H'}^{d_l}_{c_i,p_m} = H^{d_l}_{c_i,p_m} + \frac{H^{d_l}_{c_i,p_j} * (1 - \frac{v^{d_l,t_k}_{c_i}}{{v'}^{d_l,t_k}_{c_i}}) * (A^{t_k}_{p_m} - {A'}^{t_k}_{p_m})}{\sum_{p_n \in D_s(d_l,p_j^-)} (A^{t_k}_{p_n} - {A'}^{t_k}_{p_n})},$$ where

$H^{d_l}_{c_i,p_m}$ is the request ratio without congestions, and the reassigned Get request serving ratio of a replica datacenter is proportional to its number of unsolved Gets. When the p_j is not overloaded, the request ratios roll back to their previous values.

With the request redirection, a data request is transferred from a datacenter with overloaded Get workload to a datacenter with sufficient Get capacity to handle it, which helps meet the SLO requirements though it cannot be guaranteed.

5 System Infrastructure

In this section, we introduce the infrastructure to conduct the previously introduced *DAR* algorithms. It collects the information of scheduling inputs, calculates the data allocation schedule and conducts data allocation. As shown in Figure 4, *DAR*'s infrastructure has one master server and multiple agent servers, each of which is associated with a customer datacenter. Agent servers periodically measure the parameters needed in the schedule calculation and conducted by the master.

In cloud, the reservation is made at the beginning of reservation time period T and remains the same during T. Due to the time-varying feature of the inter-datacenter latency and Get/Put rates, the master needs to periodically calculate the allocation schedule after each billing period t_k and reallocate the data accordingly if the new schedule has a smaller cost or the current schedule cannot guarantee

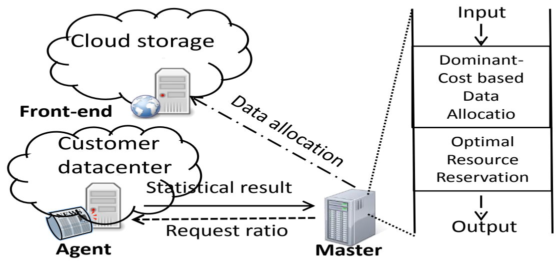


Fig. 4: Overview of DAR's infrastructure

the SLOs. Recall that new datacenters may appear when calculating a new schedule. Thus, a customer will not be rapped with obsolete providers and can always choose the optimal CSPs for the minimum cloud storage service cost, which avoids the vendor lock-in problem. Therefore, the master executes the optimal resource reservation algorithm and makes a reservation only before t_1, and then updates T to $T \backslash \{t_k\}$ and executes the dominant-cost based data allocation algorithm after each t_k.

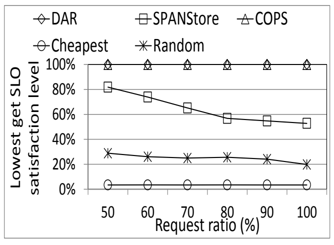
During each t_k, for the schedule recalculation, the master needs the latency's CDF of Get/Put ($F^g_{c_i,p_j}(L_g)(x)$ and $F^p_{c_i,p_j}(L_p)(x)$), the size of each d_l (s_{d_l}), and the data's Get/Put rate from c_i ($v^{d_l,t_k}_{c_i}$ and $u^{d_l,t_k}_{c_i}$). Each agent in each customer datacenter periodically measures and reports these measurements to the master server. The *DAR* master calculates the data allocation schedule and sends the updates of the new data allocation schedule to each customer datacenter. This way, our method can accommodate for the dynamically changing conditions. Specifically, it measures the differences of the data item allocation between the new and the old schedules and notifies storage datacenters to store or delete data items accordingly. In reality, billing time period t_k (e.g., one month) may be too long to accurately reflect the variation of inter-datacenter latency and Get/Put rates dynamically in some applications. In this case, *DAR* can set t_k to a relatively small value with the consideration of the tradeoff between the cost saving, SLO guarantee and the *DAR* system overhead.

6 Performance Evaluation

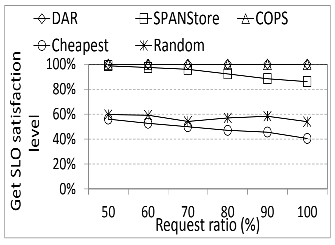
We conducted trace-driven experiments on Palmetto Cluster [24] with 771 8-core nodes and on real clouds (i.e., Amazon S3, Windows Azure Storage and Google Cloud Storage). We first introduce the experiment settings on the cluster.

