Predicting Response Latency Percentiles for 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, Huazhong University of Science and Technology, Wuhan, China
Email: {suyi,dfeng,csyhua,zshi}@hust.edu.cn

***Abstract*—As a fundamental cloud service for modern Web applications, 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. Due to the lack of suitable understanding on the response latency distribution, current practice is to use overprovision resources to meet Service Level Agreement (SLA). Hence we build a performance model for the cloud object storage system to predict the percentiles of requests meeting SLA (response latency requirement), in the context of complicated disk operations and event-driven programming model. Furthermore, we find that the waiting time for being accept()-ed at storage servers may introduce significant delay. And we quantify the impacts on system response latency, due to requests waiting for being accept()-ed. In a variety of scenarios, our model reduces the prediction errors by up to 73% compared to baseline models, and the prediction error of our model is 4.44% on average.**

***Keywords*-performance modeling; cloud storage; response latency;**

I. Introduction

The cloud object storage system, like Amazon S3 [1], OpenStack Swift [2], plays an important role in modern web-based applications and generally stores and retrieves millions or even billions of diverse data objects (also called blobs), including photos, audios, videos, documents, etc.

We build a queueing-theory based performance model for the cloud object storage system, which uses event-driven concurrency architecture (e.g. one process handles multiple transactions in time-division multiplexing manner with an event loop using epoll/poll/select function). The event-driven architecture has been widely adopted by cloud object storage systems (detailed in Section II). Different from existing analytic-based models [3]–[6] that predict the average performance metrics (e.g. throughput, mean response latency), our model predicts the percentile of requests meeting SLA (requirement of response latency), e.g. 95% of the requests could 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 [7].

Considered the large volume of data objects and the long tail distribution of data accessing [8], [9], the 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 workload anticipation and 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 [10] magnifies the necessity of cluster expansion. In addition to capacity planning, a simple yet accurate performance model of the cloud object storage system 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. And the system should meet the performance requirements at the same time.

Modeling of multi-tiered Web applications [3]–[6] and storage systems [11]–[13] 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 locating the data object on the storage device, e.g. inodes of the local file system. And the metadata is the attributes of the data object, e.g. checksum, createtime, 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 [8] (detailed in

* Corresponding author: Dan Feng (dfeng@hust.edu.cn).

Section II). 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. Models targeting other usage scenarios fail to deal with this complexity. 2). **Data Chunking**. The event-driven architecture 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 event-driven architecture, instead of reading and transmitting all of the data 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 the different requests in an interleaving manner (detailed in Section III-B). 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. [14] 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 analytics on the waiting time for being accept()-ed.

The focus of this paper is the design of an analytic-based model that can capture the impact of the factors mentioned above. It is worth to mention that our model requires some benchmarking based parameters (detailed in Section IV). However, the benchmarking in our model is independent to workloads, which makes our model differ from simulation-based models. Workloads is always a key factor for benchmarking in the simulation-based models, which makes the simulation-based models vulnerable to the changes of workloads.

We evaluate our model with an OpenStack Swift testbed by replaying a real-world trace (accessing trace of media objects from Wikipedia [15]). In a variety of scenarios, the prediction error of our model is 4.44% on average.

In summary, our contributions include:

1). **The Abstraction of Union Operation**: We build a queueing-theory 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 event-driven architecture, 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 the quantitative analytics by revealing the relationship between the waiting time for being accept()-ed and the status of request processing queues.

3). **Prototype Implementation and Evaluation**: We implement all components of our model based on OpenStack Swift and evaluate the accuracy of our model using real-world trace in various scenarios. And our implementation is open-source and available from https://github.com/ysu-hust/cosmodel.

II. Background

The cloud object storage system is a two-tiered web application, as shown in Fig. 3. 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. And each storage device has dedicated process(es) for handling its corresponding requests at the backend server. The process of a storage device perform the following operations in sequence for handling a request, 1). locating the data object on the storage device (index lookup); 2). get the metadata (metadata read), 3). read 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). open the file (index lookup), 2). read the extended attributes of the file (metadata read), 3). read the file by a chunk (data read).

