A Reinforcement Learning Based System for Minimizing Cloud Storage Service Cost

Haoyu Wang*, Haiying Shen*, Qi Liu*, Kevin Zheng*, Jie Xu♯
*University of Virginia, Charlottesville, USA,
{hw8c, hs6ms, ql8va, ksz3kd}@virginia.edu
♯George mason University, Fairfax, USA,
{jxu13}@gmu.edu

ABSTRACT

Currently, many web applications are deployed on cloud storage service provided by cloud service providers (CSPs). A CSP offers different types of storage including hot, cold and archive storage and sets unit prices for these different types, which vary substantially. By properly assigning the data files of a web application to different types of storage based on their usage profiles and the CSP's pricing policy, a cloud customer potentially can achieve substantial cost savings and minimize the payment to the CSP. However, no previous research handles this problem. Towards this goal, we present a Markov Decision Process formulation for the cost minimization problem, and then develop a reinforcement learning based approach to effectively solve the problem, which changes the type of storage of each data file periodically to minimize money cost in long term. We then propose a method to aggregate concurrently requested data files to further reduce the cloud storage service payment for a web application. Our experiments with Wikipedia traces show the effectiveness of the proposed methods for minimizing cloud customer cost in comparison with other methods.

ACM Reference Format:
Haoyu Wang*, Haiying Shen*, Qi Liu*, Kevin Zheng*, Jie Xu♯. 2020. A Reinforcement Learning Based System for Minimizing Cloud Storage Service Cost. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3404397.3404466

1 INTRODUCTION

Cloud storage, such as Microsoft Azure [5], Amazon S3 [1], and Google Cloud Storage [4], has become a popular commercial service provided by cloud service providers (CSPs). More and more individual customers and enterprises are moving their data workloads to cloud storage in order to save the capital expenditures required for building and maintaining the hardware infrastructures and avoid the complexity of managing the data centers. Cloud storage service can be used by many web applications, such as online social networks and web portals, to serve clients distributed worldwide.

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404466

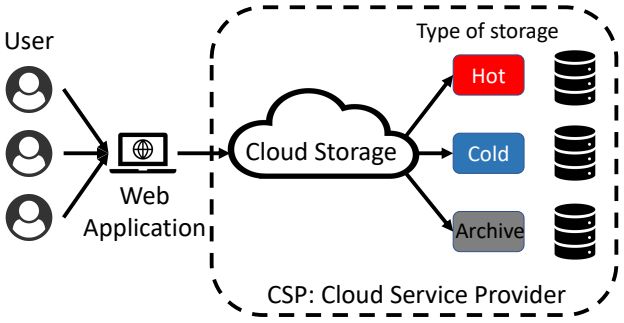

Figure 1: The structure of web application using cloud storage service.

The structure of the web application using cloud storage service is shown in Figure 1. A CSP offers different types of storage including hot, cold and archive storage and sets unit prices for these different types, which vary substantially.

The payment that a cloud customer makes to the CSP includes the costs for storage, data read/write (per operation and per GB), change of storage type. The cost for the change of storage type means that when a customer changes the type of a data file, it generates a certain cost. Operations are priced differently for the different types of storage. For example, Microsoft Azure charges user $0.0044 in US West region per 10,000 reading operations and $0.01 per GB for hot files, and charges $0.01 per 10,000 data reading operations and $0.004 per GB for cold files.

In order to maximize the profit of the web application, the owner of the web application (i.e., cloud customer) needs to minimize the total cloud storage service money cost (cost in short) according to the pricing policy set by the CSP. Because the prices can vary significantly for different types of storage (hot, cold, archive data), instead of using a single type of storage, cloud storage service customers may be able to minimize the cost of the service by carefully assigning data files into different types of storage based on their usage profiles. However, it is a non-trivial task because of the inherent stochasticity and uncertainty in different data files' request frequencies over time [45]. Suppose a cloud customer assigns a data file to the cold storage, and then unexpectedly the file's request frequency increases significantly, which then leads to a sharp increase in cost. The cloud customer can change the type of storage for this data file from cold to hot so that its read/write operation cost can be reduced. However, the total payment cost is related to other features such as the change of storage type. Frequently changing the type of a data file may generate more cost than the

cost saving. Thus, in order to reduce the total payment, the cloud customer needs to appropriately determine the type of data storage and also make dynamical adjustment in response to the fluctuations in request frequency.

As a result, there is a great need for a system that can accurately predict data file request frequency and then dynamically determine the type of storage for each data file in order to minimize the cost of a web application. Thus, the *problem* handled in this paper is: *given several features (e.g., request frequency, storage type and data size) of the data files in a web application deployed in a cloud storage service, find the data storage type assignment plan (i.e., determine the type of storage for each data file) periodically that minimizes the total payment to the CSP.* This is the first work that handles this problem.

Many previously proposed methods [9–11, 17, 24, 25, 28, 42] focus on predicting the minimum amount of resources needed to support the application workload to reduce cloud storage cost. For example, Madhyastha *et al.* [25] studied the relationship between capital expenditures on different types of hardware and different storage configurations, with an aim to reduce the total monetary cost. Adya *et al.* [9] proposed a file system that offers high availability and scalability at low cost. However, these methods depend on having perfect knowledge about the workload and also do not focus on the data type assignment problem.

In order to overcome the aforementioned challenges in handling the problem, we propose *MiniCost* (Minimizing the Cost for storage), a reinforcement learning based automated data storage type assignment system for an owner of a web application to adjust the storage types of data files stored in the CSP. It generates the data storage type assignment plan according to the variable request frequencies of data files and then adjusts the type of data storage for each data file according to the changes in several features of the data files (e.g., request frequency, data size). We first formulate this minimizing cloud storage service cost problem into a streamlined Markov Decision Process (MDP). Since the reinforcement learning method can generate the optimized actions automatically, we introduce how to use reinforcement learning-based method to solve this problem. Trace-driven experimental results show this method outperforms existing online methods by a significant margin, and even achieves performance comparable with that of the offline method using computationally expensive brutal force optimization. We summarize our contributions below:

- We first analyze the Wikipedia trace data [6] and then model the cost minimization problem using a streamlined MDP formulation.
- We introduce a reinforcement learning based data storage type assignment algorithm, which changes the storage types of each data file periodically according to a cost-efficiency objective function to minimize the total payment to the CSP in the long term.
- We propose an enhancement method to further reduce the total payment. Specifically, by combining several concurrently requested data files into one data file, the number of data requests can be reduced, which can then help reduce the total cost significantly.
- We conduct extensive trace-driven experiments on a supercomputing cluster with a trace from page view statistics of Wikipedia projects [6]. Results demonstrate the efficiency and effectiveness of our system in minimizing a customer's cloud storage service cost and reducing system overhead in comparison with existing methods.

