Journal of Parallel and Distributed Computing 128 (2019) 71–83

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Understanding the latency distribution of cloud object storage systems

Yi Su, Dan Feng *, Yu Hua, Zhan Shi *

Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Ministry of Education of China, School of Computer Science and Technology, Shenzhen Huazhong University of Science and Technology Research Institute, Huazhong University of Science and Technology, Wuhan, 430074, China

HIGHLIGHTS

- We propose an analytic-based model that predicts the response latency distribution of cloud object storage systems.
- Our model considers caching, multiple data chunks, different types of disk operations, and queue discipline of the event-driven programming model.
- We provide the quantitative analysis of the waiting time for being accept()-ed.
- We provide the method for predicting the occurrence of timeouts.

ARTICLE INFO

Article history:
Received 2 March 2018
Accepted 11 January 2019
Available online 18 February 2019

Keywords:
Performance modeling
Cloud object storage
Latency distribution
Queueing theory

ABSTRACT

As a fundamental cloud service, the cloud object storage system stores and retrieves millions or even billions of read-heavy data objects. Serving for a massive amount of requests each day makes the response latency be a vital component of user experiences. Timeout is also a key issue as it has a great impact on the response latency. Due to the lack of suitable understanding on the distribution of the response latency and the occurrence of timeouts, current practice is to use overprovision resources to meet a Service Level Agreement (SLA) on response latency. Hence, firstly, we build a performance model for the cloud object storage system, which assumes no timeout occurring. Our model predicts the percentage of requests meeting an SLA, in the context of complicated disk operations, event-driven programming model and requests waiting for being accept()-ed. Secondly, we propose a method that determines whether or not our model is applicable by predicting the occurrence of timeouts. We evaluate our model with a production system using a real-world trace. In a variety of scenarios, our model reduces the prediction errors by up to 90% compared with baseline models, and its overall average error is 2.63%. Moreover, we could also accurately predict the applicability of our model.

© 2019 Elsevier Inc. All rights reserved.

1. Introduction

Cloud object storage systems, like Amazon S3 [1] and OpenStack Swift [22], play an important role in modern web-based applications. In addition to cloud object storage systems from public cloud providers, industrials also build and maintain their own object storage systems, e.g. Facebook uses Haystack [2] for storing photos, LinkedIn employs Ambry [21] for holding media objects, and Wikipedia has deployed OpenStack Swift [36] clusters as media object stores for efficiency and scalability. As one of the fundamental cloud services, the cloud object storage system generally stores and retrieves millions or even billions of diverse data objects (also called blobs), including photos, audios, videos, documents, etc. Moreover, the cloud object storage system may also directly serves millions of latency sensitive Internet users.

In order to understand the relationship between the distribution of the response latency and the resource allocation, we develop an analytic-based performance model for the cloud object storage system using event-driven programming model (e.g. one process handles multiple transactions in time-division multiplexing manner with an event loop using epoll/poll/select function). The event-driven programming model has been widely adopted by cloud object storage systems (detailed in Section 2). Different from existing analytic-based models [4,11,14,31] that predict the average performance metrics (e.g. throughput, mean response latency), our model predicts the percentage of requests meeting a requirement of response latency, e.g. 95% of the requests could

* Corresponding authors.
E-mail addresses: suyi@hust.edu.cn (Y. Su), dfeng@hust.edu.cn (D. Feng), csyhua@hust.edu.cn (Y. Hua), zshi@hust.edu.cn (Z. Shi).

https://doi.org/10.1016/j.jpdc.2019.01.008
0743-7315/© 2019 Elsevier Inc. All rights reserved.

be responded in at most 100 ms. The response latency percentile is superior to the average metrics in the context of the cloud object storage system for the following reasons. First, the response latency is a key performance metric for cloud object storage systems due to having a great impact on user experiences, which are closely related to revenues. Second, even 1% of traffic corresponds to a significant volume of user requests for cloud object storage systems [6].

Considered the large volume of data objects and the long tail distribution of data accessing [2,12], cost-efficiency is one of the main concerns for cloud object storage systems. And a validated performance model of the cloud object storage system, which is the basis of capacity planning, plays an important role in achieving the cost-efficiency. *Capacity Planning* determines the number of resources needed for the system with the workload anticipation and the Service Level Agreement (SLA). Besides the initial deployment, cloud providers also need to perform capacity planning whenever cluster expansion occurs. The ever-growing number of blobs in the cloud object storage system [21] magnifies the necessity of cluster expansion. In addition to capacity planning, a simple yet accurate performance model is also important for performing the "what-if" analysis (the process of changing the inputs to see how those changes will affect the outcomes) for the following applications. (1) *Overload Control*, which enables the systems to turn away excess requests during transient overloads; (2) *Bottleneck Identification*, which locates the performance bottleneck from thousands or hundreds of devices; (3) *Elastic Storage*, which dynamically powers on and powers off storage nodes in reaction to workloads for energy savings or operating cost savings (the system should meet the performance requirements at the same time).

Modeling of multi-tiered Web applications [4,11,14,31] and storage systems [15,32,38] is well studied. However, extending these models to the scenarios of the cloud object storage system is nontrivial due to the following reasons. (1) **Diverse Disk Operations**. At the backend server of the cloud object storage system, serving a request involves several kinds of operations, including index lookup, metadata read, and data read. The index is used to locate the data object on the storage device, e.g. inodes of the local file system. The metadata is the attributes of the data object, e.g. checksum, create-time, user-defined attributes, etc. These indexes and metadata are stored in the same storage device along with the corresponding data objects. However, cost-sensitive cloud providers prefer not providing "enough" memory for caching the indexes and metadata at backend servers [2] (detailed in Section 2). Hence, the operations, like index lookup and metadata read, should be modeled, due to having a possibility of accessing disks. Moreover, these diverse operations have different performance characteristics and cache miss ratios. Models targeting other usage scenarios fail to deal with this complexity. (2) **Data Chunking**. The event-driven programming model uses the First Come First Serve (FCFS) queue to schedule operations. Hence, in order to prevent the system from being blocked by the operations that last a long time (e.g. the large data read), the cloud object storage system reads and transmits the data chunk by chunk in the context of the event-driven programming model, instead of reading and transmitting the whole data object at once. After having started the transmission of a data chunk, the system would then perform the next operation, which belongs to a different request, in the FCFS queue. As a result, the cloud object storage system processes different requests in an interleaving manner (detailed in Section 3.2). Models targeting other systems fail to address this particular scenario of the cloud object storage system. (3) **Waiting Time for Being Accept()-ed**. The model has to quantify the waiting time of requests for being accept()-ed at the backend servers. Accept() is a socket API function. The server uses accept() function to initialize the connection for a request. And the request has to wait in the connection pool before being accept()-ed by the server. The waiting time has a significant impact on response latency of the cloud object storage system. Tim Brecht et al. [3] first study this issue by comparing the throughput and average response latency of different accept() schemes. However, to the best of our knowledge, there is no quantitative analysis of the waiting time for being accept()-ed.

We focus on building an analytic-based model that can capture the impact of the factors mentioned above. Although our model requires some benchmarking based parameters (detailed in Section 4), the benchmarking in our model is independent of workloads, which makes our model differ from simulation-based models. The workload is always a key factor for benchmarking in the simulation-based models, which makes the simulation-based models vulnerable to the changes of the workload.

We perform the evaluation using an OpenStack Swift testbed by replaying a real-world trace (accessing trace of media objects from Wikipedia [30]). In a variety of scenarios, the prediction error of our model is 2.63% on average. Moreover, in cloud object storage systems, the heavy workload would trigger timeouts and retries. However, due to not considering the impact of timeouts and retries, our model does not work in the context of such heavy workloads. Hence, given the inputs, we also propose a method to predict whether or not our model is applicable. According to the evaluation, the method could provide accurate results on predicting the applicability of our model.

In summary, our contributions include:

(1) **The abstraction of union operation**: We build an analytic-based performance model for the cloud object storage system. The model packs complicated operations in request processing into queueing-theory friendly operations (the union operation). This abstraction comprehensively leverages caching, multiple data chunks, different types of disk operations, and queue discipline of the event-driven programming model, which fully meets the needs of the overall model.

(2) **Modeling the waiting time for being accept()-ed**: We explore and exploit the fact that the requests waiting for being accept()-ed at the backend servers may introduce a significant impact on response latency of the cloud object storage system. Furthermore, we also provide a quantitative analysis by revealing the relationship between the waiting time for being accept()-ed and the status of request processing queues.

(3) **Predicting the model applicability**: We propose a method to predict whether or not our model is applicable. The prediction results of our model could be inappropriate under the workload that triggers timeouts and retries. We discuss all kinds of timeouts that may occur in cloud object storage systems and predict the occurrence of timeouts. Then, we determine the applicability of our model according to the prediction results of timeout occurrence.

(4) **Prototype implementation and evaluation**: We implement all components of our model based on OpenStack Swift and evaluate the accuracy of them using a real-world trace in various scenarios. Moreover, our implementation is open-source and available from https://github.com/ysu-hust/cosmodel.

The remainder of this paper is structured as follows. Section 2 provides backgrounds of the cloud object storage system. We describe our model and the methods to estimate model parameters in Section 3 and Section 4, respectively. Section 5 introduces the method to determine the model applicability. Section 6 presents evaluation results. Related work is discussed in Section 7. Finally, Section 8 concludes this paper.

2. Background

The cloud object storage system is a two-tiered web application, as shown in Fig. 4. The servers in the frontend tier are responsible for routing requests to their corresponding storage devices, and the

servers in the backend tier are responsible for managing storage devices and storing data objects. Each storage device has dedicated process(es) for handling its corresponding requests at the backend server. The process of a storage device performs the following operations in sequence for handling a read request: (1) determining the data object for serving the request (request parsing); (2) locating the data object on the storage device (index lookup); (3) getting the attributes of the data object (metadata read); (4) reading the data by a chunk (data read). For the systems exploiting local file system for managing data objects at backend servers, e.g. OpenStack Swift, the specific operations for the process of a storage device are: (1) converting the name of the data object to the local file path (request parsing); (2) opening the file (index lookup); (3) reading the extended attributes of the file (metadata read); (4) reading the file by a chunk (data read).