Simulated clouds. We simulated geographically distributed datacenters in all 25 cloud storage regions in Amazon S3, Microsoft Azure and Google cloud storage [1], [3], [2]; each region has two datacenters simulated by two nodes in Palmetto. The distribution of the inter-datacenter Get/Put latency between any pair of cloud storage datacenters follows the real latency distribution as in [9]. The unit prices for Storage, Get, Put and Transfer in each region follows the prices listed online. We assumed that the reservation price ratio (α) follows a bounded Pareto distribution among datacenters with a shape as 2 and a lower bound and an upper bound as 53% and 76%, respectively [8].

Customers. We simulated ten times of the number of all customers listed in [1], [3], [2] for each cloud service provider. The number of customer datacenters for each customer follows a bounded Pareto distribution, with an upper bound, a lower bound and a shape as 10, 8 and 2, respectively. As in [9], in the SLOs for all customers, the Get deadline is restricted to $100ms$ [9], 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

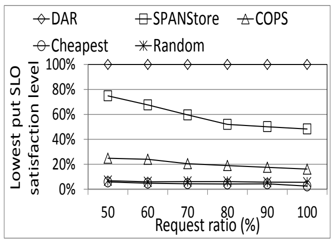


(a) In simulation

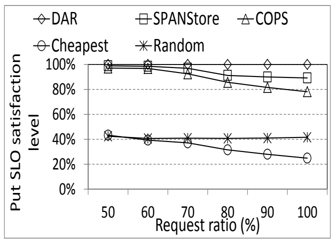


(b) In real clouds

Fig. 5: Get SLO guarantee performance.



(a) In simulation



(b) In real clouds

Fig. 6: Put SLO guarantee performance.

400ms for an over-continent customer. The minimum number of replicas of each data item was set to $\beta = 3$ [8] unless otherwise specified. The size of the aggregated data of a customer was randomly chosen from $[0.1TB, 1TB, 10TB]$ as in [9]. The number of aggregated data items of a customer follows a bounded Pareto distribution with a lower bound, an upper bound and a shape as 1, 30000 and 2 [25].

Get/put operations. The percentage of data items requested by each customer datacenter follows a bounded Pareto distribution with an upper bound, lower bound and shape as 20%, 80% and 2, respectively. Each aggregated data item is formed by data objects and the size of each requested data object was set to 100KB [9]. The Put rate follows the publicly available wall post trace from Facebook New Orleans networks [21]. We set the Get rate of each data object using the same way introduced in Section 3.1. Facebook is able to handle 1 billion/10 million Gets/Puts per second [22], and has ten datacenters over the U.S. Accordingly, we set the Get and Put capacities of each datacenter in an area to 1E8 and 1E6 Gets/Puts per second, respectively. Whenever a datacenter is overloaded, the Get/Put operation was repeated once again. We set the billing period (t_k) to 1 month and set the reservation time to 3 years [8]. We computed the cost and evaluated the SLO performance in 3 years in experiments. For each experiment, we repeated 10 times and reported the average performance.

Real clouds. We also conducted small-scale trace-driven experiments on real-world CSPs including Amazon S3, Windows Azure Storage and Google Cloud Storage. We simulated one customer that has customer datacenters in Amazon EC2's US West (Oregon) Region and US East Region [26]. Unless otherwise indicated, the settings are the same as before. Considering the small scale and the cost of using the cloud resource, the number of data items was set to 1000, the size of each data item was set to 100MB, and β was set to 2. The datacenter in each region requests all the data objects. We set the Put deadline to 200ms. One customer's Gets and Puts operations cannot generate enough workload to reach the real Get/Put rate capacity of each datacenter. We set the capacity of a datacenter in each region of all CSPs to 40% 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.

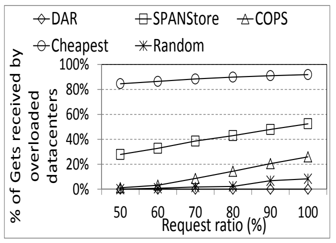
We compared *DAR* with the following methods: (1) *SPANStore* [9], which is a storage over multiple CSPs' datacenters to minimize cost supporting SLOs without considering capacity limitations and reservations; (2) *COPS* [27], which allocates requested data in a datacenter with the shortest latency to the customer datacenter but does not consider the Put latency minimization; (3) *Cheapest*, in which the customer selects the datacenters with the cheapest cost to store each data item without considering SLOs and reservations; (4)*Random*, in which the customer randomly selects datacenters to allocate each data item.