The rest of paper is organized as follows. Section 2 presents the related work. In Section 3, we show our analytical results for the Wikipedia trace. Section 4 formulates the cost minimization problem into MDP. Section 5 introduces the reinforcement learning method and the enhancement to solve this cost minimization problem. Section 6 presents the trace-driven experimental results of *MiniCost* compared with other methods. Section 7 concludes our work with remarks on our future work.

2 RELATED WORK

Cloud storage payment minimization: Alvarez *et al.* [10] proposed a dynamic programming method to determine the minimum amount of resource needed to satisfy the requirement of web applications. Anderson *et al.* [11] explored the designs of storage system which focuses on the satisfaction of storage I/O requirements. Madhyastha *et al.* [25] proposed *scc* which automates the cluster configuration decisions based on hardware properties and formal decisions. Adya *et al.* [9] proposed a file system which has high availability and minimizes the cost through lazy propagation of file updates. However, none of these earlier works studies how to determine the type of storage for data files for cloud storage service cost minimization. SPANStore (Storage Provider Aggregating Networked Store) [45] is a key-value storage system to minimize cost and guarantee SLOs (service level objectives). Since machine rental and machine purchase have different advantages in storage (e.g., easy to replace new machine and smaller short-term cost for machine rental and smaller long-term cost for machine purchase), Li *et al.*[22] evaluated the tradeoff between machine rental and machine purchase. They also provided a hybrid cloud assisted storage plan with both machine rental and purchase for cost minimization. However, they do not consider the variable request frequencies for data files which can highly affect the application demand for different resource requirements.

Cloud resource pricing: There are several previous works studying the cloud resource pricing problem. In [31, 37, 41], how to maximize the benefits of cloud service customers under dynamic pricing models (including adaptive leasing and auctions) is studied. Roh *et al.* [34] formulated the pricing and resource competition as a concave game to describe the quantity competition among application service providers reducing payment. Wu *et al.* [40] aimed to achieve lower data transmission tail latency through cloud storage position optimization among geo-distributed datacenters while reducing the total payment made to the CSP by customers. Li *et al.* [23] provided a cost minimization method that takes into account the limited bandwidth and storage capacities among multiple CSPs.

Combining cloud providers: Some works consider combining the use of multiple cloud providers to improve availability. Abu *et al.* [8] designed a function to evaluate the availability of different cloud providers and then proposed a method to achieve better availability by combining multiple cloud providers while reducing the monetary cost. Kotla *et al.* [20] proposed a similar method which

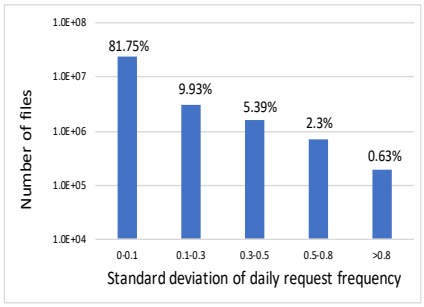

Figure 2: Distribution of cost savings achieved by optimal data assignment.

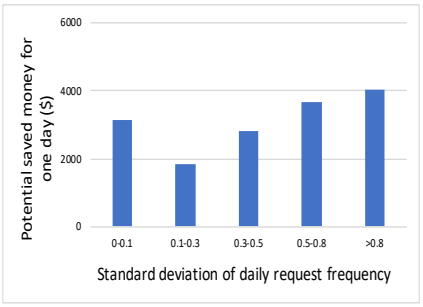

Figure 3: Distribution of cost savings achieved by optimal data assignment.

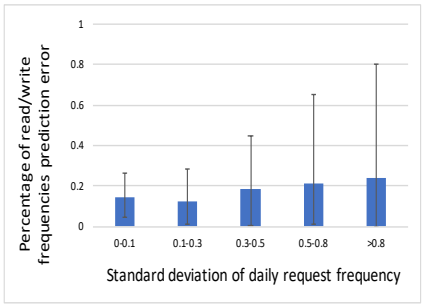

Figure 4: Distribution of request frequency prediction errors.

explores all the possible groups of multiple CSPs that can satisfy long-term data durability. Wieder *et al.* [44] proposed a system to determine the optimal plan for deploying MapReduce jobs via multiple CSPs with different user-specified goals like minimizing cost or reducing the completion time. SELECTA [18] recommends near-optimal configurations of cloud compute and storage resource for data analytic workloads. It uses machine learning to predict how a job will perform under different resource configurations and then recommend the best one. HyCloud [13] provides a cost efficient hybrid file system which converts large-size file downloads into small-size file synchronizations to reduce the download money cost based on both Amazon EFS (Elastic File System) and S3.

Unlike the above methods, the goal of our method is to minimize the total payment a cloud storage service customer made to a CSP by leveraging the different types of storage provided by the CSP.

3 TRACE ANALYSIS: REQUEST FREQUENCY VARIABILITIES

3.1 Data Analysis

We begin with the analysis of the Wikipedia trace [6] which is widely used in Cloud Storage System experiment [21, 40]. This trace includes hourly Wikipedia page views per article. We collected the data from Jul. 15th to Sep. 15th for about 4 million articles in English. We aim to use this trace to simulate people's request frequency change via web application. Thus, for the simulation of the data storage for web application, we use Poisson distribution to set the size of the data for all the contents (such as texts, videos, figures, and music files) in each webpage [33]. The average data size is 100MB and we assume the data size will not change in this two month period as in [43]. This trace doesn't consist the location information for each request since CSP will not charge the number of requests according to the resource locations. We first quantify the request frequency variabilities for data files in the time series, and analyze the impact of request frequency variability on the cloud storage service cost in a given period of time (e.g., one week in this paper, since the cycle time of the request frequencies for each data file is around one week [32]). We then identify the room for cost reduction according to the current CSP pricing policy.

In Figure 2, we plot the histogram of daily request frequency standard deviations for all data files in the trace. The standard deviation $\mathbb{S}$ for each data file is calculated by:

$$\mathbb{S} = \sqrt{\frac{1}{T-1}\sum_{i}^{T}(r_i - \bar{r})^2} \; (i = 0, 1, 2...) \tag{1}$$

where T is the number of days, r_i is ith day's data request frequency for a data file and $\bar{r}$ is its average request frequencies in T days. We can see that the data files with standard deviations between 0 and 0.1 (i.e., low request frequency variabilities) constitute around 85% of the total number of data files. However, there are still a large number of data files (in the order of 10^5) with very high request frequency variabilities (> 0.8) although they only constitute around 0.75% of the total number of data files. The rest of data files which occupy around 15% have certain request frequency variabilities ranging from 0.1 to 0.8. Therefore, the trace contains both stationary files (low deviation value) and non-stationary files (high deviation value). For a data file, when it has different amounts of request frequencies over time, it should be assigned to different storage types in order to reduce money cost. If we can optimally change the types of storage for the data files over time, we can potentially achieve substantial cost savings.