In the context of minimizing the TCO (Total Cost of Ownership), the cloud providers prefer the cheap storage device with large capacity (e.g. HDD) rather than the expensive storage device with high performance (e.g. SSD). It is because the cloud object storage system needs capacity more than performance. For example, in the OpenStack Swift cluster of Wikipedia [27], the total consumed storage capacity is about 670 TB and the *maximum* aggregated throughput of the backend tier last year is only about 1.2 GB/s. Considered that the response latency of HDD disks is in milliseconds, the HDD disk is sufficient for meeting the performance requirement of cloud object storage systems. In fact, production systems [2,21,36] commonly store data in cheap HDD disks, instead of much more expensive SSD disks (a 2 TB low-end SSD drive costs about $500, 10 times the 2 TB HDD drive). Furthermore, in order to reduce the TCO, cost-sensitive cloud providers also prefer not providing "enough" memory for caching the Index and Metadata (I&M) at backend servers [2]. It is because (1) there could be a large volume of I&M due to a large number of data objects, and (2) the ratio of the I&M volume to the data volume could be high as the majority of data objects are of small size [2,18,21,23] in production environments. Suppose that the average size of data objects is about 50 kB [21,23], and the average size of I&M of data objects is about 1 kB altogether [23]. There are about 20 GB I&M for each 1 TB data in this scenario. Hence, caching all I&M in memory is not cost-effective. As a result, while processing a request at the backend tier, all of the operations of different performance characteristics (including index lookup, metadata read, and data read) have a possibility of accessing HDD disks due to the long tail distribution of data accessing [2,12]. These specificities introduce new challenges for modeling the cloud object storage system. Methods of reducing the size of the indexes have been proposed [2,21] so that the majority of the index can be cached in memory. However, systems without such optimization, e.g. OpenStack Swift, are still widely deployed in production environments [23,24,36] due to being reliable and mature, and the system with such optimization is only a special case (index lookup rarely accesses the disks) of the systems addressed by our model.

The thread-per-connection and the event-driven programming model are the main strategies for handling concurrency for cloud object storage systems. This paper concentrates on modeling systems using the event-driven programming model because the event-driven programming model is widely adopted in many famous cloud object storage systems (like OpenStack Swift, Ceph) and production environments [5,24]. Moreover, the event-driven programming model is superior to the thread-per-connection architecture in both throughput and tail response latency [7].

3. Cloud object storage system modeling

In this section, we present the queueing-theory based model for the cloud object storage system along with the assumptions of our model.

3.1. Assumptions for modeling

We develop our model under the following assumptions.

(1) **Poisson arrival**. Our model assumes Poisson arrival of requests for all the cases studied. For scale-out workloads, Poisson process is considered a good model, which approximates the real arrival process with reasonably small errors [17].

(2) **Read heavy workloads**. The model does not consider WRITE and DELETE requests. The workloads for cloud object storage systems are read dominant [2,21,36], and the data objects are written once, read often, never modified and rarely deleted. For example, read traffic is $>$ 99% in Wikipedia OpenStack Swift cluster [36], $>$ 95% in LinkedIn Ambry [21], and $>$ 98% in Facebook Haystack [2], etc.

(3) **Sufficient resources of computation and network**. Resources of computation and network are commonly sufficient in cloud object storage systems. Take the OpenStack Swift cluster of Wikipedia [36] as an example. In the recent one year, the *maximum* aggregate arriving rate of requests is under 2000 requests per second, and for any single backend server, the throughput is only about 20 MB/s for most of the time and about 50 MB/s for the *maximum*. At the same time, a 2.4 GHz CPU core could perform 25 Million instructions per second and the 1 Gbps Ethernet provides about 100 MB/s network bandwidth.

(4) **Steady state**. Cloud object storage systems commonly work in the steady state due to the stable workloads. For example, it takes about 10 h for the request arriving rate to increase from 700 requests per second to 1500 requests per second in the OpenStack Swift cluster of Wikipedia [36].

(5) **Normal status**. The model does not consider the impact of timeouts, retries, and the software limits (e.g. system connection pool size, maximum concurrency level, etc.). Because there would be a lot of SLA violations when such software mechanisms and limitations dominate the system performance. Instead of accurate performance metrics, it is enough to know that the system does not perform well in such situations.

3.2. Performance modeling at backend tier

When a request arrives at the backend tier, the request enters one of the request processing queues of the corresponding storage device. Each storage device has one or multiple queues determined by the number of processes dedicated to the storage device. We first build the performance model for the scenario of one queue and then extend the model to the scenario of multiple queues. Suppose that the number of queues for one storage device is N_{be}.

When $N_{be} = 1$: Serving a request involves performing the following operations in sequence at backend servers, request parsing, index lookup, metadata read, and data read. As a result, the request processing queue turns into an operation queue filled by diverse operations. The left side queue in Fig. 1 is the operation queue, and we observe that the process performs operations of different requests in an interleaving manner. It is because the process reads and transmits data chunks one by one, and performs network I/O asynchronously. After having started sending a data chunk to the frontend tier, the process switches to deal with other requests instead of waiting for the data transmission to complete to send another data chunk. In order to model this complicated operation queue, we pack the diverse operations of request parsing, index lookup, metadata read, and data read, into a queueing-theory friendly operation, and we call it the union operation. A union operation starts with a request parsing operation and includes the following operations that are not request parsing operations. Hence, each union operation may contain operations of different requests. As a result, we transform the original operation queue into a queue of union operations, which is shown as the right side

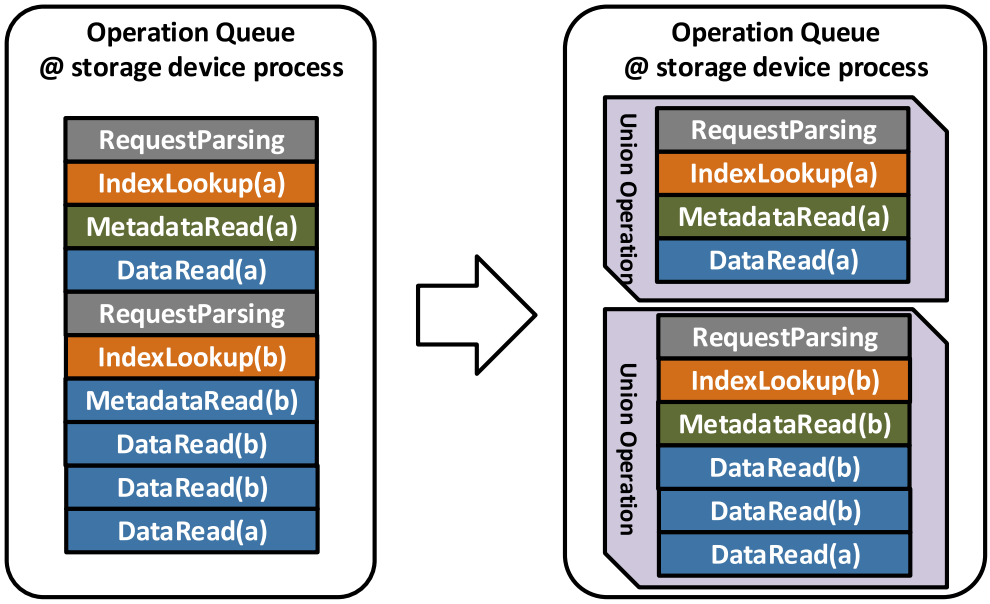


Fig. 1. Queue of a storage device (single process per storage device).

queue in Fig. 1. With the assumption of Poisson arrival (detailed in Section 3.1), we model the queue of union operations as an M/G/1 queue (a queueing system of Poisson arrivals, generally distributed service time, and a single server). To solve this model, we have to find the service time distribution of the union operation.

For a storage device, let r, r_{data} denote the arrival rate of its requests and data read operations. r_{data} is determined by r along with the chunk size and the size of data objects. It is convenient to measure or calculate these metrics. With the caching mechanism, operations including index lookup, metadata read, and data read, can be served either from memory or disk. Let $m_{index}, m_{meta}, m_{data}$ denote the cache miss ratios of these operations respectively. The $index_d(t), meta_d(t), data_d(t)$ denote the probability density functions (pdf.) of the latency while these operations are served from disk, and we get the $index_d(t), meta_d(t), data_d(t)$ via benchmarking (detailed in Section 4). As the latency of memory is negligible, we approximate it with 0. Let $parse_{be}(t), index(t), meta(t), data(t)$ represent the pdf. of service time for request parsing, index lookup, metadata read, and data read respectively. Then we can write

$$index(t) = index_d(t)m_{index} + \delta(t)(1 - m_{index}),$$
$$meta(t) = meta_d(t)m_{meta} + \delta(t)(1 - m_{meta}),$$
$$data(t) = data_d(t)m_{data} + \delta(t)(1 - m_{data}),$$

where, $\delta(t) = \begin{cases} +\infty, t = 0 \\ 0, t \neq 0 \end{cases}$ is the DiracDelta function.

Consider a data read not following its corresponding metadata read operation as the extra data read. It is safe to assume that the arrivals of the extra data chunk reads of different requests are independent because they are issued by different processes from the frontend tier [17]. In other words, we could assume that the arrival of extra data read follows Poisson arrival. So, we could use the Poisson distribution to model the amount of extra data reads in one union operation, and the average number of extra data reads in one union operation is $p = \frac{r_{data} - r}{r}$. In summary, the pdf. and mean value of the service time of the union operation are

$$B_{be}(t) = \sum_{j=0}^{\infty}[\frac{p^j e^{-p}}{j!}(parse_{be} * index * meta * data^{j+1})(t)],$$

$$\bar{B_{be}} = \sum_{j=0}^{\infty}[\frac{p^j e^{-p}}{j!}(\bar{parse}_{be} + \bar{index} + \bar{meta} + (j+1)\bar{data})],$$

Where, $\bar{parse}_{be}, \bar{index}, \bar{meta}, \bar{data}$ are the average latencies of request parsing, index lookup, metadata read and data read respectively. Since the exact correlation among the service time of different operations is unknown, we approximate the operations using independent operations here. Hence, the service time of the union operation is the convolution of the service time of involved operations. We discuss the impact of this approximation in Section 6.3.

