Minimum-cost Cloud Storage Service Across Multiple Cloud Providers

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

***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 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 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.**

I. Introduction

Cloud storage (e.g., Amazon S3 [1], Microsoft Azure [2] and Google Cloud Storage [3]) is emerging as a popular commercial service. Each cloud service provider (CSP) provides a worldwide data storage service (including Gets and Puts) using its geographically distributed datacenters. In order to save the capital expenditures to build and maintain the hardware infrastructures and avoid the complexity of managing the datacenters, more and more enterprisers shift their data workloads to the cloud storage [4].

Web applications, such as online social networks and web portals, provide services to clients all over the world. The data access delay and availability are important to web applications, which affect cloud customers' incomes. For example, experiments at the Amazon portal [5] demonstrated that a small increase of 100ms in webpage presentation 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 allocated to datacenters near the clients, which requires worldwide distribution of data replicas. Also, inter-datacenter data replication 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 be free to switch to the optimal vendor due to prohibitively high switching costs. This 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.

Different datacenters of a CSP or different CSPs offer different prices for Storage, data Gets/Puts and Transfers. For example, 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 has data d_j with a large storage size and few Gets and data d_i which is read-intensive. Then, to reduce the total payment cost, the application should store data d_j into Amazon S3, and stores data d_i into Windows Azure in the US East region. Besides the different prices, the pricing manner is even more complicated due to two charging formats: pay-as-you-go and reservation. 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?*

Though many previous works [8–10] focus on finding the minimum resource to support the workload to reduce cloud storage cost in a single CSP, there are few works that studied cloud storage cost optimization across multiple CSPs with different prices. SPANStore [11] 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 since datacenter network overload occurs frequently [12, 13]. The integer program used to create a data allocation in [11] becomes NP-hard, if it is modified with capacity-awareness, which however cannot be easily resolved. As far as we know, our work is the first that provides minimum-cost cloud storage service across multiple CSPs with the consideration of resource reservation and datacenter capacity limits.

To handle the above-stated two challenges, we propose a geo-distributed cloud storage system for Data storage and request Allocation and resource Reservation across multiple CSPs (*DAR*). It transparently helps customers to minimize their payment cost while guaranteeing their SLOs. We summarize our contributions below:

- We have modeled the cost minimization problem under multiple constraints using 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 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.

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 II depicts the cost minimization problem. Sections III and IV present the design and infrastructure of *DAR*. Section V presents the trace-driven experimental results. Section VI presents the related work. Section VII gives conclusion with remarks on our future work.

II. Problem Statement

A. Background

We call a datacenter that operates a customer's application a *customer datacenter* of this customer. 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 $dc_i \in D_c$ 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 $dp_j \in D_s$ 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. The cloud storage customers need data request (Puts/Gets) deadlines for their applications, and need to avoid the data request failures. 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 [14] 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 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]. 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 during billing period t_k (e.g., one month).

The CSPs charge the customers by the usage of three different types of resources: the storage measured by the data size stored in a specific region, the data transfer to other datacenters operated by the same or other CSPs, and the number of Get/Put operations on the data [15]. 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) [15]. 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 main notations used in the paper in Table I.

B. 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 integer programming. We first set up the objective of payment minimization. We then form the constrains including satisfying SLO guarantee, data availability, and datacenter capacity. Finally, we formulate the problem with the object and constraints.

Payment minimization objective. We aim to minimize the total payment cost for a customer (denoted as C_t). It is calculated as

$$C_t = C_s + C_c + C_g + C_p, \quad (1)$$

Table I: Notations of inputs and outputs in data allocation.