Since the files with high request frequency variabilities constitute a small percentage of the total number of data files, we want to see if changing the types of storage only for these files can generate a high cost saving, and also if changing the types of storage for files with request frequency variabilities can generate a high cost saving or not. We then explore the potential of cost savings for different standard deviation using Microsoft Azure Block Storage Pricing Policy [3]. Figure 3 shows the potential saved money in dollars versus the standard deviation of daily request frequency. In order to get the potential saved money, we first compute the total payment charged by the CSP if the customer assigns all data files as either hot or cold, depending on which one yields a lower cost. We then apply an offline brutal-force method (explore all the possible types of storage and then select the one with the minimum cost) on the trace to determine the optimal storage allocation. The potential saved money is the difference between the two values from the above two methods.

It is interesting to observe that although there are two orders of magnitudes (10^7 versus 10^5) more data files with low request frequency standard deviations (0 to 0.1) than with high standard deviations (> 0.8), those files with high request frequency variabilities actually lead to more cost savings (around $4000 versus $3000),

which means the money saved per data file is much higher than the data files with low standard deviations. However, the data files with low request frequency standard deviations still generate a high cost saving due to a large number of files. The rest of data files which occupy around 15% can save around $2000-$4000. Therefore, proper data storage type adjustment for a data file with a high request frequency variability over time leads to more potential cost savings. Also, a cloud customer should have proper data storage type selection for all data files (regardless of the request frequency variability) in order to reduce the total cost; since changing of storage type also generates cost and the cost savings of a data file with a low request frequency variability is relatively low, the decision for such a file's data storage type adjustment must be carefully made.

We use the popular time-series prediction model *ARIMA* [2, 15] that uses the first two months data to predict the daily request frequency within the next 7 days. In another word, the prediction model generates 7 predicted daily request frequencies for each data file. Figure 4 shows the 1%, median and 99% of prediction errors for all data files. The prediction error is calculated by $(True\ value - Predicted\ value)/True\ value$. We observe that the prediction errors are considerably higher for data files with high request frequency variabilities (the bars on the right) which is hard to predict by *ARIMA*. However, as observed in Figure 3, these data files have the most potential for cost savings. Therefore, in order to minimize the total cost for cloud storage, it is a challenge to identify data files with high standard deviations according to the variable request frequencies to adjust the data storage type.

3.2 Challenges

Our trace data analysis demonstrates the potential of cost minimization by leveraging the different prices of storage types and properly assign data files into different types of storage. However, the trace data analysis also reveals that the data storage type assignment system faces challenges. We explain the challenges below.

Data file request frequencies can vary significantly for different time periods. This complicates the task of determining an optimal data type assignment plan since different features of data files requires different data storage type assignment plans. To handle this problem, we can dynamically adjust the data storage type assignment plan periodically over time. However, the change of data storage type can increase the cost. For example, when deciding if the storage type of one data file should be changed from hot to cold, we must consider the possibility of the request frequency going up significantly in the future, which will then require the change of the storage type back to hot from cold. Because of the cost of data storage type change, the total payment cost may actually go up. Therefore, the data storage type assignment system needs long-term file request frequency prediction and then specifies the type of storage accordingly, instead of trying to chasing for the short-term cost savings like a typical greedy algorithm (simply select the storage type with the minimum money cost only for the next day) would do. In the following, we introduce our proposed *MiniCost* data storage type assignment system to handle the challenges.

4 MDP-BASED PROBLEM FORMULATION

4.1 System Model

To address the aforementioned challenges, we present a Markov Decision Process (MDP) problem formulation that minimizes a cloud customer's total payment cost to the CSP. All of the data files of the web application of a cloud customer are stored in the cloud storage. Those data files are distributed among one or multiple CSPs's datacenters denoted by the set D_s. Each datacenter has its own pricing policy for different data storage type.

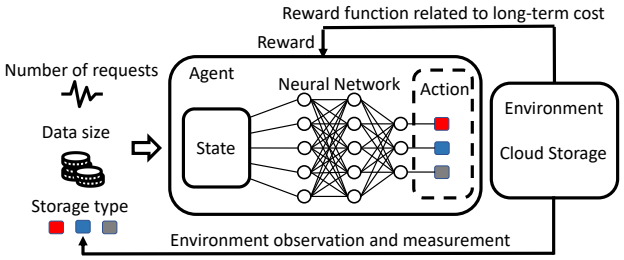

Figure 5: Applying reinforcement learning to minimize cloud storage cost.

One read/write request can be defined as the request originating from a user of the web application or the web application itself (e.g., data reallocation). According to the Azure pricing model [5], data storage service is charged in the pay-as-you-go manner based on the number of read/write operations, data file sizes, and the duration that data files occupy storage. The total payment made to the CSPs by the cloud customer would vary under different data storage type assignment plans. *MiniCost* helps a cloud customer to assign the data files to different types of storage over time to achieve the minimized total payment cost.

4.2 Problem Formulation

The problem handled by *MiniCost* is: given several features (e.g., request frequency, storage type, and data size) of the data files stored in cloud storage, find the data cloud storage type assignment plan periodically that minimizes the total payment to the CSPs. Since the information (including the request frequency, storage type, and data size) can be observed by *MiniCost* of the cloud customer from the cloud storage directly, it has all the features needed to make an optimal decision [16], we can formulate this cost minimization problem into MDP denoted by $M = (S, A, P, R)$. S denotes the state, A denotes the action, P denotes the probability between each two states and R denotes the reward. When an action is taken in a state, the state is transferred to another state with a certain probability and corresponding reward is received. A reinforcement learning (RL) agent, which is responsible for generating the data storage type assignment plan periodically, is deployed on a server belongs to the web application. It monitors the request frequencies, changes of data storage types and the change of data size. We define an action as generating a data storage type assignment plan. The agent selects a possible action $a_t \in A(s_t)$ at time step t where $A(s_t)$ refers to the set of all possible actions in the current state s_t; $R(s, a)$ denotes the reward obtained by doing action a at state s. When the agent

makes a decision for a data file that determines the data storage type of the data file in the next time step, the data file is assigned to the storage type with certainty. Thus, $P(s'|s,a)$, the transition probability from state $s \in S$ to state $s' \in S$ after taking an action $a \in A$ is always 1. According to the cost minimization problem, the objective function is to find a policy of actions that minimizes the total cost. Below, we introduce the elements in $M = (S, A, P, R)$ of the MDP.

4.2.1 State Space. The state space S consists of the information on read request frequencies, sizes, write (or update) frequencies, and storage types of data files. It is defined as follows:

$$S = \{s = (F_r, F_w, D, \Gamma)\} \tag{2}$$

where F_r, F_w, D denote read frequencies, write frequencies, and size of the data files. The cardinality of the type of storage set Γ is determined by the CSP's policy. For example, in this paper, the number of storage types on Microsoft Azure is $\Gamma = 3$, including *hot, cold and archive*. Noted that, Γ can be easily adjusted for multiple CSPs since multiple CSPs have more number of storage types.