6.1 Comparison Performance Evaluation

In order to evaluate the SLO guarantee performance, we measured the lowest SLO satisfaction levels of all customers. The Get/Put SLO satisfaction level of a customer, Q^g/Q^p, is calculated according to Equation (7) with $q^g_{t_k}/q^p_{t_k}$ as the actual percentage of Gets/Puts within deadline during t_k. We varied each data item's Get/Put rate from 50% to 100% (named as request ratio) of its original rate, with the step size of 10%.

Figures 5(a) and 5(b) show the (lowest) Get SLO satisfaction level of each system versus the request ratio on the simulation and real CSPs, respectively. We see that the lowest satisfaction level follows 100%=*DAR*=*COPS*>*SPANStore*>*Random*>*Cheapest*. *DAR* considers both the Get SLO and capacity constraints, thus it can supply a Get SLO guaranteed service. *COPS* always chooses the storage datacenter with the smallest latency. *SPAN- Store* always chooses the storage datacenter with the Get SLO consideration. However, since it does not consider datacenter capacity, a datacenter may become overloaded and hence may not meet the latency requirement. Thus, it cannot supply a Get SLO guaranteed service. *Random* uses all storage datacenters to allocate data, and the probability of a datacenter to become overloaded is low. However, since it does not consider the Get SLO, it may allocate data to datacenters far away from the customer datacenters, which leads to long request latency. Thus, *Random* generates a smaller (lowest) Get SLO satisfaction level than *SPANStore*. *Cheapest* does not consider SLOs, and stores data in a few datacenters with the cheapest price, leading to heavy datacenter overload. Thus, it generates the worst SLO satisfaction level. The figures also show that for both *SPANStore* and *Random*, the Get SLO satisfaction level decreases as the request ratio increases. This is because a higher request ratio leads to higher request load on an overloaded datacenter, which causes a worse SLO guaranteed performance due to the repeated requests. The figures indicate that *DAR* can supply a Get SLO guaranteed service with SLO and capacity awareness.

Figures 6(a) and 6(b) show the lowest Put SLO satisfaction level of each system versus the request ratio on the simulation and real CSPs, respectively. We see that the lowest SLO satisfaction level follows 100%=*DAR*>*SPANStore*>*COPS*>*Random*>*Cheapest*. *DAR* considers both Put SLOs and datacenter Put capacity, so it supplies SLO guaranteed service for Puts. Due to the same reason as Figure 5(a), *SPANStore* generates a smaller Put SLO satisfaction level. *COPS* allocates data into datacenters nearby without considering the Put latency minimization, and the Put to other datacenters except the datacenter nearby may introduce a long delay. Thus, *COPS* cannot supply a Put SLO guaranteed service, and generates a lower Put SLO satisfaction level than *SPANStore*. *Random*


(a) In simulation

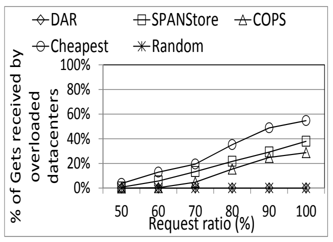
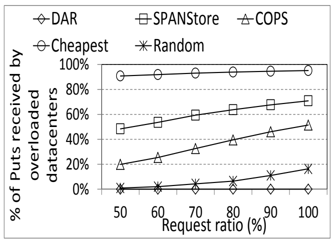

(b) In real clouds

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


(a) In simulation

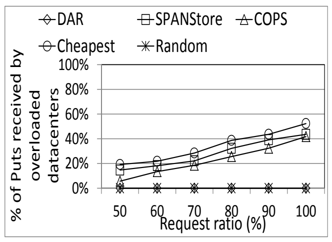

(b) In real clouds

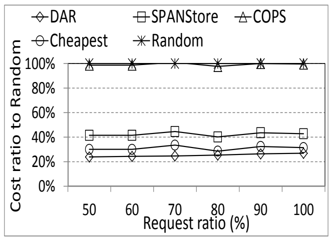
Fig. 8: Percent of Puts received by overloaded datacenters.

and *Cheapest* generate smaller Put SLO satisfaction levels than the others and the level of *SPANStore* decreases as the request ratio increases due to the same reasons as in Figure 5(a). The figures indicate that *DAR* can supply a Put SLO guaranteed service while others cannot. Note that the relative performances between different methods in the simulation and in the real cloud experiment are consistent with each other, but the magnitudes of the experimental results of a method in the simulation and real cloud experiment are not comparable due to the different experimental settings in the two environments. Particularly, the real cloud experiment has a much smaller scale and all datacenters are in the U.S., which generate low inter-datacenter communication delay, while the simulation has a much larger scale and all datacenters spread all over the world, which generate high inter-datacenter communication delay.