Input	Description	Input	Description
D_c	customer datacenter set	D_s	storage datacenter set
dc_i	i^{th} customer datacenter	dp_j	j^{th} storage datacenter
$\zeta^g_{dp_j}$	Get capacity of dp_j	$\zeta^p_{dp_j}$	Put capacity of dp_j
$p^s_{dp_j}$	dp_j's unit storage price	$p^t_{dp_j}$	dp_j's unit transfer price
$p^g_{dp_j}$	unit Get price of dp_j	$p^p_{dp_j}$	unit Put price of dp_j
$F^g(x)$	CDF of Get latency	$F^p(x)$	CDF of Put latency
α_{dp_j}	reservation price ratio	D	entire data set
d_l/s_{d_l}	data l and d_l's size	L^g/L^p	Get/Put deadline
ϵ^g/ϵ^p	allowed % of Gets /Puts beyond deadline	$v^{d_l,t_k}_{dc_i}$ $/u^{d_l,t_k}_{dc_i}$	Get/Put rates towards d_l from dc_i in t_k
Q^g/Q^p	SLO satisfaction level	β	number of replicas
t_k	k^{th} billing period in T	T	reservation time
C_t	total cost for storing D and serving requests	$X^{d_l,t_k}_{dp_j}$	existence of d_l in dp_j during t_k

where C_s, C_c, C_g and C_p are the total Storage, Transfer, Get and Put cost during entire reservation time T, respectively. The storage cost is calculated by:

$$C_s = \sum_{t_k \in T} \sum_{d_l \in D} \sum_{dp_j \in D_s} X^{d_l,t_k}_{dp_j} * p^s_{dp_j} * s_{d_l}, \quad (2)$$

where s_{d_l} denotes the size of data d_l, $p^s_{dp_j}$ denotes the unit storage price of datacenter dp_j, and $X^{d_l,t_k}_{dp_j}$ denotes a binary variable: it equals to 1 if d_l is stored in dp_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}, but is stored in the datacenter in the current period t_k. Thus, the data transfer cost is:

$$C_c = \sum_{t_k \in T} \sum_{d_l \in D} \sum_{dp_j \in D_s} X^{d_l,t_k}_{dp_j}(1 - X^{d_l,t_{k-1}}_{dp_j}) * p^t(dp_j) * s_{d_l}, \quad (3)$$

where $p^t(dp_j)$ is the cheapest unit transfer price of replicating d_l to dp_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 $R^g_{dp_j}$ and $R^p_{dp_j}$) 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 [15]. Then, we use α to denote the reservation price ratio, which means that the unit price for reserved Gets/Puts is $\alpha * p^g_{dp_j}$ and $\alpha * p^p_{dp_j}$, respectively. Thus, the Get/Put cost is calculated by:

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

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

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

SLO guarantee. Recall that *DAR*'s SLO specifies both deadline constraint and data availability constraint. 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 dc_i to dp_j within the deadlines L^g and L^p, denoted by $F^g_{dc_i,dp_j}(L^g)$ and $F^p_{dc_i,dp_j}(L^p)$.

To calculate $F^g_{dc_i,dp_j}(L^g)$ and $F^p_{dc_i,dp_j}(L^p)$, *DAR* records the Get/Put latency from dc_i to dp_j, and periodically calculates their cumulative distribution functions (CDFs) represented by $F^g_{dc_i,dp_j}(x)$ and $F^p_{dc_i,dp_j}(x)$. To calculate the average Get and Put rates from dc_i to dp_j per unit time in t_k (i.e., $r^{t_k}_{dc_i,dp_j}$ and $w^{t_k}_{dc_i,dp_j}$), *DAR* needs to predict the average Get and Put rates on each data (denoted by d_l) from dc_i per unit time in t_k (denoted by $v^{d_l,t_k}_{dc_i}$ and $u^{d_l,t_k}_{dc_i}$), respectively. For easy Get/Put rate prediction, as in [16, 11], *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 [17]. We use $H^{d_l,t_k}_{dc_i,dp_j} \in [0,1]$ to denote the ratio of requests for d_l from dc_i resolved by dp_j during t_k. Then,

$$r^{t_k}_{dc_i,dp_j} = \sum_{d_l \in D} v^{d_l,t_k}_{dc_i} * H^{d_l,t_k}_{dc_i,dp_j}$$
$$w^{t_k}_{dc_i,dp_j} = \sum_{d_l \in D} u^{d_l,t_k}_{dc_i} * X^{d_l,t_k}_{dp_j}$$