Finally, we get the Laplace Transform of the waiting time pdf. of the backend queue based on Pollaczek–Khinchin formulas [29].

$$\mathcal{L}[W_{be}](s) = \frac{(1 - \bar{B_{be}}r)s}{r\mathcal{L}[B_{be}](s) + s - r} \tag{1}$$

The process uses the metadata to form the response headers and starts responding a request after it gets the metadata and the first data chunk. So the pdf. of the response latency at backend tier is

$$S_{be}(t) = (W_{be} * parse_{be} * index * meta * data)(t). \tag{2}$$

When $N_{be} \in \{2, 3, 4, 5 \ldots\}$: Fig. 2 displays the queueing status for a storage device with multiple processes at the backend server. When a request arrives at the backend server, one of the N_{be} processes accepts the request. Then the request turns into a bunch of operations and enters the operation queue of the corresponding process. Serving some operations requires accessing disk due to the cache miss, and such operations will enter the operation queue of the disk. The process will be blocked until the operation, which enters the disk, completes. There is no ready-to-use solution to predict the distribution of the response latency for such a queueing network, as discussed in Section 7. Moreover, we cannot directly apply the solution for $N_{be} = 1$ here. In order to solve this problem, we continue relying on the abstraction of union operation. The key idea is to transform the queueing network of $N_{be} > 1$ to the queueing network of $N_{be} = 1$.

In order to conduct the transformation, we first divide the union operation into two classes: the Cache Hit Union operation (CHU) and the Cache Miss Union operation (CMU). A union operation is a CHU if it does not contain any operation accessing the disk, otherwise, it is a CHM. As all of the cache miss operations enter one operation queue of the disk as shown in Fig. 2, the CMUs in different process's operation queues are performed sequentially.

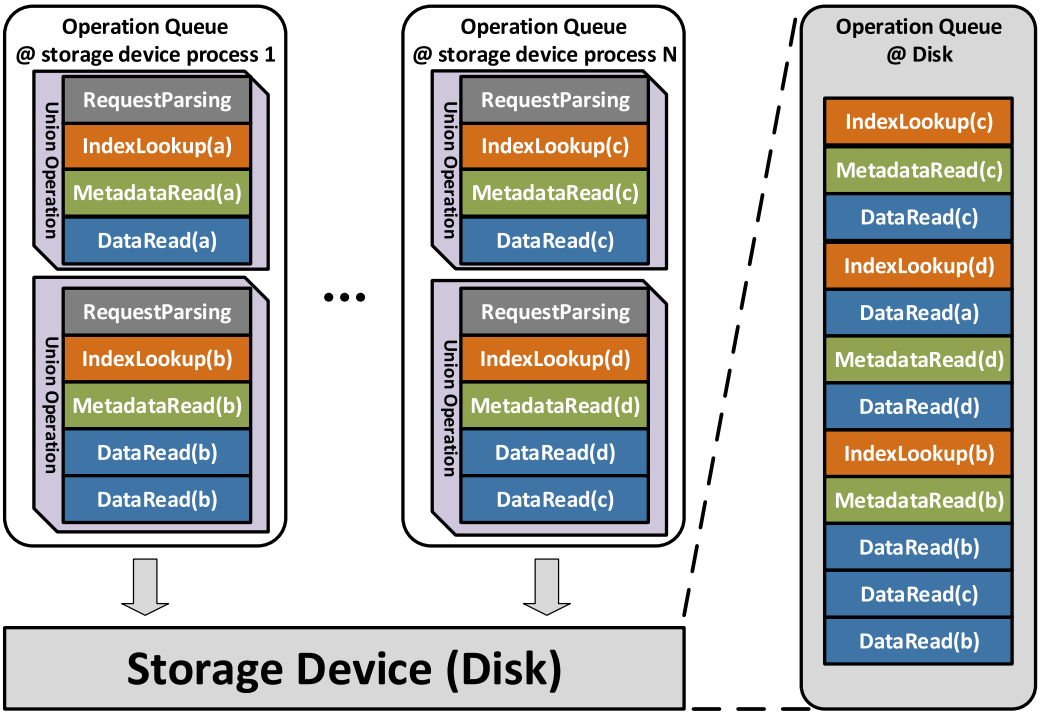


Fig. 2. Queues of a storage device (N_{be} processes per storage device).

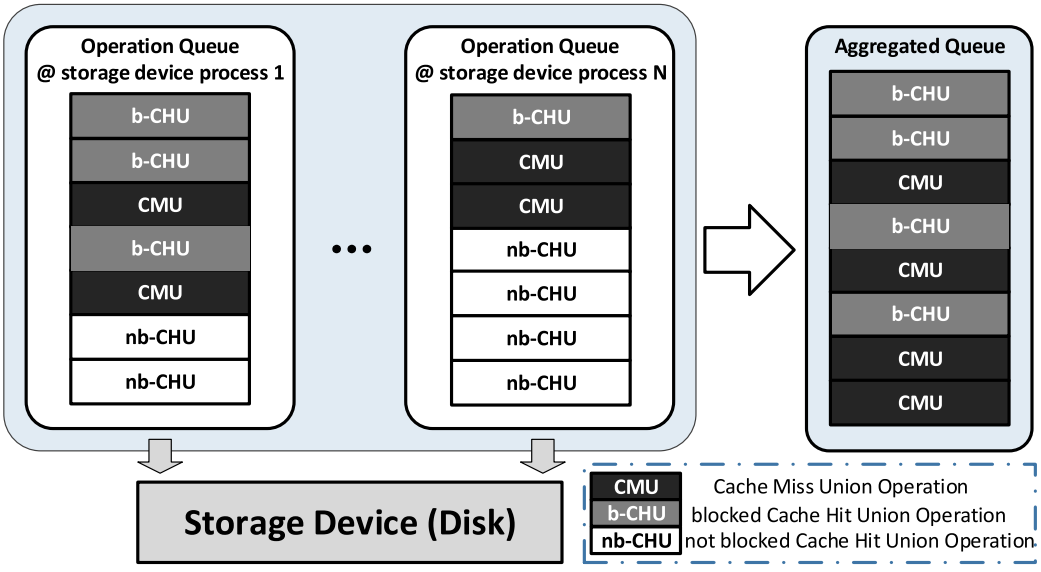


Fig. 3. The aggregated queue that approximates the multiple queues of a storage device.

According to whether or not a CHU is blocked by a CMU, we further divide the CHU into two classes: the blocked CHU (b-CHU) and the not blocked CHU (nb-CHU). Considered the huge performance gap between memory and disk, we assume that the service time of a CHU is negligible. Hence the response latency of a CHU is equal to its waiting time in the operation queue of a process. In another word, a CHU's response latency is determined by the time of being blocked by CMUs. Under such assumption, the nb-CHUs are finished processing instantly. Therefore we could ignore nb-CHUs while analyzing the queueing network. Due to the sequential execution of CMUs, we use an aggregated queue to approximate the multiple operation queues for CMUs and b-CHUs as shown in Fig. 3. The aggregated queue is a queueing network that is same as the queueing network of $N_{be} = 1$. To solve this model, we have to: (1) find the proportion of nb-CHUs (P_{nb-CHU}) and (2) determine properties of the aggregated queue, including the arriving rate, the mean value and the distribution of service time.

A CHU is an nb-CHU only if there are idle process(es) when it arrives, and a process is idle when its operation queue is empty. The number of union operations in the system determines the probability of idle process. We only consider CMUs while counting the union operations in the system due to the negligible service time of CHUs. Suppose $l_{CMU}(j)$ is the probability mass function (pmf.) of the number of CMUs in the system. In order to get $l_{CMU}(j)$,

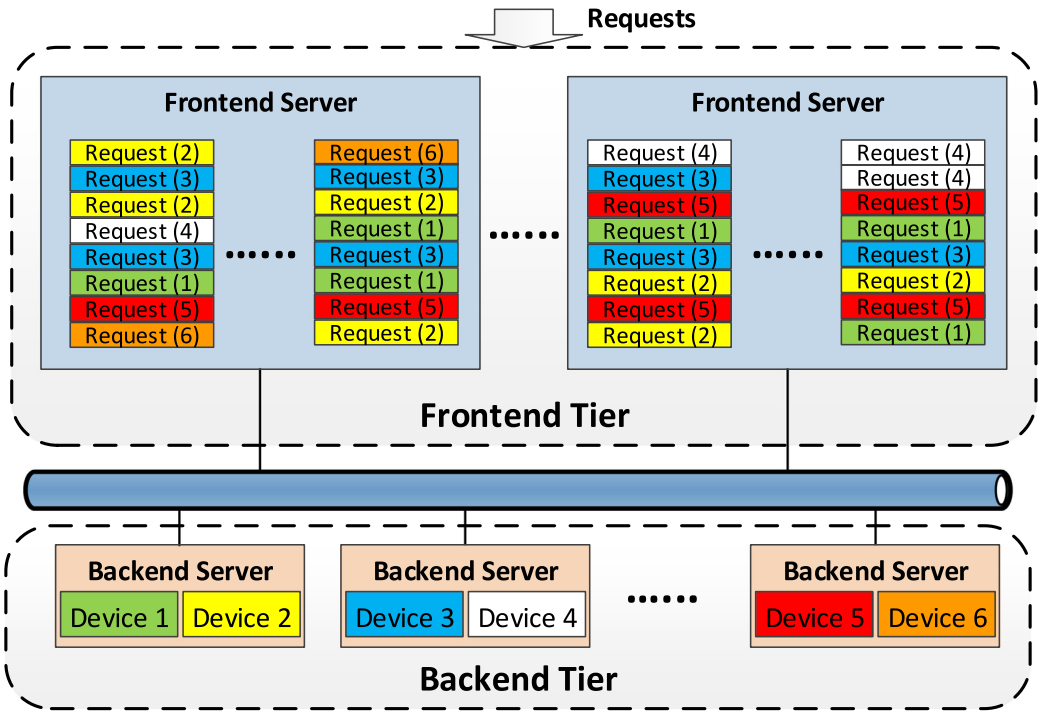


Fig. 4. The request queues at the frontend servers.