Figures 7(a) and 7(b) show percentage of Gets received by overloaded datacenters versus the request ratio on the simulation and real CSPs, respectively. We see that the percentage values follows 0%=*DAR*<*Random*<*COPS*<*SPANStore*<*Cheapest*. Due to the capacity-awareness, *DAR* can always avoid datacenter overloads, thus it has no requests received by overloaded datacenters. *Random* allocates data items over all storage datacenters randomly, thus it has a smaller probability of making storage datacenters overloaded. Even with a large request ratio, since *Random* randomly assigns data items to all datacenters, which can be considered as uniform workload distribution among all datacenters, the datacenters are less likely to be overloaded. *COPS* always allocates data items to the datacenters near data requesters without considering their capacities, thus it has a higher probability of making datacenters overloaded. *SPANStore* considers both cost and latency deadlines but neglects datacenter capacity, so it has fewer datacenter choices than *COPS* for allocating a data item, which leads to a higher probability of making datacenters overloaded than *COPS*. By biasing a limited number of cheapest datacenters, *Cheapest* produces the highest probability of making datacenters overloaded. From the figures, we also see, except *DAR* (and *Random* in real CSPs), the percentage value of each system increases as the request ratio increases. This is because heavy load leads to a higher probability of making storage datacenters overloaded.

Figures 8(a) and 8(b) show the percentage of Puts received by overloaded datacenters on the simulation and real CSPs, respectively. They show the same trends and orders between all systems as Figure 7(a) due to the same reasons. All these figures indicate that *DAR* can effectively avoid overloading datacenters using capacity-aware data allocation, which ensures the Get/Put SLOs, while other systems cannot.

Figure 9(a) shows the payment costs of all systems compared to *Random* by calculating the ratio of the each system's cost to the cost of *Random* in the simulation.


(a) In simulation

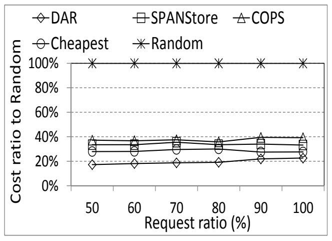

(b) In real clouds

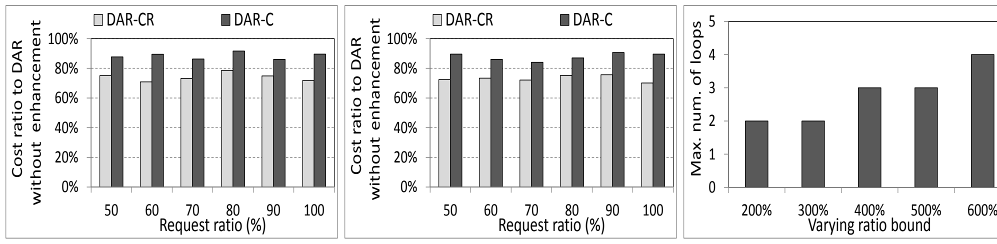
Fig. 9: Cost minimization performance without capacity awareness.

The figure shows that the cost follows *COPS*≈*Random*> *SPANStore*>*Cheapest*>*DAR*. Since both *COPS* and *Random* do not consider cost, they produce the largest cost. *SPANStore* selects the cheapest datacenter within the deadline constraints, thus it generates a smaller cost than systems without cost considerations. However, it produces a larger cost than *Cheapest*, which always chooses the cheapest datacenter in all datacenters. *DAR* generates the smallest cost because it chooses the cheap datacenter under SLO constraints and makes a reservation to further maximally save cost. The figure also shows that the cost of *DAR* increases as the request ratio increases, but it always generates the smallest cost. This is because when the datacenters with the cheapest price under constraints are used up, the second optimal candidates will be chosen to allocate the remaining data. All others do not consider the capacities of datacenter, and hence violate the Get/Put SLO by making some datacenters overloaded. Figure 9(b) shows the payment costs of all systems compared to *Random* on the real CSPs. It shows the same order and trends of all systems as Figure 9(a) due to the same reason, except that *COPS*<*Random*. This is because the storage datacenters nearest to the customer datacenters happen to have a low price. The figures indicate that *DAR* generates the smallest payment cost in all systems.