4.2.2 Action Space. The action space A is defined as

$$A = \{a = (a_0 ... a_N) | a_i \in \{1, ..., \Gamma\}, i = 1, \ldots, N\} \tag{3}$$

where N is the number of data files under consideration, and a_i is the action to assign ith data file to one data storage type. For example, in this paper, the possible options for one action a_i is 3, corresponding to keep the file in the same storage type or assign it to either of the other two storage types. The action space is $3 * N$. According to Figure 2, over 80% of files in the trace have 0-0.1 standard deviation value, so these files will stay in one storage type. We focus on one single CSP though our model can be used for multiple CSPs.

4.2.3 Reward. The main goal of the agent is to make an optimal data type assignment decision at each decision period, e.g., one week in this paper (the typical cycle time for request frequency of data file [32]), to minimize the total cost. Thus, we define the reward function $R(s_t, a_t)$ given an action a_t at the state s_t as follows:

$$R(s_t, a_t) = \frac{\alpha}{C(s_t, a_t)} + \Delta \tag{4}$$

where $C(s_t, a_t)$ is the total money cost of taking action a_t at state s_t. We introduce two parameters α and Δ here which can be set manually. Our principle of the reward function is setting a higher reward for the action that decreases the total money cost.

The total money cost includes the cost of storage, the cost of changing data storage type, and the cost of read/write operations, which are denoted by C_s, C_c, C_r, C_w respectively. The total cost incurred by the web application is given by:

$$C(s_t, a_t) = C_s + C_c + C_r + C_w. \tag{5}$$

The storage cost in a datacenter is the product of the data size and the unit storage price within each datacenter. Then, for the ith data file d_i stored in j storage type p_j, the storage cost is calculated by:

$$C_s = \sum X_{P_j}^{d_l,t} * u_{p_j} * D_{d_l}, \tag{6}$$

where D_{d_i} denotes the size of data item d_i, u_{p_j} denotes the unit storage price of storage type p_j, and $X_{P_j}^{d_i,t}$ a binary variable: it is 1 if d_i is stored in p_j during time step t and 0 otherwise.

The read/write cost is the sum of the payment made to the CSP for the read/write operations and can be calculated by:

$$C_r = \sum F_r^t * (u_{rf} + u_{rs} * D_{d_i}), \tag{7}$$

$$C_w = \sum F_w^t * (u_{wf} + u_{ws} * D_{d_i}), \tag{8}$$

where F_r^t and F_w^t denote the read/write frequencies at time step t. u_{rf} and u_{wf} denote the unit price (e.g., 10000 in Azure) per operations. u_{rs} and u_{ws} denote the read/write data size unit price per GB.

The cost of changing data storage type is a one-time cost. If the data storage type of one data file is changed, the state of this data s_t is not the same with the state s_{t-1} in the last time step. Thus, we use a binary variable Θ_{d_i} to denote whether the data storage type of data d_i is changed and u_{tran} to denote the one time cost for changing data storage type. The cost of data transmission is the product of the unit price and the size when $\Theta_{d_i} = 1$.

$$C_c = \sum \Theta_{d_i,t} * u_{tran} * D_{d_i}. \tag{9}$$

5 MAIN METHODOLOGY

5.1 Deep Q-Network based Algorithm

To solve the minimizing cloud storage service cost problem formulated by the MDP, we propose a reinforcement learning (RL) based method because RL is a widely-used technique to solve the MDP problem [19, 35]. An RL agent observes previous environment states and rewards, and then decides an action in order to maximize the calculated reward. RL can react to different complex environments immediately and efficiently [12]. In this paper, *MiniCost* uses a widely-used RL-based training algorithm, Asynchronous Advantage Actor Critic (A3C) [29], which is a state-of-the-art RL method involved training two Deep Q-Networks (DQNs). DQN has been successfully used to solve large-scale RL tasks [26, 27, 30, 36].

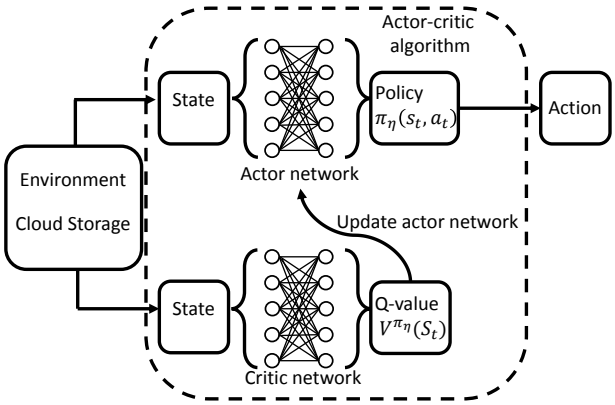

Figure 6: The A3C algorithm used in *MiniCost*.

As shown in Figure 6, A3C includes two types of DQN which are actor network and critic network. The actor network selects actions according to the probability distribution. The critic network generates the reward depending on the action generated by the actor network. The actor network finally updates the probability distribution according to the reward from the critic network. We define $\pi(s_t, a_t)$ as the probability that one action a_t taken for state s_t and $\pi : \pi(s_t, a_t)$ as the probability distribution. Finally, given s_t, the agent takes an action based on this probability distribution; that is, it takes action a_t with probability $\pi(s_t, a_t)$. The actor network uses policy gradient method [39] to gradient ascent to convergence but the critic network can tell the actor network if the direction of gradient ascent is correct or not. Thus, the performance of A3C is better than the typical RL methods as indicated in [14, 38].

In the following, we briefly introduce the mathematical derivation of the A3C algorithm used in *MiniCost*. For more details, please refer to [27, 39]. Before the DQN is trained, the agent uses the data type specified by the cloud customer for training. For training the DQN, the agent takes the real-time data or historical data as input and then outputs the actions with the reward calculated in Equation (4). The process of the DQN recording the relationship among the reward value, the different input states and the different output actions is the learning process of the agent. The following mathematical derivation shows us how to update the parameters in DQN in the training process.