we assume all CMUs enter an M/G/1 queue, which is reasonable due to the sequential execution of CMUs. Hence $l_{CMU}(j)$ is equal to the queue length pmf. of the M/G/1 queue. Suppose m_{union} is the probability that a union operation is a CMU. According to the definition of CMU, we have

$$m_{union} = 1 - \sum_{j=0}^{\infty}[\frac{p^j e^{-p}}{j!}(1 - m_{index})(1 - m_{meta})(1 - m_{data})^{j+1}].$$

Suppose $\bar{B_{CMU}}, V_{CMU}, B_{CMU}(t)$ are the mean value, variance and pdf. of the CMU service time, respectively. Suppose $\bar{B_{CHU}}, V_{CHU}, B_{CHU}(t)$ are the mean value, variance and pdf. of the CHU service time, respectively. It is easy to obtain $\bar{B_{CHU}}, V_{CHU}, B_{CHU}(t)$, then we could obtain $\bar{B_{CMU}}, V_{CMU}, B_{CMU}(t)$ with following formulas:

$$\bar{B_{CMU}} = (\bar{B_{be}} - (1 - m_{union})\bar{B_{CHU}})/m_{union},$$
$$V_{CMU} = (V_{be} + (1 - m_{union})V_{CHU})/m_{union},$$
$$\mathcal{L}[B_{CMU}](s) = (\mathcal{L}[B_{be}](s) - (1 - m_{union})\mathcal{L}[B_{CHU}](s))/m_{union},$$

where V_{be} is the service time variance of the overall union operation. According to Pollaczek–Khinchin formulas [29], a close form queue length pmf. of an M/G/1 queue does not always exist. Therefore, we use an estimating approach from Myers and Vernon [19]. Then we can write

$$l_{CMU}(j) = u_{CMU} q_{tmp}^j (1 - q_{tmp}),$$

where $u_{CMU} = \bar{B_{CMU}} m_{union} r$ is the utilization of the M/Q/1 queue, $q_{tmp} = \frac{u_{CMU}(c_v^2+1)}{2+u_{CMU}(c_v^2-1)}$, and $c_v^2 = \sqrt{V_{CMU}}/\bar{B_{CMU}}$.

Suppose there are j CMUs in the system. For a CMU, we assume that the possibility of this CMU in either operation queue is equal. Then, calculating the distribution of the j CMUs among the multiple operation queues is a classical problem (distributing distinct balls into distinct bins) in Combinatorial Theory [10]. When $0 \le j < N_{be}$, there must be at least one idle process. When $j \ge N_{be}$, we could calculate the probability of idle process $P_{idle}(j) = 1 - \frac{m!StirlingS2(j,N_{be})}{N_{be}^j}$, where $StirlingS2(x, y)$ is the Stirling number of the second kind, which is the number of ways to puts x balls into y non-empty bins.

Finally, we can get the proportion of nb-CHUs (or the probability that a union operation is a nb-CHU):

$$P_{nb-CHU} = (1 - m_{union})(\sum_{j=0}^{N_{be}-1} l_{CMU}(j) + \sum_{j=N_{be}}^{\infty}[l_{CMU}(j)P_{idle}(j)]).$$

Suppose the arriving rate of the aggregated queue is r_{ag}, then $r_{ag} = (1 - P_{nb-CHU})r$ as the aggregated queue only contains CMUs and b-CHUs. For the operations of index lookup, metadata read, and data read in the aggregated queue, let $m_{index-ag}, m_{meta-ag}, m_{data-ag}$ denote the cache miss ratios of these operations, respectively. Then we have $m_{index-ag} = m_{index}/(1 - P_{nb-CHU})$, $m_{meta-ag} = m_{meta}/(1 - P_{nb-CHU})$, and $m_{data-ag} = m_{data}/(1 - P_{nb-CHU})$. With r_{ag}, $m_{index-ag}$, $m_{meta-ag}$, and $m_{data-ag}$, we could obtain the waiting time pdf. of the aggregated queue ($W_{ag}(t)$) using the same method for solving the queue model in $N_{be} = 1$ scenario. Then the pdf. of the waiting time (W_{be}) and the pdf. of the response latency ($S_{be}(t)$) at the backend tier are:

$$W_{be}(t) = P_{nb-CHU}\delta(t) + (1 - P_{nb-CHU})W_{ag}(t) \quad (3)$$
$$S_{be}(t) = (W_{be} * parse_{be} * index * meta * data)(t). \quad (4)$$

3.3. Performance modeling at frontend tier

The response latency of a request at the frontend tier contains three main components: the queueing latency at the frontend tier, the waiting time for being accept()-ed at the backend tier, and the response latency of the storage device at the backend tier.

Queueing latency at the frontend tier: In the frontend tier of homogeneous servers, the processes in the frontend tier are identical to each other. So, the distribution of the overall queueing latency is the same as that of any single process. In the frontend tier of heterogeneous servers, we can divide the servers into several sets of homogeneous servers, and calculate distributions of the queueing latency for each set separately. Fig. 4 shows the queues of different processes in the frontend tier. Suppose that there are N_{fe} frontend tier processes, and the requests arriving rate is R. So,

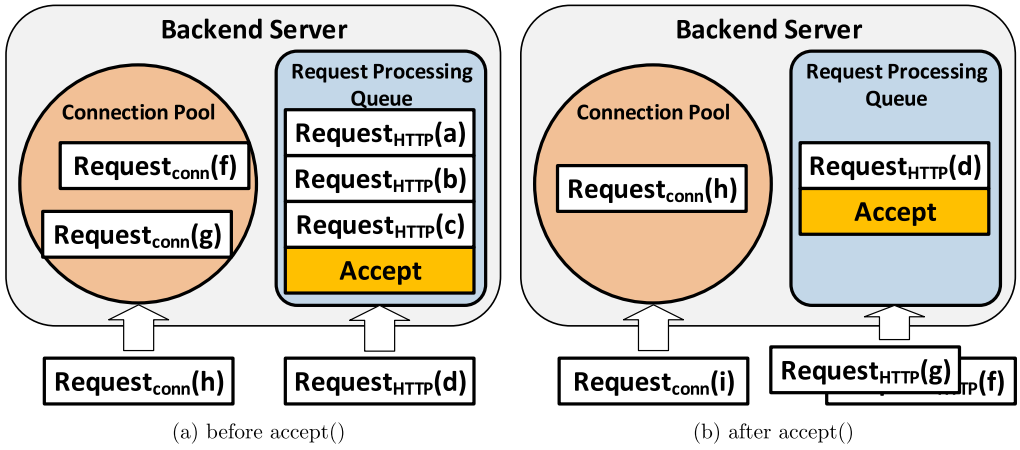


Fig. 5. Waiting time for being accept()-ed.

arriving rate for one process P_i at the frontend tier is $r_i = \frac{R}{N_{fe}}$. The pdf. and mean value of request parsing time for processes in the frontend tier are $parse_{fe}(t)$ and $\bar{parse}_{fe}$. We could also use the M/G/1 queue to model the queue of one process in the frontend tier [17]. Then, the Laplace transform of queueing latency pdf. is

$$\mathcal{L}[S_q](s) = \frac{(1 - \bar{parse}_{fe} r_i)s\mathcal{L}[parse_{fe}](s)}{r_i\mathcal{L}[parse_{fe}](s) + s - r_i}. \tag{5}$$

Waiting time for being accept()-ed: Sending a request from the frontend tier to the backend tier involves two steps in sequence, building a TCP connection and sending an HTTP request. However, a connecting request from the frontend tier has to wait in the connection pool before being accept()-ed by a process of the storage device at the backend server. Since the accept() operation is scheduled as identical as ordinary operations for processing requests, the accept() operation also has to wait in the request processing queue. Fig. 5a shows the situation before the connecting requests of "f" and "g" being accept()-ed. They have to stay in the connection pool until the process finished processing the HTTP requests of "a", "b", and "c". Fig. 5b shows the situation after the connecting requests of "f" and "g" are accept()-ed. After being accept()-ed, the frontend servers will send the HTTP requests of "f" and "g" to the backend server according to their queueing statuses. Given an accept() operation, its life begins at the last time when the point requests in the connection pool are accept()-ed, and its life ends at the time point when the requests are accept()-ed. The arrival of an accept() operation refers to the accept() operation being appended to the tail of requests processing queue and starting to wait for being performed. Assuming that the arrival of accept() operations follows the Poisson process, the lifetime pdf. ($A(t)$) of accept() operations is the same as the waiting time pdf. of the request processing queue at the backend server according to the PASTA theorem [37]. A connecting request may arrive at any time point during the lifetime of an accept() operation. Suppose that the waiting time pdf. of the connecting request for being accept()-ed is $W_a(t)$. Then $W_a(t) = \int_{x \geq t}(A(x)\frac{x-t}{x})dx$. In our model, we use an approximation of $W_a(t)$, which assumes that the waiting time equal to the accept() lifetime for all of the connecting requests that arrive during the life of the accept() operation. So the pdf. of waiting time for being accept()-ed is

$$W_a(t) = A(t) = W_{be}(t). \tag{6}$$

Our approximation overestimates the waiting time of the connecting requests, which arrive after the life of the accept() having already started. This overestimation increases as the length of requests processing queue increases. We evaluate the accuracy of the model of waiting time for being accept()-ed along with its approximation in Section 6.3.