6.2 Performance of Cost and Latency Efficiency Enhancement

6.2.1 Coefficient Based Data Reallocation Method

In this section, we present the performance of cost and latency efficiency enhancements. We first measured the effectiveness of coefficient based data reallocation method. Recall that the data reallocation method aims to balance the workloads among all billing periods in order to minimize the payment cost by maximizing the reservation benefit. Thus, we varied the Get/Put rate of each data item over billing periods by setting the Get/Put rate to $x\%$ of the Get/Put rate in the previous billing period, where x is called *varying ratio* and was randomly chosen from [50, 200] [9]. We use *DAR-CR* to denote *DAR* with the coefficient based data reallocation method. Recall that this method has a recursive process as shown at Line 2 in Algorithm 3. In order to show the effectiveness of the recursive process, we compared *DAR-CR* without the recursive process by conducting the



(a) Cost reduction in simulation (b) Cost reduction in real clouds (c) Maximum number of loops

Fig. 10: Effectiveness and efficiency of coefficient based data reallocation.

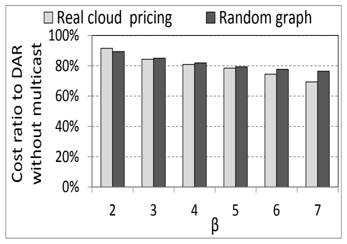


Fig. 11: Cost reduction of multicast based data transferring.

algorithm from Line 4 to Line 16 only once, denoted by *DAR-C*. Figures 10(a) and 10(b) show the cost ratio of *DAR-CR* and *DAR-C* compared with *DAR* without the coefficient based data reallocation method versus the request ratio in simulation and real clouds, respectively. They show that the cost ratio of all methods follows *DAR-CR*<*DAR-C*<100%. Both *DAR-CR* and *DAR-C* generate cheaper payment cost than *DAR* without the data reallocation method. This is because the data reallocation balances the Get/Put workloads of each storage datacenter in different billing periods during T, so that R_{p_j} is close to the $\max\{\mathbf{A}\}$. Therefore, more Gets/Puts are paid in the reservation manner, and the payment cost is reduced. *DAR-CR* generates a smaller payment cost than *DAR-C*. This is because once the data allocation schedule is changed, the aggregated Get/Put rates of different billing periods are changed. Therefore, some other data items need to be reallocated in order to balance the workloads more. The figures demonstrate that the coefficient based data reallocation method is effective in balancing the workloads of storage datacenters among billing periods in order to minimize the payment cost, and the recursive process can balance the workloads more.

Recall that the coefficient based data reallocation is a recursive process. We measured the convergence efficiency by the maximum number of loops conducted in Algorithm 3. Figure 10(c) shows the number of loops versus different upper bound of varying ratio increasing from 200% (as default) to 600% with a step size of 100% in both simulation and on the real cloud testbed. Since the results are the same, we only present one figure. It shows that the number of loops increases when the upper bound of the varying ratio increases. This is because a larger upper bound leads to more unbalanced Get/Put rates among billing periods, which needs more loops to converge to the workload balance. It shows that the number of loops is no larger than 4. It was indicated that the Get/Put rate varying ratio for the data items in Facebook is bounded by 200% [9]. From the figure, we can see that with varying ratio bounded by 200%, there are 2 iteration of data selection and transfer or redirection as shown in Lines 3-16 in Algorithm 3. Therefore, the figure demonstrates that the coefficient based data reallocation method converges fast.