As a result, 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 $q^{t_k}_p$):

$$q^{t_k}_g = \frac{\sum_{dc_i \in D_c} \sum_{dp_j \in D_s} r^{t_k}_{dc_i,dp_j} * F^g_{dc_i,dp_j}(L^g)}{\sum_{dc_i \in D_c} \sum_{dp_j \in D_s} r^{t_k}_{dc_i,dp_j}},$$
$$q^{t_k}_p = \frac{\sum_{dc_i \in D_c} \sum_{dp_j \in D_s} w^{t_k}_{dc_i,dp_j} * F^p_{dc_i,dp_j}(L^p)}{\sum_{dc_i \in D_c} \sum_{dp_j \in D_s} w^{t_k}_{dc_i,dp_j}}. \quad (6)$$

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.

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

We see that if

$$Q^g \cdot Q^p = 1 \quad (8)$$

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 stores the requested data and satisfy the Get deadline SLO for each Get request of dc_i during t_k. The set of all datacenters satisfying the Get deadline SLO for requests from dc_i (denoted by $S^g_{dc_i}$) is represented by:

$$S^g_{dc_i} = \{dp_j | F^g_{dc_i,dp_j}(L^g) \geq (1-\epsilon^g)\}. \quad (9)$$

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

$$\forall dc_i \forall t_k \forall d_l \in G^{t_k}_{dc_i} \sum_{dp_j \in S^g_{dc_i}} X^{d_l,t_k}_{dp_j} \geq \beta \quad (10)$$

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

Datacenter capacity constraint. Beside the SLO constraints, each datacenter has limited capacity to supply Get and Put service, respectively [18]. Therefore, the cumulative Get rate and Put data rate of all data in a datacenter dp_j should not exceed its Get capacity and Put capacity (denoted by $\zeta^g_{dp_j}$ and $\zeta^p_{dp_j}$), 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 $\phi^g_{dp_j}$ and $\phi^p_{dp_j}$:

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

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

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

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

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

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

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

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

$$\forall dc_i \forall dp_j \forall t_k \forall d_l \ H^{d_l,t_k}_{dc_l,dp_j} \leq X^{d_l,t_k}_{dp_j} \leq 1 \tag{13}$$

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

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 I indicates the input and output parameters in this integer program. 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 ($F^g_{dc_i,dp_j}(x)$ and $F^p_{dc_i,dp_j}(x)$), the size of new data items d_l (s_{d_l}), and the data Get/Put rate from each dc_i ($v^{d_l,t_k}_{dc_i}$ and $u^{d_l,t_k}_{dc_i}$). The output is the data storage allocation ($X^{d_l,t_k}_{dp_j}$), request servicing ratio allocation ($H^{d_l,t_k}_{dc_i,dp_j}$) and the total cost C_t. The optimal Get/Put reservation in each storage datacenter ($R^g_{dp_j}$/$R^p_{dp_j}$) 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 represented by $T\backslash\{t_k\}$. *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.

This integer programming problem is NP-hard. A simple reduction from the generalized assignment problem [19] 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.

III. Data Allocation and Resource Reservation

DAR has two steps. First, its dominant-cost based data allocation algorithm (Section III-A) 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 III-B) 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.

A. 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), i.e., can supply a Get/Put SLO guaranteed service for a specific customer datacenter dc_i. Then, *DAR* selects datacenters from the candidates to store each data item requested by dc_i to satisfy other constraints and achieve cost minimization 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 dp_j serves Get/Put from dc_i, dp_j has $F^g_{dc_i,dp_j}(L^g) \geq 1-\epsilon^g$ and $F^p_{dc_i,dp_j}(L^p) \geq 1-\epsilon^p$. That is:

$$\forall t_k \in T \ \forall dc_i \in D_c, r^{t_k}_{dc_i,dp_j} > 0 \rightarrow F^g_{dc_i,dp_j}(L^g) \geq 1-\epsilon^g \tag{15}$$

$$\forall t_k \in T \ \forall dc_i \in D_c, w^{t_k}_{dc_i,dp_j} > 0 \rightarrow F^p_{dc_i,dp_j}(L^p) \geq 1-\epsilon^p \tag{16}$$

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

Algorithm 1: Dominant-cost based data allocation.