In summary, at the frontend tier, the response latency pdf. ($S_{fe}(t)$) of a storage device can be calculated by combining (using convolution) the 3 latency components: pdf. of the queueing latency at the frontend tier (S_q), pdf. of waiting time for being accept()-ed at the backend tier (W_a), and pdf. of the response latency at the backend tier (S_{be}). The S_{be} is from Section 3.2.

$$S_{fe}(t) = (S_q * W_a * S_{be})(t) \tag{7}$$

3.4. System modeling

Suppose the set of the storage devices is D. Given a storage device D_j, $D_j \in D$, the cumulative distribution function (cdf.) of the response latency for device D_j at the frontend tier is $S_j(t)$. The requests arriving rate of D_j is r_j. The latency distribution of the whole system is actually a mixture distribution. The mixture components are the latency distributions of each storage device at the frontend tier, and for each storage device, the mixture weight is its workloads proportion. Since we have already known the distribution of the response latency for each storage device at the frontend tier, we could calculate the cdf. of the response latency for the whole system ($S(t)$) with the following formula:

$$S(t) = \frac{\sum_{D_j \in D}[r_j S_j(t)]}{\sum_{D_j \in D} r_j}. \tag{8}$$

4. Estimating the model parameters

Our model requires several parameters as inputs, and the various parameters of our model fall into two categories: device performance properties and system online metrics. In this section, we describe the methods of estimating these various parameters required by our model.

4.1. Device performance properties

The distribution of disk service time. We assume random accessing of data objects for a storage device because the requests come from millions of users and the data objects are randomly distributed among storage devices based on hashing. Hence, we benchmark the disk with the following steps. First, we fill the disk with data objects; Second, we sequentially access (perform the operations of index lookup, metadata read, and data read) a number of randomly selected data objects, and record the latency for each operation. We also limit the max amount of outstanding

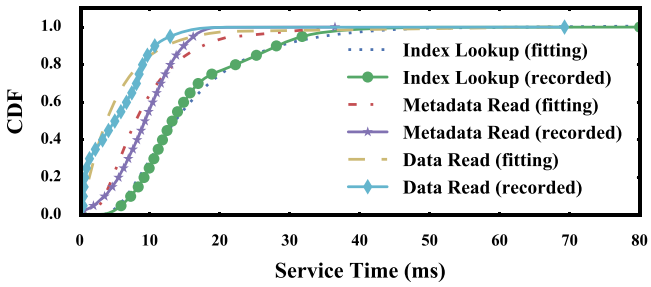


Fig. 6. The fitting results of the disk service time.

operation to 1 to avoid queueing of operations; Finally, we use the distribution fitting to get the distribution of disk service time for different kinds of operations separately. Suppose the $f(t)$ is the pdf. of the distribution, which is used to fit the recorded latencies. The Laplace Transform of $f(t)$ ($\mathcal{L}[f(t)]$) or an analytical approximation of the $\mathcal{L}[f(t)]$ should exist, and the mean value of the distribution should exist as well, because our model needs them for performing the calculations as shown in Section 3. Moreover, a mixture distribution could be used for fitting the recorded latencies, as long as the mixture distribution satisfies the above requirements. For our testbed, we test 4 distributions for fitting, including the Exponential, Degenerate, Normal, and Gamma distribution. The Gamma distribution demonstrates the best result among them. Fig. 6 shows the fitting results using Gamma distribution. The Gamma distribution is defined by two parameters k (shape parameter) and l (rate parameter), the Laplace Transform of its pdf. is $\mathcal{L}[B](s) = l^k(s+l)^{-k}$, and the mean value is $b = \frac{k}{l}$. Suppose the b_i, b_m, b_d are the mean disk service times of index lookup, metadata read, and data read respectively, we assume that the proportion of b_i, b_m, b_d remains in the context of fluctuating disk service time.

The distribution of request parsing latency. In order to obtain the raw latency of request parsing, we benchmark the cloud object storage system following two principles: avoiding accessing disks and avoiding requests queueing. To satisfy these two restrictions, we generate a close loop workload, with which all requests read the same data object during benchmarking. So, the data object could be served from memory due to being cached. We also limit the max amount of outstanding requests to 1 to avoid requests queueing. We record the following metrics for each request: D_{fp} (duration between a frontend tier process receives a request and starts responding) and D_{bp} (duration between a backend tier process receives a request and starts responding). The network latency of sending data from backend tier to frontend tier is $D_{net} = \frac{Data\ Size}{Network\ Bandwidth}$. For one request, its parsing latency at the backend tier is D_{bp}, and its parsing latency at frontend tier is $D_{fp} - D_{bp} - D_{net}$. Similarly, we use the distribution fitting to get the distribution of request parsing latency. In our testbed, the request parsing latency is almost constant (Degenerate distribution) at both the frontend tier and the backend tier.

4.2. System online metrics

Generally, the request arriving rates are available from the monitoring software of storage systems, and it is also easy to obtain the arriving rate of data read operations by counting data chunks. In terms of cache miss ratios, we use a latency threshold to distinguish a cache hit from a miss. Thanks to the huge speed gap between memory and disk, the approach of latency threshold could provide an accurate estimation of cache miss ratio. In our testbed, we use 0.015 ms as the latency threshold. As Linux only provides the summary value for disk service time, in order to obtain the average service times of different kind operations, we exploit the proportion of service times from Section 4.1. Suppose that r is the request arriving rate, and r_d is the arriving rates of data read operations. Suppose the overall service time is b, and b_i, b_m, b_d are the service times for index lookup, metadata read, and data read respectively. m_i, m_m, m_d are the corresponding cache miss ratios. p_i, p_m, p_d are the corresponding proportions. So we can obtain b_i, b_m, b_d by solving the following equations.

$$b_i/p_i = b_m/p_m = b_d/p_d$$
$$m_i b_i r + m_m b_m r + m_d b_d r_d = (m_i r + m_m r + m_d r_d) b$$

5. Applicability of the model

In this section, we discuss the applicability of our model. Timeout and retry are the essential exception handling mechanisms in cloud object storage systems. On one hand, timeouts could be triggered by some unexpected factors (e.g. hardware failures, network congestion, shared-resource contention, etc.), and the discussion of such factors is out of the scope of this paper. On the other hand, timeouts occur due to the heavy workloads. Since our model assumes that there is no timeout and retry, the prediction results would be inappropriate if the workload was beyond a threshold. Therefore, given the inputs, we should develop a method to determine whether or not our model is applicable. The key of the method is to predict the occurrence of timeouts. Then, we could use the following rule to determine the applicability of our model.

Our model is applicable when there is no timeout.

There are two kinds of timeouts in the cloud object storage system: the Connect Timeout and the Network Timeout. The connect timeout occurs when a frontend process does not receive the response of a connecting request within the specified time period. The network timeout happens when a frontend process does not receive the response of an HTTP request within the specified time period.

Suppose the maximum wait time for a connecting request and an HTTP request is T_{conn} and T_{net}, respectively. In the context of Poisson arrival, when T_{conn} and T_{net} are finite, the possibility (P) of timeout occurrence is always larger than 0 according to the queueing theory, as long as the request arriving rate is greater than 0. However, it does not mean that we can consider that timeouts always occur while predicting the occurrence of timeouts. In practice, we should consider that there is no timeout when the possibility of timeout occurrence is too low that can be ignored. Hence we use a threshold P_{thres} to determine whether there are timeouts with the following rule:

Timeouts occur when $P \geq P_{thres}$, and No timeout when $P < P_{thres}$.

We first consider the scenario that T_{conn} and T_{net} are both infinite. In this scenario, we could predict the percentage (P_{inf}) of requests not being responded within the wait time bounds, exploiting the techniques that are used to develop our model in Section 3. Suppose P_{conn} is the percentage of connecting requests that do not response in T_{conn}, P_{net} is the percentage of HTTP requests that do not response in T_{net}, and then there is

$$P_{inf} = P_{conn} + P_{net}.$$

The requests, of which the frontend processes do not receive the corresponding response in time, result in timeouts when there is a finite limitation on the waiting time for a request. Nevertheless, P_{inf} is not the exact possibility of timeout occurrence (P) due to the impacts of timeouts and retries on the system performance. Because either P_{inf} or P increases monotonically relative to the workload, there is a positive correlation between P_{inf} and P. Moreover, we could use P_{inf} to approximate P when P is small. It is because the small P suggests that there are only a small number of timeouts, which can only have a limited impact on the response latency.

In theory, the range of P_{thres} is $0 \leq P_{thres} \leq 1$. However, The P_{thres} should be small to effectively distinguish the case of timeouts existing from the case of no timeout. The impact of timeouts and retries should be negligible for any $P \leq P_{thres}$. Hence, when $P = P_{thres}$, we have $P = P_{inf}$. Due to the positive correlation between P_{inf} and P, we could also obtain following conclusions: (1) when $P > P_{thres}$, there is $P_{inf} > P_{thres}$; (2) when $P < P_{thres}$, there is $P_{inf} < P_{thres}$. Therefore, we could use P_{inf} to replace P for predicting the occurrence of timeouts. Then we have the following rule:

Timeouts occur when $P_{inf} \geq P_{thres}$, and No timeout when $P_{inf} < P_{thres}$.

The methods for calculating P_{conn} and P_{net} are as follows. Suppose the set of the storage devices is D. Given a storage device D_j, $D_j \in D$, the request arriving rate for D_j is r_j, the cdf. of response waiting time for connecting requests is W^j_{conn}, and the cdf. of response waiting time for HTTP requests is W^j_{net}. As the response waiting time for a connecting request is approximate to the waiting time for being accept()-ed of the connecting request, $W^j_{conn}(t) = \int_0^t [W^j_a(x)]dx$, where $W^j_a(x)$ is the pdf. of waiting time for being accept()-ed of D_j (we can get $W^j_a(x)$ according to Section 3.3). As the response waiting time for a HTTP request is approximate to the response latency at backend tier, $W^j_{net} = \int_0^t [S^j_{be}(x)]dx$, where $S^j_{be}(x)$ is the pdf. of the response latency at backend tier for D_j (we can get $S^j_{be}(x)$ according to Section 3.2). Hence, P_{conn}, P_{net} are:

$$P_{conn} = 1 - \frac{\sum_{D_j \in D}[r_j W^j_{conn}(T_{conn})]}{\sum_{D_j \in D} r_j}$$

$$P_{net} = 1 - \frac{\sum_{D_j \in D}[r_j W^j_{net}(T_{net})]}{\sum_{D_j \in D} r_j}.$$

6. Evaluation

In this section, we present our experimental setup and evaluate our model with following goals.

(1) Evaluate the accuracy of our model for diverse SLAs, workloads, and system configurations.

(2) Evaluate the contributions of the core components (the abstraction of union operation, the model of waiting time for being accept()-ed) to the overall accuracy of our model.