6.2.2 Multicast Based Data Transferring Method

We then measured the performance of the multicast based data transferring method. The method is efficient in replicating multiple replicas, so we measured its performance in simulation, since the real cloud experiment has a small scale. In this experiment, we initially simulated β replicas for each data item that were randomly distributed among all storage datacenters, and then in the next period, we created β replicas for each data item in randomly selected storage datacenters. We simulated two different scenarios. In *Real cloud pricing*, we simulated all storage datacenters and their transfer pricing policies in the default setting. In *Random graph*, we simulated the same number of storage datacenters, and the unit Transfer price from one datacenter to the other is randomly chosen from [0,2.5]. Figure 11 shows the Transfer cost (excluding all other costs) ratio of *DAR* with the multicast based data transferring method compared to *DAR* without this method in the two different scenarios. It shows that this method can save at least 8.5% and up to 30.7% in both scenarios. This is because this method uses newly created data replicas to transfer the data items to other storage datacenters to create new replicas, which have cheaper Transfer prices than current existing replicas. Therefore, the multicast based transferring method can save the Transfer cost. The figure also shows that the cost ratio decreases when β decreases. This is because more new replicas lead to a higher probability of having a cheaper Transfer price to create new replicas than current existing replicas. The figure demonstrates that the multicast based data transferring method is effective in minimizing the Transfer cost.

6.2.3 Request Redirection Based Congestion Control Method

Next, we measured the performance of request redirection based congestion control method in guaranteeing SLOs. We use *DAR-Redirection* to denote *DAR* with this redirection method. The redirection method is more effective when there are more replicas to redirect requests to. Therefore, we measured its performance in simulation instead of on the real cloud testbed due to its small scale. In this experiment, we randomly chose the varying ratio of Get rate of a data item from $[1-v, 1+v]$, where v is the largest varying ratio and was varied from 10% to 50% with a step size of 10%. The varying ratio is constant during one experiment and the experiments are repeated with various varying ratios. Figure 12 shows the lowest Get SLO satisfaction level of all customers of different methods versus the largest varying ratio. From the figure, we can see that *DAR-Redirection* can guarantee the Get SLO while *DAR* without the redirection method cannot guarantee the Get SLO. This is because the storage datacenters can be overloaded when some data items have larger request rates than expected, and without the request redirection, the requests cannot be redirected to underloaded datacenters so they cannot be resolved in time. The figure indicates that the request redirection based congestion control method can help supply a Get SLO guaranteed service under the varying Get rate during a period. Figure 12 also shows the cost ratio of *DAR-Redirection* compared to *DAR* that randomly selects a storage datacenter to redirect a request. It shows

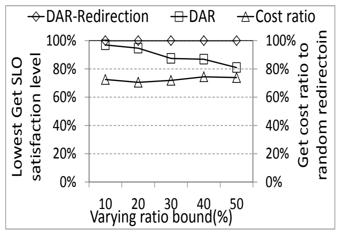
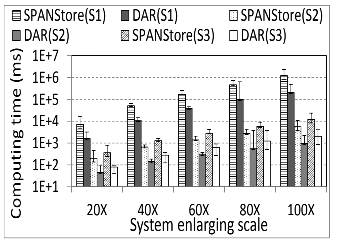
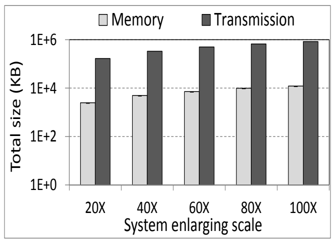


Fig. 12: Congestion control and additional Get cost minimization.



(a) Computing time



(b) Data size

Fig. 13: System overhead with different scales in simulation.

that the cost ratio is always under 100%, that is, the request redirection always saves cost. This is because the random selection may select a datacenter that already uses up its expected Gets but does not fully utilize the unused expected Gets or even reserved Gets in other datacenters. The redirection method considers the difference between the expected number of Gets and the number of currently used Gets in order to fully utilized the expected Gets to minimize the Get cost. As a result, *DAR-Redirection* saves cost than *DAR* with random selection. This figure demonstrates that the request redirection based congestion control method can reduce the Get cost by fully utilizing the expected Gets.

6.3 System Overhead

In this experiment, we measure the system overhead of SLO-aware and cost minimization systems (i.e., *DAR* and *SPANStore*) on the testbed with a large customer scale in three scenarios. In this first scenario, we enlarge the number of storage datacenters and the number of customers by x times, where x (called system enlarging scale) was varied from 20 to 100 times, with a step size as 20; in the second scenario, we enlarge the number of customers by x times, and in order not to overload all datacenters, we also enlarge the capacity of all datacenters by x times; in the third scenario, we only enlarge the number of datacenters by x times. We use the system name and scenario number (e.g., S_1 dentes the first scenario) to distinguish the different results. For example, we use $DAR(S_1)$ to denote the results of *DAR* in the first scenario.