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

Min-cost storage datacenter selection. After the datacenter candidates $S^g_{dc_i} \cap S^p_{d_l}$ 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 $dp_j \in S^g_{dc_i} \cap S^p_{d_l}$ (denoted as $C^{d_l}_{dc_i,dp_j}$) is calculated based on Equation (1) and the dp_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 dc_i and stores d_l in the datacenter that minimizes the dominant cost.

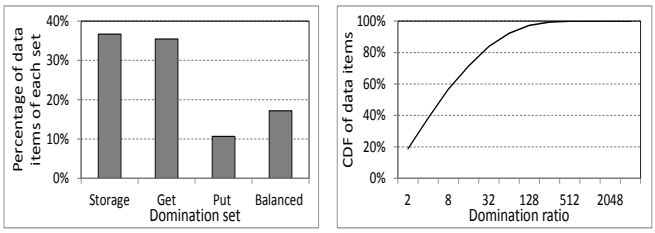
If one cost based on its minimum unit price among datacenters is larger than the sum of the other costs based on their maximum unit prices among datacenters, we consider this cost as the dominant cost. We do not consider the transfer cost for importing data when determining the dominant cost of a data 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 dc_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 $S^g_{dc_i} \cap S^p_{d_l}$ 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 dc_i is also time-consuming. Note that $S^g_{dc_i}$ is common for all data items requested by dc_i. Then, to reduce time complexity, we can calculate $S^g_{dc_i}$ only one time. From $S^g_{dc_i}$, we select the datacenter that belongs to $S^p_{d_l}$ to allocate each d_l requested by a given dc_i.

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 dc_i, we sort $S^g_{dc_i}$ by increasing order of unit Put cost, unit Get cost and unit Storage cost, respectively, which results in three sorted lists. We call them 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 $S^g_{dc_i}$.

For each data d_l requested by a given dc_i, we calculate its maximum/minimum Storage cost, Get cost and Put cost, respectively:
$Max^{d_l}_s = \sum_{t_k \in T} Max_s * s_{d_l} * t_k$,
$Max^{d_l}_g = \sum_{t_k \in T} Max_g * v^{d_l,t_k}_{dc_i} * t_k$,
$Max^{d_l}_p = \sum_{t_k \in T} Max_p * u^{d_l,t_k}_{dc_i} * t_k$,
$Min^{d_l}_s$, $Min^{d_l}_g$ 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



(a) Data distribution over dominant sets (b) CDF of data items over dominant ratio

Figure 1: Efficiency and the validity of the dominant-cost based data allocation algorithm .

allocation. Therefore, we choose a number of subsequent datacenters, calculate $C^{d_l}_{dc_i,dp_k}$ for d_l, where $k \in [j{+}1, j{+}c]$, and choose dp_j to create a replica and assign requests to (Constraint (13)) (Lines 15-17) if $C^{d_l}_{dc_i,dp_j}$ is smaller than all $C^{d_l}_{dc_i,dp_k}$. 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 $S^g_{dc_i} \cap S^p_{d_l}$ that generates the lowest total cost for d_l. In the datacenter selection process, the constraints in Section II-B 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 [20], 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 [21], 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 [1–3]. 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 1(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 1(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 dp_j in the sorted datacenter list of the dominant resource of a data item.

B. 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 $F_{dp_j}(x)$, 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. Thus, we can calculate the benefit by

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

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

$$O_{dp_j}(x) = \sum_{t_k \in T} Max\{0, x - \sum_{d_l \in D} \sum_{dc_i \in D_c} r^{t_k}_{dc_i,dp_j} * t_k\}. \quad (18)$$

Recall that $R^g_{dp_j}$ is the optimal number of reserved Gets for each billing period during T in a schedule. That is, when $x = R^g_{dp_j}$, $F_{dp_j}(x)$ reaches the maximum value, represented by $B_{dp_j} = F_{dp_j}(R^g_{dp_j}) = Max\{F_{dp_j}(x)\}_{x \in N^+}$.