(3) Evaluate the accuracy of the method for predicting the applicability of our model.

6.1. Experimental setup

Our testbed is a 7-nodes OpenStack Swift cluster, including 3 frontend servers and 4 backend servers, and we use 1 Gbps Ethernet to connect the frontend and backend servers. There is a 1 TB HDD disk attached to each backend server. Data objects are mapped to 1024 partitions based on hashing, and each partition has 3 replicas. OpenStack Swift evenly distributes all replicas among the 4 disks (the replicas of the same partition are placed on different disks). There are 7 extra nodes serving as workload generators. The workload generators and frontend servers are connected via 40 Gbps Infiniband. Such configuration prevents workload generators from being the bottleneck of the whole system. Each node has four 2.4 GHz Intel E5620 quad-core CPUs, 24 GB of memory, and runs Centos 7. Except that we limit the memory of the backend servers to 5 GB, and we perform such limitation to imitate the production environments of the cloud object storage system, in which backend servers do not have sufficient memory space for serving as a cache (e.g. in the OpenStack Swift cluster of Wikipedia, the RAM-to-disk ratios of the backend server range from about 1:300 to 1:800 [27]).

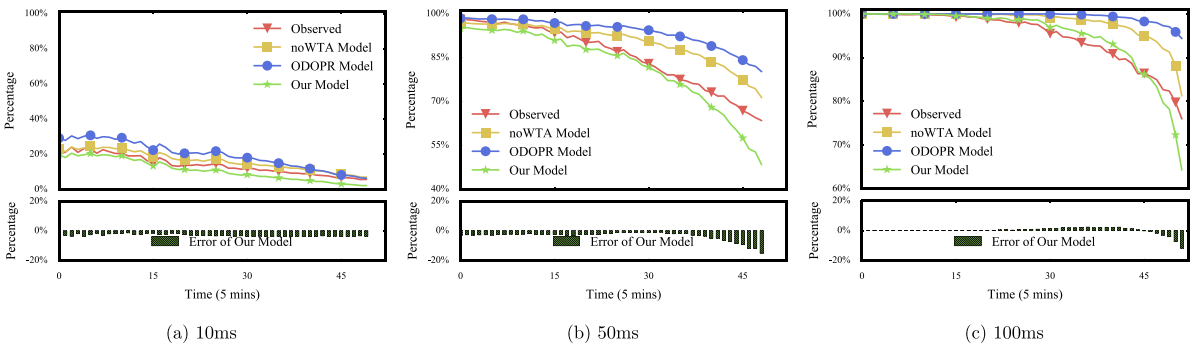
We generate the workload based on a 50 h trace of media objects accessing from Wikipedia. This trace is extracted from the trace provided along with wikibench [30] (the URL of media object contains "upload.wikimedia.org"). However, the trace does not provide any information on object size. We determine the size of each media object by directly requesting the object from Wikipedia. And about 45% of the objects no longer exist in Wikipedia, and so the requests for these objects are overlooked in our workload. The average size of remaining objects is about 32 kB. The average size of requests is about 10 kB.

We use the SwiftStack Benchmark Suite (ssbench) [28] as our workload generator. Ssbench contains multiple workers (as OpenStack Swift clients, performing requests) and one master (generating requests and distributing them among the workers). Load balancing is a built-in feature of ssbench, which sends each request to a random frontend server, so we do not use dedicated load balancers in our system. We modify ssbench to support replaying trace and issuing requests in an open loop manner. We measure the requests latency at frontend servers instead of ssbench workers. Because, in practice, the latency introduced by clients is out of the control from the perspective of a cloud object storage system (the client could be any laptop behind Internet), and our model focuses on the response latency of the cloud object storage system. We control the rate of generating requests at the ssbench master.

6.2. The accuracy of the model

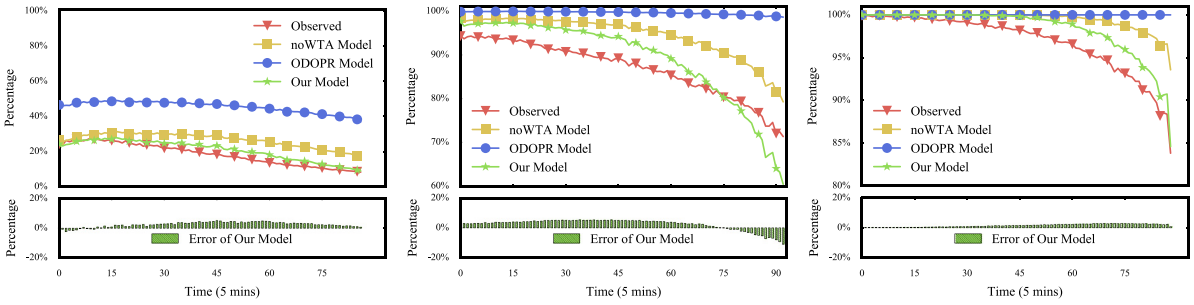
We conduct a set of experiments to evaluate the accuracy of our model on predicting the percentage of requests meeting a response latency requirement. We perform the evaluation for two scenarios, *S1* and *S16*. At the backend tier, we use the configuration of 1/16 process(es) per storage device for the scenario *S1*/*S16* respectively. For each scenario, we carry out the experiments with 3 different SLAs (response latency requirements of 10 ms, 50 ms, and 100 ms). We conduct separate experiments to validate the model for different SLAs in the same scenario. In each experiment, the system counts the number of requests that meet or violate the SLA for each storage device at frontend tiers for each minute. We predict the percentage of requests meeting an SLA using the average values of system metrics in every 5 min of the same arriving rate of requests. As discussed in Section 3.1, we only analyze the prediction results when there is no timeout and retry. We do not perform a direct comparison with existing models due to the following reasons. (1) Existing models [4,11,14,31,38] predict the average performance metrics rather than the percentage of request meeting an SLA; (2) Existing models [35,39] rely on simulation-based technique for prediction; (3) Existing models [15,32,38] focus on modeling different factors (e.g. data striping) of the system instead of the factors addressed by our model.

We generate workloads by changing the request arriving rate of the trace described in Section 6.1. In order to control the arriving rate of requests, we change the timestamp field of each request in the trace. With our modification, a workload contains 3 phases: *warmup phase* lasts 3 h with a fixed arriving rate, *transition phase* lasts 1 h with a fixed arriving rate, and *benchmarking phase* with a varying arriving rate and each arriving rate lasts 5 min. The *transition phase* exists to make sure that all of the requests from the *warmup phase* finish before the *benchmarking phase*. The arrival of requests follows Poisson process. We generate such synthetic workloads so that we could experiment with a broader range of arriving rates, which is not limited by the actual arriving rates of the trace. The workloads for scenario *S1* and *S16* are different due to different system configurations. During warmup phase, the arriving rate is 300 requests per second for scenario *S1* and 500 for scenario *S16*. The arriving rate is 10 requests per second for all scenarios during the transition phase. During the benchmarking



(a) 10ms (b) 50ms (c) 100ms

Fig. 7. Prediction results of different SLAs for scenario *S1* (1 process per storage device). The *x*-axis shows the execution time.



(a) 10ms (b) 50ms (c) 100ms

Fig. 8. Prediction results of different SLAs for scenario *S16* (16 processes per storage device). The *x*-axis shows the execution time.

phase, the arriving rate starts at 10 requests per second and ends at 350 for *S1* (600 for *S16*), with the increase of 5.

Figs. 7 and 8 show the observed percentages of requests meeting SLAs (10 ms, 50 ms, 100 ms), and the predicted percentages of our model for the scenario *S1* and *S16*, respectively. And Figs. 7 and 8 also display the prediction errors of our model (the difference between predicted and observed percentages) at the bottom of each graph. In Figs. 7 and 8, the *x*-axis depicts the execution time of the corresponding experiment, and the execution time actually corresponds the arriving rate during the benchmarking phase. The number of points is different in the graphs of Fig. 7, so are the graphs in Fig. 8. The reason is that timeouts do not occur at the exact identical time point in different experiments (randomness exists in the replica choosing scheme of OpenStack Swift, etc.). For all figures in Section 6, we use identical scales of *y*-axis for prediction errors to enable comparability.

For the scenario *S1* (Fig. 7), our model underestimates the percentage of requests meeting an SLA in general. The underestimation is due to the assumption that the service time of an operation is independent. As a matter of fact, a correlation exists because operations of a request are more likely to have similar cache behaviors. For example, if an index lookup operation is a cache-hit operation, the corresponding metadata read operation would be more likely a cache-hit operation. Hence, our model underestimates the probability that a majority of operations in a request are cache hit operations. In another word, our model underestimates the percentage of fast-responding requests. Similarly, our model also underestimates the percentage of long-processing requests. On one hand, the underestimation of fast-responding requests leads to the underestimation of requests meeting an SLA when the latency requirement is relatively low. On the other hand, the underestimation of long-processing requests results in the overestimation of requests meeting a relatively high latency requirement. Whether a latency requirement being low or high is not absolute, it depends on the distribution of the response latency. Consider the SLA of 100 ms, in the low load region, 100 ms is a high latency requirement as the response latency is relatively low in general. However, in the high load region, 100 ms is a low latency requirement due to the overall high response latency.

Furthermore, we observe that the underestimation is more serious in high load region. The reasons are as follows: (1) the greater overestimation of Waiting Time for being Accept()-ed (WTA) at higher loads due to the longer request processing queue (detailed in Section 3.3); (2) the greater overestimation of waiting time in Request Processing Queue (RPQ) at higher workload. It takes longer for the system to reach the steady state at higher workload due to the exponentially increased expected length of RPQ, and our model overestimates the length of RPQ for a growing workload due to assuming a steady state system. So, there is a greater overestimation of the length of RPQ at higher loads, and the length of RPQ is positively correlated with the waiting time in RPQ.