Figure 13(a) shows the 95th percentile, median and 5th percentile of computing time of *DAR* and *SPANStore* in all three different scenarios, which is the time period to compute the data allocation schedule among datacenters. It shows that in each scenario, all percentiles of computing time follows *DAR*<*SPANStore*. *DAR* uses the dominant-cost based data allocation algorithm, which runs faster than solving the integer program in *SPANStore*. Also, though *DAR* has an additional optimal reservation algorithm, since it runs fast, *DAR* still needs smaller computing time than *SPANStore*. This figure indicates the higher computing efficiency of *DAR* compared to *SPANStore*.

Figure 13(b) shows the memory consumption at each customer datacenter to store the table for request allocation among all replicas, and the total size of communication packets between *DAR* agents and the master. Since these kinds of overhead is only proportional to number of data items, not the number of storage datacenters. We only measure the performance of *DAR* under the first scenario, but instead we enlarged the number of data items for each customer by x times instead of the number of customers. We see these two types of overheads increase as the system scale increases, since a larger number of data items per customer increases the table size. We observe that the memory usage is bounded by 8MB, which is very small, and the total packet size is bounded by 500MB per customer with maximum 10 customer datacenters, which is small according to current bandwidth among datacenters. The figure indicates that the overhead of *DAR* is small and tolerable.

7 Related Work

Deploying on multiple clouds. SafeStore [28], RACS [7] and DepSky [29] 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. In [30], an application execution platform across multiple CSPs was proposed. COPS [27] and Volley [17] automatically allocate user data among datacenters in order to minimize user latency. Blizzard [31] is a high performance block storage for clouds, which enables cloud-unaware applications to fast access any remote disk in clouds. Unlike these systems, *DAR* additionally considers both SLO guarantee and cost minimization for customers across multiple cloud storage systems.

Minimizing cloud storage cost. In [10], [11], [12], cluster storage automate configuration methods are proposed to use the minimum resources needed to support the desired workload. Adya *et al.* [13] proposed a file system with high availability and scalability and low cost, named as Farsite. It depends on randomized replication to achieve data availability, and minimizes the cost by lazily propagating file updates. None of the above papers study the cost optimization problem for geo-distributed cloud storage over multiple providers under SLO constraints. SPANStore [9] is a key-value storage over multiple CSPs' datacenters to minimize cost and guarantee SLOs. However, it does not consider the capacity limitation of datacenters, which makes its integer program a NP-hard problem that cannot be solved by its solution. Also, SPANStore does not consider resource reservation to minimize the cost. *DAR* is advantageous in that it considered these two neglected factors and effectively solves the NP-hard problem for cost minimization.

Pricing models on clouds. There are several works studying resource pricing problem for CSPs and customers. In [32], [33] and [34], 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.* [35] formulated the pricing competition of CSPs and resource competition of cloud service customers as a concave game. The solution enables the customers to reduce their payments while receive a satisfied service. Different from all these studies, *DAR* focuses on the cost optimization for a customer deploying geo-distributed cloud storage over multiple cloud storage providers with SLO constraints.

Improving network for SLO guarantee. Several works [36], [37], [38], [39] have been proposed to schedule network flows or packages to meet deadlines or achieve high network throughput in datacenters. All these papers focus on SLO ensuring without considering the payment cost optimization.

8 Conclusion

This work aims to minimize the payment cost of customers while guarantee their SLOs by using the worldwide distributed datacenters belonging to different CSPs with different resource unit prices. We first modeled this cost minimization problem using integer programming. Due to its NP-hardness, we then introduced the *DAR* system as a heuristic solution to this problem, which includes a dominant-cost based data allocation algorithm among storage datacenters and an optimal resource reservation algorithm to reduce the cost of each storage datacenter. We also proposed several enhancement methods for *DAR* to further reduce the payment cost and service latency including i) coefficient based data reallocation, ii) multicast based data transferring, and iii) request redirection based congestion control. *DAR* also incorporates an infrastructure to conduct the algorithms. Our trace-driven experiments on a testbed and real CSPs show the superior performance of *DAR* for SLO guaranteed services and payment cost minimization in comparison with other systems. Since more replicas of a more popular data item can help relieve more loads from overloaded datacenters, in our future work, we will study how to adjust the number of replicas of each data item to further improve the performance of SLO conformance. Further, we will conduct experiments against varying workload conditions and using other traces.