In the following, we first prove Corollary III.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 III.2 that helps find reservation x to achieve the maximum reservation benefit. Finally, we present Theorem III.1 that shows how to find this reservation x.

Corollary III.1. *Given a datacenter dp_j that already stores a set of data items, allocating a new data item d_l and its requests to this datacenter, its maximum reservation benefit B_{dp_j} is non-decreasing.*

Proof: After allocating d_l to dp_j, we use $F'_{dp_j}(x)$ to denote the new reservation benefit function since $r^{t_k}_{dc_i,dp_j}$ in Equation (18) is changed. Then, we can get $F'_{dp_j}(R^g_{dp_j}) \geq F_{dp_j}(R^g_{dp_j})$ since $r^{t_k}_{dc_i,dp_j}$ is not decreasing. Since the new

reserved benefit $B'_{dp_j} = Max\{F'_{dp_j}(x)\}_{x \in N^+}$, thus $B'_{dp_j} \geq F'_{dp_j}(R^g_{dp_j}) \geq F_{dp_j}(R^g_{dp_j}) = B_{dp_j}$ after d_l is allocated. ■

We define the number of Gets in t_k as $m = Max\{\sum_{d_l \in D} \sum_{dc_i \in D_c} r^{t_k}_{dc_i,dp_j} * t_k\}_{t_k \in T}$. Then, according to Equation (17), we can get the optimal reservation Gets $R^g_{dp_j} \in [0, m]$. Thus, by looping all integers within $[0, m]$, we can get the optimal reservation that results in maximum F_{dp_j}. This greedy method, however, is time consuming. In order to reduce the time complexity, we first prove Corollary III.2, based on which we introduce a binary search tree based optimal reservation method.

Corollary III.2. *For a datacenter dp_j, its benefit function $F_{dp_j}(x)$ is increasing when $x \in [0, R^g_{dp_j})$ and decreasing when $x \in (R^g_{dp_j}, m]$.*

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

Algorithm 2: Binary search tree based resource reservation.

```
Build a balanced binary search tree of A with A^{t_k}_{dp_j} ;
N_1 = ⌊n*(1−α)⌋ + 1; N_2 = ⌈n*(1−α)⌉ + 1;
x_1 = the N_1^{th} smallest value of A;
x_2 = the N_2^{th} smallest value of A;
if F_{dp_j}(x_1) ≥ F_{dp_j}(x_2) then
    R^g_{dp_j} = x_1;
else
    R^g_{dp_j} = x_2;
```

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

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

Proof: The proof of Corollary III.2 indicates that when $O_I(x) = \lceil n*(1-\alpha) \rceil$ or $\lfloor n*(1-\alpha) \rfloor$, $F_{dp_j}(x)$ can reach B_{dp_j}. As indicated above, $O_I(x)$ represents the number of billing periods during T with $A^{t_k}_{dp_j} = \sum_{dc_i \in D_c} r^{t_k}_{dc_i,dp_j} * t_k < x$. Therefore, when x is the N^{th} smallest value in $\mathbf{A}$, where N equals $\lceil n*(1-\alpha) \rceil + 1$ or $\lfloor n*(1-\alpha) \rfloor + 1$, $F_{dp_j}(x)$ reaches B_{dp_j}. ■

We then use the binary search tree algorithm to find the optimal reservation number of Gets. Its pseudocode is shown in Algorithm 2. The time complexity of this algorithm is $O(n*\log n)$. It builds a binary search tree using $O(n*\log n)$, and then finds the N^{th} and $(N+1)^{th}$ smallest values in the tree using $O(\log n)$, and all other operations takes $O(1)$.