Under variable request frequencies for different data files, for the input state s_t, we first define the total expected reward from state s_t is $V^{\pi}(s_t)$ and the total reward from taking a_t in s_t as $Q^{\pi}(s_t, a_t)$. Thus, the advantage function is:

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{10}$$

which can represent how much better of the action selection a_t. Using policy gradient method [38], the gradient of cumulative reward with gradient parameter η is:

$$\nabla_{\eta} J(\eta) \approx \nabla_{\eta} log \pi_{\eta}(s_t, a_t) A^{\pi}(s_t, a_t) \tag{11}$$

where $J(\eta)$ is the cumulative reward function, $\pi_{\eta}(s_t, a_t)$ is the probability that action a_t is taken in state s_t with actor network parameter η, and η can be updated by:

$$\eta \leftarrow \eta + \alpha \nabla_{\eta} J(\eta) \tag{12}$$

where α is the learning rate. The direction in Equation (11) determines how to change the policy parameter η and then improve the policy $\pi_{\eta}(s_t, a_t)$. Note that, there are no shared features between actor network and critic network. The critic network only evaluates the worthwhile of the action selected by the actor network and the actor network will select actions according to the updated policy distribution. The advantage of this RL method is unbiased and stable results.

After the DQN is trained, we deploy the trained DQN in the agent server. The agent keeps track of all the necessary information of all data files. Everyday, the trained agent runs one time for all data files, generates the action for each data files in the next day. The data file will be maintained in the same storage type or changed to other types according to the action made by the agent. The DQN is kept being trained all the time. The pseducode of the DQN-based decision making process is shown in Algorithm 1.

Algorithm 1: Pseducode of the DQN-based decision making algorithm.

```
1 Initialize memory for Neural Network, several default parameters
2 for t=1,2,... do
3     Observe current state s_t
4     Select action a_t: with π_η(s_t, a_t)
5     Execute the action for one data file as a_t and observe reward r_t = R(s_t, r_t) and the next state s_{t+1}
6     Update A^π_η(s_t, a_t)
7     Randomly select a set of actions (s_t, a_t, r_t, s_{t+1}) from the memory of neural network
8     Train the neural network based on gradient calculated and updated in Equation (12)
9 end for
```

The algorithm first initializes the memory of neural network, several default parameters using in RL (line 1). The RL agent observes the state s_t via monitoring the several features of all the data files (line 3). The agent selects the action a_t according to the policy distribution $\pi_{\eta}(s_t, a_t)$ (line 4). Then, the agent executes the selected actions and then calculate the reward in Equation (4) for training (line 5). Finally, the neural network updates the advantage function, determines the direction of gradient and then finally updates the policy distribution for the next time step (lines 6-8). Since the decision making process replies on the policy distribution, the data type assignment decision can be made in *O(1)* time. Therefore, for all the data files, the time complexity is *O(n)* where n is the number of data files.

5.2 Concurrent Requested Data Files Aggregation

For a web application, some data files are usually requested concurrently. For example, different data files, linked to one webpage, are usually requested concurrently. Based on this key observation, we propose to aggregate two or more data files that are often requested concurrently into another data file to reduce the total request frequencies and thus may help decrease the total cost determined by multiple features of data files and the storage types.

Not all concurrently requested files should be aggregated because the new aggregated data file replica takes extra storage and thus also increases cost. If the increase cannot be offset by the decrease in cost due to reduced request frequencies, aggregation may backfire. For instance, consider data file d_1 and d_2 with the number of concurrent request frequencies denoted by r_{d_c}. The total number of requests are r_{d_1} and r_{d_2} respectively. In this case, aggregation would reduce the number of total requests for these two data files from $r_{d_1} + r_{d_2}$ to $r_{d_1} + r_{d_2} - r_{d_c}$. However, the storage size of these two data files will be doubled with a replica containing the aggregated d_1 and d_2. So the storage cost will double and the increase can be more than the reduction in read/write cost caused by aggregation. Therefore, we must carefully evaluate the tradeoff before aggregating data files with concurrent requests.

In the following, we describe how to find the suitable concurrent requested data files to combine. For n data files with r_{d_c} concurrent read requests (all the requests from the same clients and also to the same data files) and r_{d_i} total read requests for data files from i ($i = 1, \ldots, n$), according to Equation (6), (7) and (8), the total money cost M_n is

$$M_n = \sum_{i=1}^{n}(u_{p_j} * D_{d_i} + r_{d_i}(u_{rf} + u_{rs} * D_{d_i})) \tag{13}$$

Where D_{d_i} is the data size of d_i and r_{d_i} is the request frequencies of data file d_i.

If we aggregate these n data files into one data file, then the total money cost M'_n is

$$\begin{aligned} M'_n = &\sum_{i=1}^{n}(u_{p_j} * D_{d_i} + (r_{d_i} - r_{d_c})(u_{rf} + u_{rs} * D_{d_i})) \\ &+u_{p_j}\sum_{i=1}^{n} D_{d_i} + r_{d_c} * (u_{rf} + u_{rs} * \sum_{i=1}^{n} D_{d_i}) \end{aligned} \tag{14}$$

In order to reduce the money cost, we need to guarantee $M'_n < M_n$. So the number of concurrent requests r_{d_c} needs to satisfy:

$$r_{d_c} > \frac{u_{p_j}\sum_{i=1}^{n} D_{d_i}}{(n-1) * u_{rf}}. \tag{15}$$

Using the above equation, the server hosting the data file storage type assignment algorithm first collects the historical data from a time period (e.g., one week) for all the concurrent requests. Since the number of requests for all the data files is variable, we use the average number of concurrent requests within one week to determine which data files need to be aggregated. We define the data file aggregation coefficient Ω as:

$$\Omega = \frac{(n-1)r_{d_c}}{\sum_{i=1}^{n} D_{d_i}} - \frac{u_{p_j}}{u_{rf}} \tag{16}$$

If $\Omega > 0$, we can benefit from data file aggregation; otherwise, there is no benefit from data file aggregation. Furthermore, the group of data files with a higher Ω can achieve a higher cost reduction in total cost from data file aggregation. Next, we will introduce how to deploy this data file aggregation enhancement. Algorithm 2 shows the pseudocode of the concurrent requested data files aggregation algorithm. The server hosting the data file storage type assignment algorithm first collects the request information (including the number of concurrent requests and the related group of data files) from a period of historical data. The server then selects the groups of data files which satisfy Equation (15) and then generates a list containing these selected groups. The server calculates the data file aggregation coefficient Ω for each group of data files in the list, and sorts the list in descending order. The server then selects the top manually set Ψ (which can control the number of data files selected to deploy the data files aggregation) which is manually set, groups of data files to generate the aggregated data files. The concurrent requested data file aggregation procedure runs periodically (e.g., one week) to update the list of the aggregated data files. The overhead of aggregating two 1MB files is in milisecond level and the response time of the aggregated file is similar to that for a non-aggregated file aaccording to our experiment. Furthermore, in order to avoid the situation that concurrent request frequencies drop down substantially after data files aggregation, which would reduce or even nullify the benefit of data file aggregation, we delete an aggregated data file replica if Ω is lower than 0 for a long-term period (e.g., two consecutive weeks).