Acknowledgements

This research was supported in part by U.S. NSF grants ACI-1719397 and CNS-1733596, and Microsoft Research Faculty Fellowship 8300751. An early version of this work was presented in the Proceedings of IEEE ICDCS 2016 [40].

References

[1] Amazon S3. http://aws.amazon.com/s3/, [accessed in July 2015].

[2] Microsoft Azure. http://www.windowsazure.com/, [accessed in July 2015].

[3] Goolge Cloud storage. https://cloud.google.com/products /cloud-storage/, [accessed in July 2015].

[4] H. Stevens and C. Pettey. Gartner Says Cloud Computing Will Be As Influential As E-Business. Gartner Newsroom, Online Ed., 2008.

[5] R. Kohavl and R. Longbotham. Online Experiments: Lessons Learned, 2007. http://exp-platform.com/Documents/IEEE Computer2007OnlineExperiments.pdf, [accessed in July 2015].

[6] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannona, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!s Hosted Data Serving Platform. In *Proc. of VLDB*, 2008.

[7] A. Hussam, P. Lonnie, and W. Hakim. RACS: A Case for Cloud Storage Diversity. In *Proc. of SoCC*, 2010.

[8] Amazon DynnamoDB. http://aws.amazon.com/dynamodb/, [accessed in July 2015].

[9] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services. In *Proc. of SOSP*, 2013.

[10] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. A. Becker-Szendy, R. A. Golding, A. Merchant, M. Spasojevic, A. C. Veitch, and J. Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-Scale Storage Systems. *ACM Trans. Comput. Syst.*, 2001.

[11] 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.

[12] 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.

[13] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. of OSDI*, 2002.

[14] J. Dean. Software Engineering Advice from Building Large-Scale Distributed Systems. http://research.google.com/people /jeff/stanford-295-talk.pdf, [accessed in July 2015].

[15] X. Wu, D. Turner, C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. In *Proc. of SIGCOMM*, 2012.

[16] Service Level Agreements. http://azure.microsoft.com/en-us /support/legal/sla/, [accessed in July 2015].

[17] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Proc. of NSDI*, 2010.

[18] G. Liu, H. Shen, and H. Chandler. Selective Data Replication for Online Social Networks with Distributed Datacenters. In *Proc. of ICNP*, 2013.

[19] D. Borthakur, J. Gray, J. S. Sarma, 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.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[21] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the Evolution of User Interaction in Facebook. In *Proc. of WOSN*, 2009.

[22] 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 NSDI*, 2013.

[23] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 1994.

[24] Palmetto Cluster. http://http://citi.clemson.edu/palmetto/, [accessed in July 2015].

[25] 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, [accessed in July 2015].

[26] Amazon EC2. http://aws.amazon.com/ec2/, [accessed in July 2015].

[27] 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.

[28] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A Durable and Practical Storage System. In *Proc. of ATC*, 2007.

[29] A. N. Bessani, M. Correia, B. Quaresma, F. Andr, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *TOS*, 2013.

[30] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *Proc. of NSDI*, 2012.

[31] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *Proc. of NSDI*, 2014.

[32] 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.

[33] Y. Song, M. Zafer, and K.-W. Lee. Optimal Bidding in Spot Instance Market. In *Proc. of INFOCOM*, 2012.

[34] D. Niu, B. Li, and S. Zhao. Quality-assured Cloud Bandwidth Auto-scaling for Video-on-Demand Applications. In *Proc. of INFOCOM*, 2012.

[35] H. Roh, C. Jung, W. Lee, and D. Du. Resource Pricing Game in Geo-Distributed Clouds. In *Proc. of INFOCOM*, 2013.

[36] C. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. of SIGCOMM*, 2012.

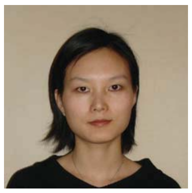
[37] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). In *Proc. of SIGCOMM*, 2012.

[38] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *Proc. of CoNEXT*, 2010.

[39] 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.

[40] G. Liu and H. Shen. Minimum-cost Cloud Storage Service Across Multiple Cloud Providers. In *Proc. of ICDCS*, 2016.

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.



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 the CS Department at the University of Virginia. Her research interests include distributed computer systems and computer networks, cloud computing and cyber-physical systems. She is a Microsoft Faculty Fellow of 2010, a senior member of the IEEE and a member of the ACM.