Different from the scenario *S1* (Fig. 7), our model generally overestimates the percentage of requests meeting an SLA for the scenario *S16* (Fig. 8). There are following reasons: (1) our model uses 0 to approximate the service time of the Cache Hit Union operation (detailed in Section 3.2). Such approximation results in an underestimation of response latency due to ignoring the service time of request parsing. (2) our model assumes that requests are distributed among backend processes corresponding to the storage device with the equivalent probability. Such assumption is true in the long term. However, load imbalance occurs in short periods of time. It is because each process may batch accept requests or be blocked by long processing requests. Our model underestimates the response latency as the load imbalance will lead to the increased response latency. As a result, our model overestimates the percentage of requests meeting an SLA due to the underestimation of the response latency.

Table 1 summarizes the prediction errors (absolute value) of our model for the different scenarios and SLAs. For all cases, the average

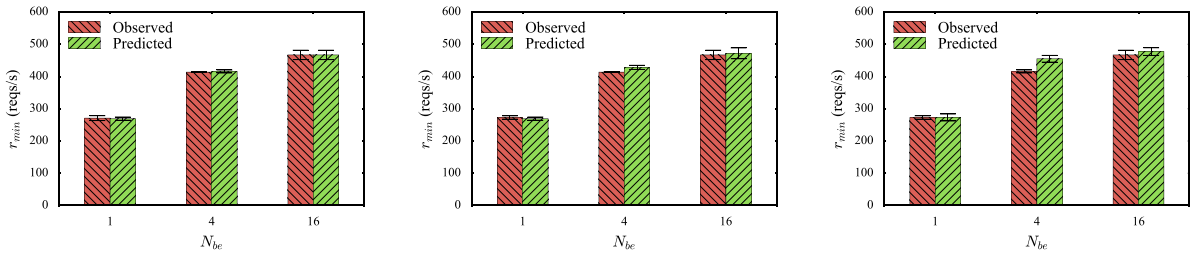


Fig. 9. The observed and predicted request arriving rate (r_{min}) that divides the applicable cases of our model from inapplicable cases when P_{thres} = 0.1%, 0.5%, and 1.0%.

Table 1
The summary of prediction errors for our model.

Scenario	SLA	Best case	Worst case	Mean
S1	10 ms	1.01%	3.82%	2.91%
	50 ms	0.86%	15.04%	3.47%
	100 ms	0.02%	11.70%	1.26%
S16	10 ms	0.16%	5.03%	2.64%
	50 ms	0.00%	10.90%	4.00%
	100 ms	0.07%	2.96%	1.50%

Table 2
The mean prediction errors of different models.

Scenario	SLA	Our model	ODOPR model	noWTA model
S1	10 ms	2.91%	6.54%	2.45%
	50 ms	3.47%	9.41%	5.18%
	100 ms	1.26%	4.80%	3.26%
S16	10 ms	2.64%	27.06%	8.19%
	50 ms	4.00%	12.78%	7.60%
	100 ms	1.50%	3.13%	2.61%

prediction error of our model is 2.63% and the worst case error is 15.04%.

6.3. The contribution of core components

We reveal the contributions of the core components of our model to the overall accuracy by comparing our model with two baseline models: ODOPR model and noWTA model. First, the ODOPR model considers cache hit for all index lookup, metadata read, and extra data read. The ODOPR model imitates the existing models assuming no more than One Disk Operation Per Request (ODOPR) at storage servers. Second, the noWTA model considers that there is no Waiting Time for being Accept()-ed (noWTA). The noWTA model imitates the existing models not taking the WTA into consideration. Table 2 shows the average prediction errors (absolute value) of the ODOPR model, the noWTA model and our model in the context of devise scenarios and SLAs.

The contribution of the abstraction of union operation: Our model relies on the abstraction of union operation for modeling diverse disk operations (index lookup, metadata read, data read) and data chunking of event-driven programming model. Compared with the ODOPR model, which does not consider these factors, our model reduces the average prediction errors by 52% to 90% (relative percentage) for different scenarios and SLAs.

The contribution of modeling waiting time for being accept()-ed: Compared with the noWTA model, which does not consider the WTA, our model reduces the average prediction errors by 33% to 68% (relative percentage) for different scenarios and SLAs, except the 10 ms SLA in the scenario *S1*. As a matter of fact, our model increases the average prediction errors by 19% (relative percentage) for the 10 ms SLA in the scenario *S1*. It is because the overestimation of the WTA introduces more errors than overlooking the WTA. The 10 ms SLA is an extreme case, and less than 25% of requests respond within 10 ms even under the lightest workload.

6.4. The accuracy of prediction on model applicability

We conduct a set of experiments to evaluate the accuracy of the method on predicting the applicability of our model. In the experiments, we use the default timeout configurations of OpenStack Swift. To be specific, the maximum wait time for a connecting request is 0.5 s and the maximum wait time for an HTTP request is 10 s. We evaluate the method for 3 different scenarios: the scenario of $N_{be} = 1$, $N_{be} = 4$, and $N_{be} = 16$, where the N_{be} is the number of backend processes per storage device. For each scenario, we generate the workload as same as the workload of scenario *S16* from Section 6.2 and record the number of timeouts for each time period of same request arriving rate. And we run each experiment for 3 times.

We determine whether or not our model is applicable according to the rules in Section 5. Suppose n_r is the number of total timeouts when the request arriving rate is r reqs/s (each request arriving rate lasts 5 min), the observed percentage of timeouts over requests is $P_{observed} = \frac{n_r}{300r}$. Then, *Our model is applicable when* $P_{observed} \geq P_{thres}$, *and inapplicable when* $P_{observed} < P_{thres}$, where P_{thres} is the threshold distinguishing the case that there are timeouts from the case of no timeout.

Considered the situation that timeouts occur when the request arriving rate is r_{occur}, we can infer that there are timeouts when the request arriving rate (r) is higher than r_{occur}. The reason is that the queueing latency grows along with the request arriving rate while other conditions (including cache miss ratios, load distributions, etc.) remain. Suppose r_{min} is the minimum request arriving rate that leads to the occurrence of timeouts. Then, we could use r_{min} to divide applicable cases of our model from inapplicable cases (our model is applicable when $r < r_{min}$). Hence, we measure the accuracy of the prediction on model applicability by comparing the observed and predicted r_{min}.

Fig. 9 shows the observed and predicted results on r_{min} with diverse scenarios and SLAs when $P_{thres} = 0.1\%, 0.5\%$, and 1.0%. Fig. 9 reports both the mean and variance of r_{min}. On average, the relative error of the prediction is about 0.4% when P_{thres} = 0.1%, about 2.0% when $P_{thres} = 0.5\%$, and about 3.9% when $P_{thres} = 1.0\%$. The average prediction error grows along with the P_{thres} due to the decreased accuracy of our model with more timeouts.

7. Related work

7.1. Queueing network

General queue networks fail to model the cloud object storage systems due to assuming that successive response times of the

queues in a path through the network are independent, however, disk operations block the request processing queue at the backend tier of the cloud object storage system. As a matter of fact, the cloud object storage system could be modeled by Layered Queueing Network (LQN), which is an extension to queueing networks. In LQN, the service time of upper layer queue is given by the response time of a lower layer queue. However, there is no LQN solver that calculates the distribution of the response latency for the LQN using FCFS queueing discipline. The state-of-the-art LQN solvers, including LQNS [8] and DiffLQN [34], focus on calculating the mean values like throughput and average response time, and Line [25] calculates the distribution of the response latency for LQN using PS discipline. Some LQN solvers support estimating response latency distribution with simulation, which is high time consumption.

7.2. Multi-tiered application performance modeling

Modeling multi-tiered web application has been studied extensively. The initial attempts, including Villela et al. [33] and Ranjan et al. [26], use bottleneck tier to build models. Yaksha [14] models an entire e-commerce application as M/GI/1/PS queue. These models generally assume that the applications are computation intensive, which makes them fail to catch the performance characteristics of I/O intensive cloud object storage systems.

On the one hand, different from our model that predicts the percentage of requests meeting a response latency requirement, the recent models generally predict the average performance metrics (e.g. throughput, average response latency) for a particular scenario. For instance, Liu et al. [16] and Urgaonkar et al. [31] use closed queueing networks to model session-based web applications, Calheiros et al. [4] and Jung et al. [13] rely on queueing networks for modeling applications running in virtualized environments, Ghaith et al. [9] consider the software contention while modeling 3-tiered web applications using LQN, and Han et al. [11] build a performance model for latency-critical applications in the context of sharing resources with offline batch jobs. On the other hand, Nguyen et al. [20] use the mean value and variance of latencies to predict tail latency in the high load region. However, we use important metrics, say workload and cache miss ratio, to predict the percentage of requests meeting a response latency requirement.

7.3. Storage system performance modeling

The focuses of modeling different types of storage systems are different. Wu et al. [38] propose a general guideline of constructing LQN for modeling the interaction of different components in the distributed file system (e.g. HDFS). For parallel storage systems (e.g. Lustre, PVFS) and RAID (redundant array of independent disks), performance models [15,32] generally exploit fork-join queue for modeling data striping, where an IO request is split into several sub-requests of different storage devices. These models fail to deal with the cloud object storage system because they assume no more than one disk access per request at storage servers and overlook the waiting time of being accept()-ed, and our model addresses these issues. Yanggratoke et al. [39] build a performance model to predict the distribution of the response latency for the Spotify backend (the Spotify backend works as a cache tier of Amazon S3), and the difference between their model and our model is that their model introduces a workload-dependent parameter q, which is the probability that an arriving request can be served from memory and not blocked by requests served from disks. They use benchmarking to obtain the relationship between requests arriving rate and q for each individual backend server. Although our model needs benchmarking to get device performance properties (detailed in Section 4), the crucial difference is that the device performance properties are independent of the workload.

8. Conclusion

In this paper, we present an analytic-based performance model that predicts the percentage of requests meeting an SLA for the cloud object storage system using event-driven programming model. Moreover, we also provide a method that predicts the applicability of our model. Our model addresses the complexity of diverse disk operations being scheduled in an interleaving manner at storage servers. Our model also quantifies the impact of the waiting time for being accept()-ed on the response latency of the system. We comprehensively consider different types of timeouts that invalidate our model and determine the applicability of our model by predicting the occurrence of timeouts. We validate our model by replaying a real-world trace against an OpenStack Swift cluster. Our experiments demonstrated that our model faithfully captures the performance of the cloud object storage system, and the prediction results on the applicability of our model are also accurate. Moreover, the implementation is available as open-source software from https://github.com/ysu-hust/cosmodel.