IV. 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 2, *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.

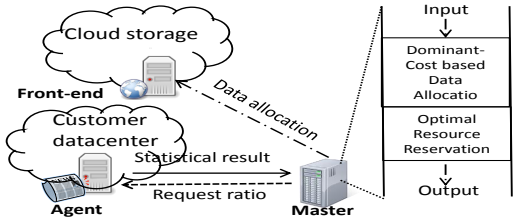
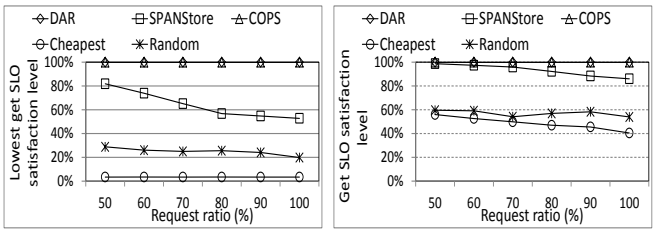


Figure 2: Overview of DAR's infrastructure

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 the SLOs. 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.

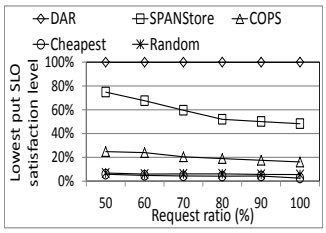
During each t_k, for the schedule recalculation, the master needs the latency's CDF of Get/Put ($F^g_{dc_i,dp_j}(x)$ and $F^p_{dc_i,dp_j}(x)$), the size of each d_l (s_{d_l}), and the data's Get/Put rate from dc_i ($v^{d_l,t_k}_{dc_i}$ and $u^{d_l,t_k}_{dc_i}$). Each agent in each customer datacenter 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. 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.



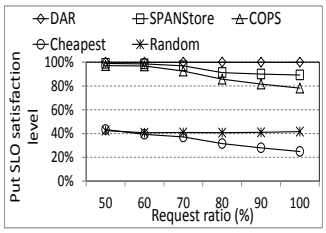
(a) In simulation

(b) In real world

Figure 3: Get SLO guarantee performance.



(a) In simulation



(b) In real world

Figure 4: Put SLO guarantee performance.

V. Performance Evaluation

We conducted trace-driven experiments on Palmetto Cluster [22] with 771 8-core nodes and on real clouds. 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 [11]. 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 [15].

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 [11], in the SLOs for all customers, the Get deadline is restricted to $100ms$ [11], the percentage of latency guaranteed Gets and Puts is 90%, and the Put deadline for a customer's datacenters in the same continent is 250ms and is 400ms for an over-continent customer. The minimum number of replicas of each data item was set to $\beta = 3$ [15]. The size of the aggregated data of a customer was randomly chosen from $[0.1TB, 1TB, 10TB]$ as in [11]. 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 [23].

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 [11]. The Put rate follows the publicly available wall post trace from Facebook New Orleans networks [20], 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 [21], from which we set the Get rate of each data item accordingly. Facebook is able to handle 1 billion/10 million Gets/Puts per second [21], 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 [15]. 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 [24]. Unless otherwise indicated, the settings are the same as before. Due to the small scale, the number of data items was set to 1000, the size of each 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:

- *SPANStore* [11], which is a storage over multiple CSPs' datacenters to minimize cost supporting SLOs without considering capacity limitations and reservations.
- *COPS* [25], which allocates requested data in a datacenter with the shortest latency to the customer datacenter.
- *Cheapest*, in which the customer selects the datacenters with the cheapest cost to store each data item without considering SLOs and reservations.
- *Random*, in which the customer randomly selects datacenters to allocate each data item.