Algorithm 2: Pseducode of the concurrent requested data files aggregation algorithm.

```
for Collect the concurrent requests information of all the data files;
do
    for each group of data files;
    do
        Calculate the data file aggregation coefficient Ω according to Equation 16);
    Sort the group of data files in descending order according to Ω;
    Select top Ψ groups of data files to generate the aggregated data files;
    if Ω of one group of data files is smaller than 0 then
        Delete the aggregated data file related to this group
    end for
end for
```

6 PERFORMANCE EVALUATION

6.1 Experiment Setup

We use a machine with Intel *i7-8700k*, 32GB memory and two *NVIDIA 1080ti* graphic card to train our DQN. We use the Wikipedia trace and send it to DQN with 128 filters, each of size 4 with stride 1. Results from these layers are then aggregated with other inputs in a hidden layer that uses 128 neurons. We set the default learning rate 0.0027 and use a greedy rate $\epsilon = 0.1$ throughout the experiment, unless we explicitly mention other settings for these parameters in experiments. We implemented this architecture using TensorFlow [7]. For compatibility, we leveraged the TFLearn deep learning library's TensorFlow API to define the neural network during both training and testing.

The trace used in the experiment is Wikipedia trace same to Section 3 which consists the request information of 4000000 data files. We re-formated the trace data into daily request frequencies because the payment made to CSP is calculated by days. Meanwhile, unless otherwise noted, we used a random sample of 80% of our collected trace data as a training set for *MiniCost*; we used the remaining 20% as a test set for *MiniCost* and other comparison methods. The pricing policy used in all the experiments is from Microsoft Azure [3]. For all the plots in this section, we ran the experiments 10 times independently.

Since there is no previous work on the problem handled in this paper, we compare *MiniCost* with four other data storage type assignment algorithms. Noted that, *MiniCost* in Section 6.2 and 6.3 don't have the enhancement method. The comparison methods are: *Hot*: we always put data files into the hot storage type; *Cold*: we always put data files into cold storage type; *Greedy*: we use an offline greedy algorithm for each day. The algorithm calculates the cost difference between putting files into cold and hot including the cost of change the data storage type. Then it assigns the data file

into the storage type with lower total cost. *Optimal* (Offline-brutal-force method): in the simulation, since we know all the request frequency change for all the data files, we can always calculate the total money cost for all the data files in one week for all the possible data storage type assignment plans, and then select the plan with the minimum total money cost. Therefore, *Optimal*, which can generate the best data storage assignment plan, sets the lower bound of the total money cost for all online optimization methods. We use *Optimal* as a baseline to compare our method with the best offline solution.

Our experimental results answer the following questions: 1) How does *MiniCost* compare to the comparison algorithms in terms of the total payment on cloud storage for time-varying usage profiles? We find that, compared with *Optimal*, *MiniCost* is able to outperform the best scheme and the closest to the lower bound (Section 6.2, Figure 7 and 8). 2) How sensitive is *MiniCost* to various algorithm parameters such as the neural network architecture, the learning rate and the greedy rate? Our experiment results provide suggested parameter settings to achieve the best performance between final performance (monetary cost saving) and the convergence speed (Section 6.3). 3) How does *MiniCost* compare to the comparison algorithms in terms of the computing overhead? We find that *MiniCost* can achieve similar computing overhead but much better cost minimization performance compared with *Greedy* algorithm (Section 6.4). 4) How does the data aggregation method enhance the performance of *MiniCost*? Our results show that the enhancement with data file aggregation can lead to substantial improvement in cost savings by balancing the between data file sizes and the number of request frequencies (Section 6.5).

6.2 Performance of MiniCost

Figure 7 shows the normalized money cost per data file (set the money cost per data file from *Optimal* for 7 days as 1) versus the number of days. We see that the cost per data file from using the above four algorithms follow the *Cold*>*Hot*>*Greedy*>*MiniCost*>*Optimal*. Although the difference between each method is small for each data file, since the number of data file is as large as millions

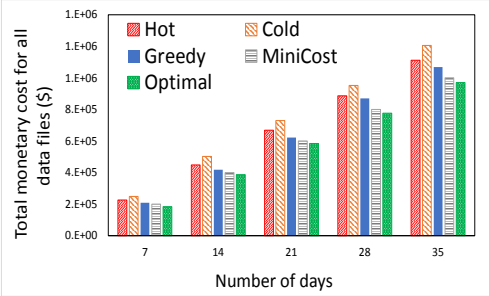

Figure 7: Comparison of total costs.

level, the total saved money can be several grants per day.

Both *Hot* and *Cold* assign all the data files into the same storage type. Since the request frequencies of data files are highly variable (see Figure 2), the costs incurred by the cloud customer using *Hot* and *Cold* are always higher than other methods. *Greedy* makes the decision only based on the cost incurred during one day. Consequently, *Greedy* cannot achieve long-term cost minimization. We observe that it has lower cost compared with *Hot* and *Cold* but higher than *MiniCost*. As previously discussed, *Optimal* performs an exhaustive offline search and is the optimal solution for any online algorithms. It is encouraging to see that the total costs achieved by *MiniCost* is closest to the lower bound from *Optimal*, demonstrating the effectiveness of the RL-based approach that makes the data assignment decisions.

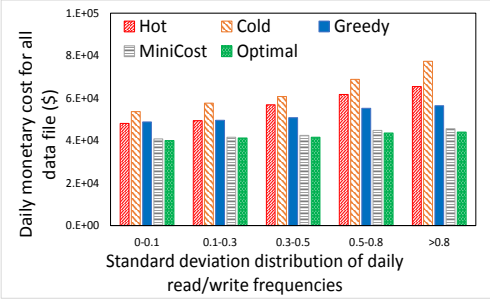

Figure 8: Cost per data file by standard deviations of daily request frequencies.

Figure 8 shows the money cost per data file versus the standard deviations of daily request frequencies for all the data file. We see that the costs follow the order *Cold*>*Hot*>*Greedy*>*MiniCost*>*Optimal*. Meanwhile, for *Hot*, *Cold* and *Greedy*, costs are higher for data files with larger request frequency variabilities. Obviously, *Hot* and *Cold* are expected to perform poorly for data files with large request frequency variabilities over time. While *Greedy* performs better than *Hot* and *Cold* for data files with large request frequency variabilities, it doesn't consider the long-term effect of a storage type assignment decision. In general, the optimal decision for each single day may not be the optimal decision over the long-term (e.g., one week). Thus, *Greedy* cannot achieve close performance to *Optimal*. In comparison, *MiniCost*, with the trained DQN, generates a data storage type assignment plan that not only considers the cost in the next day, but also considers the total cost in the next seven days. Therefore, *MiniCost* can achieve the best performance, which is the closest to the lower bound from *Optimal*.