In the context of saving the TCO (Total Cost of Ownership), the cloud providers prefer the cheap storage device of large capacity (e.g. HDD) rather than the expensive storage device of high performance (e.g. SSD) because the cloud object storage system needs capacity more than performance. For example, in the OpenStack Swift cluster of Wikipedia [16], the total consumed storage capacity is about 670TB and the *maximum* aggregated throughput of the backend tier last year is only about 1.2 GB/s. And the response latency of HDD disk is in milliseconds, which is sufficient for cloud object storage systems. So, production systems [8], [10], [17] commonly store data in cheap HDD disks, instead of much more expensive SSD disks (a 2TB low-end SSD drive costs about \$500, 10 times the 2TB 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 the backend servers [8]. And the majority of data objects are of small size [8], [10], [18], [19] in production environments. Suppose that the average size of data objects is about

50KB [10], [18], and the average size of I&M of data objects is about 1KB altogether [18]. There are about 20GB I&M for each 1TB data in this scenario. And it is not cost-effective to cache all these I&M in memory. As a result, while processing a request at the backend tier, all of the operations with diverse performance characteristics, e.g. index lookup, metadata read, and data read, may access HDD disks due to the long tail distribution of data accessing [8], [9]. These specificities of the cloud object storage system introduce new challenges for modeling. The methods of reducing the size of the indexes have been proposed [8], [10] 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 [17], [18], [20] 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 system addressed by our model.

The thread-per-connection and the event-driven architectures are the main strategies for handling concurrency for cloud object storage systems. And this paper concentrates on modeling behaviors of event-driven architecture because the event-driven architecture is widely adopted in many famous cloud object storage systems (like OpenStack Swift, Ceph) and production environments [20], [21]. Moreover, the event-driven architecture is superior to the thread-per-connection architecture in both throughput and tail response latency [22].

III. 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 for the model.

A. Assumptions for Modeling

We build the model under the following assumptions.

1). **Poisson arrival**. The 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 [23]. 2). **Read heavy workloads**. The model does not consider WRITE and DELETE requests. The workloads for cloud object storage systems are read dominant [8], [10], [17], 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 [17], $> 95\%$ in LinkedIn Ambry [10], and $> 98\%$ in Facebook Haystack [8], etc. 3). **Sufficient resources of computation and network**. Resources of network and computation are commonly sufficient in cloud object storage systems. Take the OpenStack Swift cluster of Wikipedia [17] 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 2.4GHz CPU core could perform 25 Million instructions per second and the 1Gbps 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 hours for the request arriving rate to increase from 700 requests per second to 1500 requests per second in the OpenStack Swift cluster of Wikipedia [17]. 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.

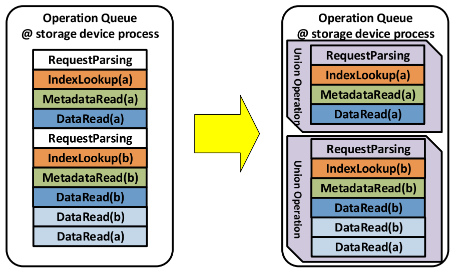


Figure 1. Queue of a storage device (single process per storage device)

B. 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 for 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 the 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 represent the series of different kinds of operations appearing in the system, we pack the diverse operations of request parsing, index lookup, metadata read, and data read, into one union operation. And each union operation may contain operations of different requests. As a result, the original operation queue is transformed into the queue of union operations, which is shown as the right side queue in Fig. 1. With the assumption of Poisson arrival (detailed in Section III-A), we model the queue of union operations as an M/G/1 queue (a queueing system of Poisson arrivals, generally distributed service times, 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. And these metrics are easy to be measured or calculated. With the caching mechanism, index lookup, metadata read, and data read can be served either from memory or disk. And the $m_{index}, m_{meta}, m_{data}$ 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 latencies 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 IV). The latency of memory is negligible, and we therefore approximate it with 0. Let $parse_{be}(t), index(t), meta(t), data(t)$ represent the pdf. of latencies 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, $\delta(t)$ 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 [23]. In other words, we could assume that the arrival of extra data read follows Possion 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 times for 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.