Acknowledgments

This work was supported by the National High Technology Research and Development Program (863 Program), China No. 2015AA015301, the National Key Research and Development Program of China under Grant 2016YFB1000202; NSFC, China No. 61472153, No. 61502191; State Key Laboratory of Computer Architecture, China, No. CARCH201505; This work was also supported by Engineering Research Center of data storage systems and Technology, Ministry of Education, China.

References

[1] Amazon s3, https://aws.amazon.com/s3/, 2016-10-19.
[2] D. Beaver, S. Kumar, H.C. Li, J. Sobel, P. Vajgel, Finding a needle in haystack: Facebook's photo storage, in: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, in: OSDI'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 1–8.
[3] T. Brecht, D. Pariag, L. Gammo, Acceptable strategies for improving web server performance, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, in: ATEC '04, USENIX Association, Berkeley, CA, USA, 2004, pp. 20–20.
[4] R.N. Calheiros, R. Ranjan, R. Buyya, Virtual machine provisioning based on analytical performance and qos in cloud computing environments, in: 2011 International Conference on Parallel Processing, 2011, pp. 295–304, http://dx.doi.org/10.1109/ICPP.2011.17.
[5] Ceph use cases, http://ceph.com/users/, 2016-10-19.
[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, in: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, in: SOSP '07, ACM, New York, NY, USA, 2007, pp. 205–220, http://dx.doi.org/10.1145/1294261.1294281.
[7] Q. Fan, Q. Wang, Performance comparison of web servers with different architectures: A case study using high concurrency workload, in: 2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), 2015, pp. 37–42, http://dx.doi.org/10.1109/HotWeb.2015.11.
[8] G. Franks, T. Al-Omari, M. Woodside, O. Das, S. Derisavi, Enhanced modeling and solution of layered queueing networks, IEEE Trans. Softw. Eng. 35 (2) (2009) 148–161, http://dx.doi.org/10.1109/TSE.2008.74.
[9] S. Ghaith, M. Wang, P. Perry, L. Murphy, Software contention aware queueing network model of three-tier web systems, in: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, in: ICPE '14, ACM, New York, NY, USA, 2014, pp. 273–276, http://dx.doi.org/10.1145/2568088.2576760.
[10] M. Hall, Combinatorial Theory, John Wiley & Sons, 1998.
[11] R. Han, J. Wang, S. Huang, C. Shao, S. Zhan, J. Zhan, J.L. Vazquez-Poletti, Pcs: Predictive component-level scheduling for reducing tail latency in cloud online services, in: 2015 44th International Conference one Parallel Processing, 2015, pp. 490–499, http://dx.doi.org/10.1109/ICPP.2015.58.
[12] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, H.C. Li, An analysis of facebook photo caching, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, in: SOSP '13, ACM, New York, NY, USA, 2013, pp. 167–181, http://dx.doi.org/10.1145/2517349.2522722.

[13] G. Jung, K.R. Joshi, M.A. Hiltunen, R.D. Schlichting, C. Pu, Generating adaptation policies for multi-tier applications in consolidated server environments, in: International Conference on Autonomic Computing, 2008. ICAC '08, 2008, pp. 23–32, http://dx.doi.org/10.1109/ICAC.2008.21.
[14] A. Kamra, V. Misra, E.M. Nahum, Yaksha: A self-tuning controller for managing the performance of 3-tiered Web sites, in: Twelfth IEEE International Workshop on Quality of Service, 2004. IWQOS 2004, 2004, pp. 47–56, http://dx.doi.org/10.1109/IWQOS.2004.1309356.
[15] A.S. Lebrecht, N.J. Dingle, W.J. Knottenbelt, Analytical and simulation modelling of zoned RAID systems, Comput. J. (2010) bxq053, http://dx.doi.org/10.1093/comjnl/bxq053.
[16] X. Liu, J. Heo, L. Sha, Modeling 3-tiered Web applications, in: 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005, pp. 307–310, http://dx.doi.org/10.1109/MASCOTS.2005.40.
[17] D. Meisner, C.M. Sadler, L.A. Barroso, W.-D. Weber, T.F. Wenisch, Power management of online data-intensive services, in: Proceedings of the 38th Annual International Symposium on Computer Architecture, in: ISCA '11, ACM, New York, NY, USA, 2011, pp. 319–330, http://dx.doi.org/10.1145/2000064.2000103.
[18] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, S. Kumar, F4: Facebook's warm BLOB storage system, in: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), USENIX Association, Broomfield, CO, 2014, pp. 383–398.
[19] D.S. Myers, M.K. Vernon, Estimating queue length distributions for queues with random arrivals, SIGMETRICS Perform. Eval. Rev. 40 (3) (2012) 77–79, http://dx.doi.org/10.1145/2425248.2425268.
[20] M. Nguyen, Z. Li, F. Duan, H. Che, H. Jiang, The tail at scale: how to predict it? in: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), USENIX Association, Denver, CO, 2016.
[21] S.A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, R.H. Campbell, Ambry: LinkedIn's scalable geo-distributed object store, in: Proceedings of the 2016 International Conference on Management of Data, in: SIGMOD '16, ACM, New York, NY, USA, 2016, pp. 253–265, http://dx.doi.org/10.1145/2882903.2903738.
[22] Openstack swift, https://docs.openstack.org/developer/swift/, 2016-10-19.
[23] Openstack swift and many small files, http://engineering.spilgames.com/openstack-swift-lots-small-files/, 2016-10-19.
[24] Openstack swift use cases, https://www.swiftstack.com/customers, 2016-10-19.
[25] J.F. Pérez, G. Casale, Assessing SLA compliance from palladio component models, in: 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013, pp. 409–416, http://dx.doi.org/10.1109/SYNASC.2013.60.
[26] S. Ranjan, J. Rolia, H. Fu, E. Knightly, QoS-driven server migration for Internet data centers, in: Tenth IEEE International Workshop on Quality of Service, 2002, 2002, pp. 3–12, http://dx.doi.org/10.1109/IWQoS.2002.1006569.
[27] Swift eqiad cluster report, https://ganglia.wikimedia.org/latest/?r=week&cs=&ce=&c=Swift+eqiad, 2016-10-19.
[28] Swiftstack benchmark suite (ssbench), https://github.com/swiftstack/ssbench, 2016-10-19.
[29] J. Sztrik, Basic Queueing Theory, Vol. 193, University of Debrecen, Faculty of Informatics, 2012.
[30] G. Urdaneta, G. Pierre, M. van Steen, Wikipedia workload analysis for decentralized hosting, Elsevier Comput. Netw. 53 (11) (2009) 1830–1845.
[31] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, A. Tantawi, An analytical model for multi-tier internet services and its applications, in: Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, in: SIGMETRICS '05, ACM, New York, NY, USA, 2005, pp. 291–302, http://dx.doi.org/10.1145/1064212.1064252.
[32] E. Varki, Response time analysis of parallel computer and storage systems, IEEE Trans. Parallel Distrib. Syst. 12 (11) (2001) 1146–1161, http://dx.doi.org/10.1109/71.969125.
[33] D. Villela, P. Pradhan, D. Rubenstein, Provisioning servers in the application tier for e-commerce systems, ACM Trans. Internet Technol. 7 (1) (2007) http://dx.doi.org/10.1145/1189740.1189747.
[34] T. Waizmann, M. Tribastone, DiffLQN: Differential equation analysis of layered queuing networks, in: Companion Publication for ACM/SPEC on International Conference on Performance Engineering, in: ICPE '16 Companion, ACM, New York, NY, USA, 2016, pp. 63–68, http://dx.doi.org/10.1145/2859889.2859896.
[35] B.J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, Z. Wang, Probabilistic performance modeling of virtualized resource allocation, in: Proceedings of the 7th International Conference on Autonomic Computing, in: ICAC '10, ACM, New York, NY, USA, 2010, pp. 99–108, http://dx.doi.org/10.1145/1809049.1809067.
[36] Wikipedia openstack swift cluster live status, https://grafana.wikimedia.org/dashboard/file/swift.json?var-DC=eqiad, 2016-10-19.
[37] R.W. Wolff, Poisson arrivals see time averages, Oper. Res. 30 (2) (1982) 223–231.
[38] Y. Wu, F. Ye, K. Chen, W. Zheng, Modeling of distributed file systems for practical performance analysis, IEEE Trans. Parallel Distrib. Syst. 25 (1) (2014) 156–166, http://dx.doi.org/10.1109/TPDS.2013.19.
[39] R. Yanggratoke, G. Kreitz, M. Goldmann, R. Stadler, Predicting response times for the Spotify backend, in: 2012 8th International Conference on Network and Service Management (Cnsm) and 2012 Workshop on Systems Virtualization Management (Svm), 2012, pp. 117–125.

Yi Su received the B.S. degree in Computer Science from the Huazhong University of Science and Technology (HUST), China, in 2012. He is currently a Ph.D. candidate of the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. His research interests include cloud storage systems, big data processing systems.

Dan Feng received the BE, ME, and Ph.D. degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and the dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEETC, IEEETPDS, ACM-TOS, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She has served as the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012, 2015. She is a member of IEEE and a member of ACM.

Yu Hua received the BE and Ph.D. degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He is a full professor at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing, and network storage. He has more than 100 papers to his credit in major journals and international conferences including IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS), USENIX ATC, USENIX FAST, INFOCOM, SC and ICDCS. He has been on the program committees of multiple international conferences, including USENIX ATC, RTSS, INFOCOM, ICDCS, MSST, ICNP and IPDPS. He is a senior member of the IEEE, ACM and CCF, and a member of USENIX.

Zhan Shi received his B.S. degree and Master degree in Computer Science, and Ph.D. degree in Computer Engineering from Huazhong University of Science and Technology (HUST, China). He is working at the Huazhong University of Science and Technology (HUST) in China, and is an Associate Researcher in Wuhan National Laboratory for Optoelectronics. His research interests include storage management, distributed storage system and cloud storage.