6.3 Performance of RL with Different Parameter Settings and Overhead

Figure 9 plots the number of step for the RL algorithm to achieve RL convergence versus the different learning rate. We see that the reinforcement learning agent makes the same decision as *Optimal* does in 14 days, as a function of learning rates ranging from 0.00001 to 0.0055. The learning rate represents the speed of that an agent accumulates learned information. We can see that it takes more steps to converge if the learning rate is set too high or too low. The ideal learning rate, in this case, is around 0.0028. When we set a smaller learning rate (from 0.00001 to 0.0028), the agent needs to take more steps to arrive at the optimal decision. For larger learning rate (from 0.0028 to 0.0055), since the size of each single step is large, it may take even more steps for the agent to zigzag towards the optimal decision.

Figure 10 shows the optimal action rate versus the number of steps. The optimal action rate measures the percentage of times

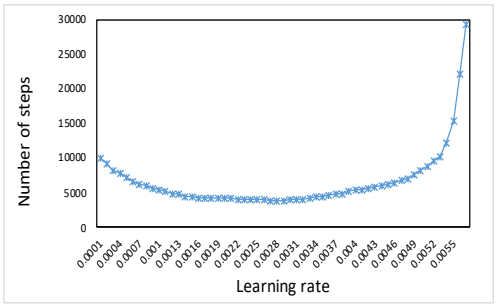

Figure 9: The convergence speed for different learning rates.

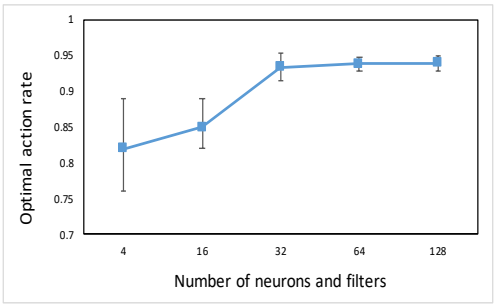

Figure 11: The performance for different number of neurons and filters.

that the agent can take the same action as *Optimal* in 14 days. The optimal action rate is the ratio between the actions made by the RL

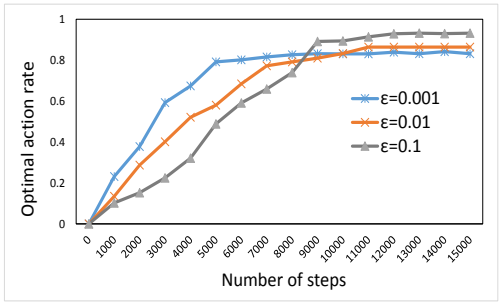

Figure 10: The performance for different greedy rates.

agent and the actions from *Optimal*. It shows the convergence speed of the RL algorithm. We can see that the convergence speed results follow $\epsilon = 0.1 < \epsilon = 0.01 < \epsilon = 0.001$. The final performance results follow the opposite order $\epsilon = 0.1 > \epsilon = 0.01 > \epsilon = 0.001$. Recall that ϵ in RL determines the probability of an agent taking a random decision for the next step. The larger ϵ value lead to a higher exploration rate (the possibility of the agent doesn't select the optimal actions from the neural network), which tends to slow down initial progress but can achieve better final performance. In contrast, the smaller ϵ values lead to a lower exploration rate or equivalently more exploitation. The agent tends to select the decision that has returned higher rewards in the past, and thus makes fast progress. But the final performance may be sub-optimal due to a lack of exploration.

Figure 11 shows the effect of the number of filters and hidden neurons in *MiniCost* learning architecture versus the optimal action rate. One neural network consists many neurons and filters and the number of them can highly affect the performance of RL. For each setting, we repeat the experiment 10 times, and the error bars for the measured optimal action rates are also given in Figure 11. We observe that the performance begins to stabilize once the number of filters and neurons reaches 32. When the number of neurons and filters reaches 64, we observe that the variance of the optimal action rate becomes negligible. Obviously, compared with a small number of filters and neurons (e.g., 4), using a large number (e.g., 64) of neurons and filters can better approximate a more complex value function and thus delivers better and more consistent performance (95% optimal action rate).

6.4 Overhead Performance

Figure 12 shows the computing overhead of each online method for each day in 34 days. We didn't show the computing overhead of *Optimal* since it is an offline method which is not comparable to other

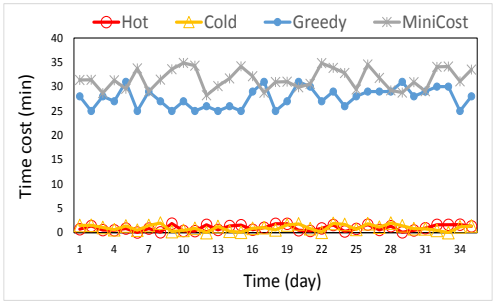

Figure 12: Overhead.

online methods. We can see that the total computing overhead for *Greedy* and *MiniCost* is in the range from 28 minutes to 36 minutes and the computing overhead for *Hot* and *Cold* is around 1 minute. *Hot* and *Cold* produce very small computing overhead, which is only for checking each data storage type. For *MiniCost*, the average time cost for one data file storage type assignment per day is less than 1 millisecond (ms) which is much smaller than the most data transmission latency (10 ms to several hundred ms.) Therefore, the computing overhead of *MiniCost* is negligible and will not affect the data transmission. *MiniCost* can achieve much better cost minimization performance than *Greedy* with similar computing overhead.

6.5 Performance Enhancement via Data File Aggregation

We now compare the performance of MiniCost with and without data file aggregation. In Figure 13, we plot the normalized cost per data file versus the number of days which is how long the cloud customer uses the cloud storage service. We use *MiniCost w/ E* to denote *MiniCost* with the enhancement, and use *MiniCost* to denote *MiniCost* without the enhancement. Because of Equation 16, the enhancement can always achieve a better tradeoff between extra storage cost and the reduced cost caused by request frequencies.

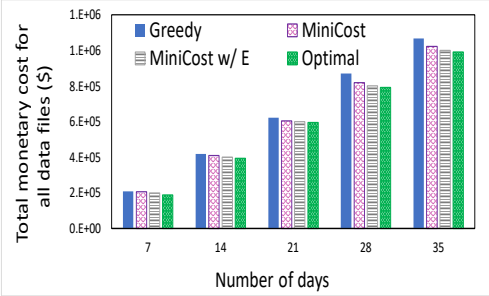

Figure 13: The performance with and without data file aggregation.

We observe that data aggregation leads to further performance improvement and brings the total cost even closer to *Optimal*.

7 CONCLUSION

Minimizing the money cost for cloud storage is important, however, there is no previous work on the data storage type assignment to reduce the total money cost even for a long-term period. In this paper, we first present an analysis of the Wikipedia trace to demonstrate that substantial request frequency variabilities may make it cost-inefficient for a cloud storage service customer. We then model the cost minimization problem using a streamlined MDP formulation. To solve this problem, we introduce an RL based data storage type assignment algorithm that generates data storage type assignment plans periodically according to the request frequencies and CSP pricing policy to minimize the total payment to CSP in long-term. Furthermore, we introduce an effective method to enhance the performance by aggregating data files sharing a large number of concurrent requests. Finally, the results from the trace-driven experiments show that our online RL based method can achieve significant cost savings. In the future, we will extend the method to address a suite of cloud services running on virtual machines.