A. Comparison Performance Evaluation

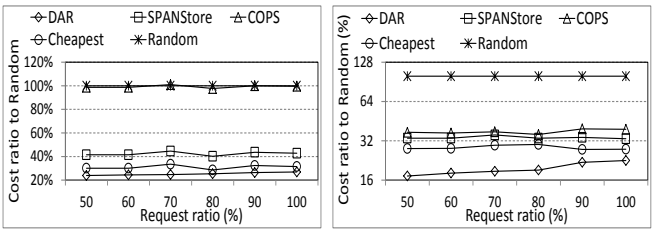
To evaluate the SLO guarantee performance, we measured the lowest SLO satisfaction levels of all customers. The Get/Put SLO satisfaction level of a customer, Q^g/Q^p, is calculated according to Equation (7) with $q^g_{t_k}/q^p_{t_k}$ as the actual percentage of Gets/Puts within deadline during t_k. We varied

each data item's Get/Put rate from 50% to 100% (called request ratio) of its original rate with a step size of 10%.

Figures 3(a) and 3(b) show the (lowest) Get SLO satisfaction level of each system versus the request ratio on the testbed 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 4(a) and 4(b) show the lowest Put SLO satisfaction level of each system versus the request ratio on the testbed 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 3(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* 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 3(a). The figures indicate that *DAR* can supply a Put SLO guaranteed service while others cannot.

Figure 5(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* on the testbed. The figure shows that the cost follows *COPS*≈*Random*>



(a) In simulation (b) In real world

Figure 5: Cost minimization performance.

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. While all others do not consider the capacities of datacenter, and hence violate the Get/Put SLO by making some datacenters overloaded. Figure 5(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 5(a) due to the same reason, except that *COPS*<*Random*. That 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.

VI. Related Work

Deploying on multiple clouds. RACS [7] and DepSky [26] 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 [27], an application execution platform across multiple CSPs was proposed. Wang *et al.* [28] studied content propagation in social media traces and found that the propagation is quite localized and predictable. Based on this pattern, they proposed a social application deployment using local processing for all contents and global distribution only for popular contents among cloud datacenters. COPS [25] and Volley [16] automatically allocate user data among datacenters in order to minimize user latency. Blizzard [29] 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 [8–10], cluster storage automate configuration methods are proposed

to use the minimum resources needed to support the desired workload. None of the above papers study the cost optimization problem for geo-distributed cloud storage over multiple providers under SLO constraints. SPANStore [11] 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.

Improving network for SLO guarantee. Several works [30–33] 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.

VII. 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. *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. In our future work, we will explore methods to handle the situations, in which the data request rate varies largely and datacenters may become overloaded.

Acknowledgements

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

References

[1] Amazon S3. http://aws.amazon.com/s3/.
[2] Microsoft Azure. http://www.windowsazure.com/.
[3] Goolge. https://cloud.google.com/products/cloud-storage/.
[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.
[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] 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.
[9] 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.
[10] 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.
[11] 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.
[12] J. Dean. Software Engineering Advice from Building Large-Scale Distributed Systems. http://research.google.com/people/jeff/stanford-295-talk.pdf.
[13] 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.
[14] Service Level Agreements. http://azure.microsoft.com/en-us/support/legal/sla/.
[15] Amazon DynnamoDB. http://aws.amazon.com/dynamodb/.
[16] 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.
[17] G. Liu, H. Shen, and H. Chandler. Selective Data Replication for Online Social Networks with Distributed Datacenters. In *Proc. of ICNP*, 2013.
[18] 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.
[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
[20] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the Evolution of User Interaction in Facebook. In *Proc. of WOSN*, 2009.
[21] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. of NSDI*, 2013.
[22] Palmetto Cluster. http://http://citi.clemson.edu/palmetto/.
[23] 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.
[24] Amazon EC2. http://aws.amazon.com/ec2/.
[25] 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.
[26] A. N. Bessani, M. Correia, B. Quaresma, F. Andr, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *TOS*, 2013.
[27] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *Proc. of NSDI*, 2012.
[28] Z. Wang, B. Li, L. Sun, and S. Yang. Cloud-based Social Application Deployment using Local Processing and Global Distribution. In *Proc. of CoNEXT*, 2012.
[29] 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.
[30] C. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. of SIGCOMM*, 2012.
[31] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). In *Proc. of SIGCOMM*, 2012.
[32] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *Proc. of CoNEXT*, 2010.
[33] 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.