ACKNOWLEDGMENTS

This research was supported in part by U.S. NSF grants NSF-1827674, CCF-1822965, OAC-1724845, and Microsoft Research Faculty Fellowship 8300751, and AWS Machine Learning Research Awards.

REFERENCES

[1] [n.d.]. Amazon S3. https://aws.amazon.com/cn/s3/, [accessed in Jan. 2020].
[2] [n.d.]. ARIMA model for Time Series Forecasting. https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/, [accessed in Jan. 2020].
[3] [n.d.]. Azure Storage Pricing Policy. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/ , [accessed in Jan. 2020].
[4] [n.d.]. Google Cloud Storage. https://cloud.google.com/storage/, [accessed in Jan. 2020].
[5] [n.d.]. Microsoft Azure. https://azure.microsoft.com/en-us/, [accessed in Jan. 2020].
[6] [n.d.]. Page View statistics from Wikimedia Projects. https://dumps.wikimedia.org/other/pagecounts-ez/, [accessed in Jan. 2020].
[7] Martín A., Paul B., Jianmin C., Zhifeng C., Andy D., Jeffrey D., Matthieu D., Sanjay G., Geoffrey I., and Michael I. 2016. Tensorflow: a system for large-scale machine learning.. In *Proc. of OSDI*.
[8] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. 2010. RACS: a case for cloud storage diversity. In *Proc. of SOCC*.
[9] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review* (2002).
[10] G. Alvarez, E. Borowsky, S. Go, T. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. 2001. Minerva: An automated resource provisioning tool for large-scale storage systems. *Trans. on TOCS* (2001).
[11] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. 2002. Hippodrome: Running Circles Around Storage Administration.. In *Proc. of FAST*.
[12] L. Chen, J. Lingys, K. Chen, and F. Liu. 2018. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proc. of SIGCOM*.
[13] J. E, Y. Cui, M. Ruan, Z. Li, and E. Zhai. 2019. HyCloud: Tweaking Hybrid Cloud Storage Services for Cost-Efficient Filesystem Hosting. In *Proc. of INFOCOM*.
[14] J. Gao, H. Wang, and H. Shen. 2020. Smartly Handling Renewable Energy Instability in Supporting A Cloud Datacenter. In *Proc. of IPDPS*.
[15] S. Hillmer and G. Tiao. 1982. An ARIMA-model-based approach to seasonal adjustment. *J. Amer. Statist. Assoc.* (1982).
[16] R. Howard. 1964. Dynamic programming and Markov processes. (1964).
[17] H. Jin, H. Guo, L. Su, K. Nahrstedt, and X. Wang. 2019. Dynamic Task Pricing in Multi-Requester Mobile Crowd Sensing with Markov Correlated Equilibrium. In *Proc. of INFOCOM*.
[18] Ana K., Heiner L., and Christos K. 2018. Selecta: heterogeneous cloud storage configuration for data analytics. In *Proc. of USENIX ATC*.
[19] Leslieb K. and Andrew L., Michaeland M. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* (1996).
[20] R. Kotla, L. Alvisi, and M. Dahlin. 2007. SafeStore: A durable and practical storage system. In *Proc. of ATC*.
[21] Yang L., Li G., Akara S., and Yike G. 2014. Enabling performance as a service for a cloud storage system. In *Proc. of CLOUD*.
[22] H. Li, L. Zhong, J. Liu, B. Li, and K. Xu. 2011. Cost-effective partial migration of VoD services to content clouds. In *Proc. of Cloud*.
[23] M. Li, C. Qin, J. Li, and P. Lee. 2016. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. *Prof. of ATC* (2016).
[24] G. Liu, H. Shen, and H. Wang. 2017. An economical and SLO-guaranteed cloud storage service across multiple cloud service providers. *Trans. on TPDS* (2017).
[25] H. Madhyastha, J. McCullough, G. Porter, R. Kapoor, S. Savage, A. Snoeren, and A. Vahdat. 2012. scc: cluster storage provisioning informed by application characteristics and SLAs.. In *Proc. of FAST*.
[26] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. 2016. Resource management with deep reinforcement learning. In *Proc. of HotNet*.
[27] H. Mao, R. Netravali, and M. Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *Proc. of SIGCOM*.
[28] W. Mao, Z. Zheng, and F. Wu. 2019. Pricing for revenue maximization in iot data markets: An information design perspective. In *Proc. of INFOCOM*.
[29] V. Mnih, P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*.
[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, and G. Ostrovski. 2015. Human-level control through deep reinforcement learning. *Nature* (2015).
[31] Di Niu, Hong Xu, and Baochun Li. 2012. Quality-assured cloud bandwidth auto-scaling for video-on-demand applications.. In *Proc. of INFOCOM*.
[32] B. Plaza. 2011. Google Analytics for measuring website performance. *Tourism Management* (2011).
[33] Z. Pooranian, K. Chen, C. Yu, and M. Conti. 2018. RARE: Defeating side channels based on data-deduplication in cloud storage. In *Proc. of INFOCOM workshop*.
[34] H. Roh, C. Jung, W. Lee, and D. Du. 2013. Resource pricing game in geo-distributed clouds. In *Proc. of INFOCOM*.
[35] Richard S., Doina P., and Satinder S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *AI* (1999).
[36] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, and V. Panneershelvam. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* (2016).
[37] Y. Song, M. Zafer, and K. Lee. 2012. Optimal bidding in spot instance market. In *Proc. of INFOCOM*.
[38] R. Sutton, A. Barto, and F. Bach. 1998. *Reinforcement learning: An introduction*.
[39] R. Sutton, D. McAllester, S. Singh, and Y. Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Proc. of ANIPS*.
[40] Zhe W., Curtis Y., and Harsha V M. 2015. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services.. In *Proc. of NSDI*.
[41] F. Wang, J. Liu, and M. Chen. 2012. CALMS: Cloud-assisted live media streaming for globalized demands with time/region diversities. In *Proc. of INFOCOM*.
[42] H. Wang and H. Shen. 2018. Proactive incast congestion control in a datacenter serving web applications. In *Proc. of INFOCOM*.
[43] B. Wickremasinghe and R. Buyya. 2009. CloudAnalyst: A CloudSim-based tool for modelling and analysis of large scale cloud computing environments. *Prof. of MEDC* (2009).
[44] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. 2012. Orchestrating the Deployment of Computations in the Cloud with Conductor.. In *Proc. of NSDI*.
[45] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. Madhyastha. 2013. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. of SOSP*.