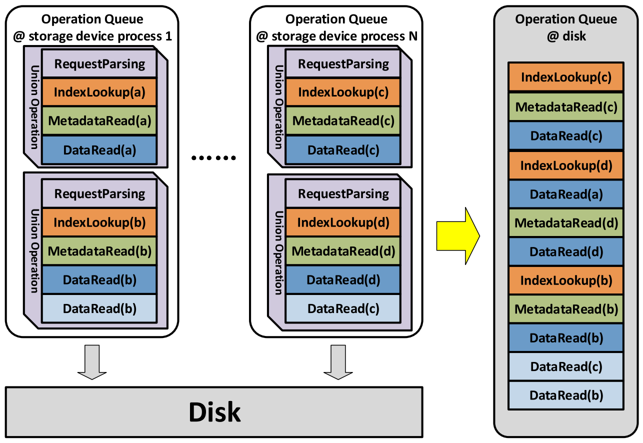


Figure 2. Queues of a storage device (N_{be} processes per storage device)

Finally, we get the Laplace Transform of the waiting time pdf. for the M/G/1 queue of union operations based on Pollaczek-Khinchin formula [24].

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

The process uses 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 response latencies at backend tier are

$$S_{be}(t) = (W_{be} * parse_{be} * index * meta * data)(t). \quad (1)$$

When $N_{be} \in \{2, 3, 4, 5...\}$: Fig. 2 displays the queueing status for a storage device with multiple processes at the backend server. When a request enters the system, 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. Operations that cannot be served from memory enter the operation queue at 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 response latencies for such a queueing network, as discussed in Section VI. And the solution for $N_{be} = 1$ cannot be directly applied here. To solve this problem, we continue relying on the elegant abstraction of union operation. And the key idea is to transform the queueing network of $N_{be} > 1$ to the queueing network of $N_{be} = 1$. To conduct such transformation, we treat the disk response latency as “disk service time” for each process. And then we only have to deal with one process similar to $N_{be} = 1$. The overall distribution of response latencies is the same as the latency distribution of the one process scenario because the processes are identical to each other.

There are at most N_{be} operations in the disk queueing system, and we hence use the M/G/1/K queue (M/G/1 queue with K buffer) to model the disk queue, where $K = N_{be}$. However no closed-form solution exists for the sojourn time (terminology in queueing theory, equals to response latency) distribution of M/G/1/K queue. Considered that

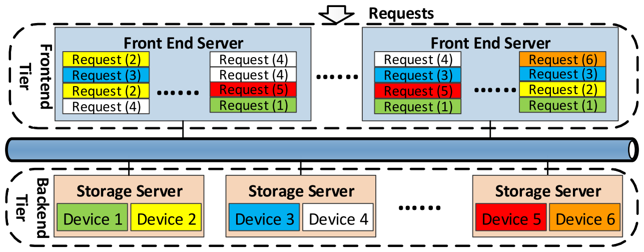


Figure 3. The request queues at the frontend servers

a lot of approximating approaches for solving M/G/1/K queue are developed based on M/M/1/K queue [25], we use M/M/1/K queue as the approximation for simplicity. The accuracy of the approximation is acceptable according to our evaluation shown in Section V-B. Moreover, J.M.Smith [25] provides an analysis and evaluations on approximating M/G/1/K queue using M/M/1/K queue. Other approximating approaches would be also applicable for the model, on the condition that the sojourn time pdf. of the approximation has a closed-form Laplace Transform, which is needed in the following calculations. We do not distinguish different operations here because the different type operations mix together in the disk queue. Suppose that the raw average service time of disk is b, and the operation arriving rate at disk is $r_{disk} = m_{index}r + m_{meta}r + m_{data}r_{data}$. The service rate of disk is $v = 1/b$, utilization is $u = r_{disk}b$. According to the M/M/1/K formula [24], The Laplace transform of the the pdf., and the mean value of disk response latencies (or the "disk service time" from the standpoint of one process) are

$$\mathcal{L}[S_{diskN}](s) = \frac{vP_0}{1-P_K}\frac{(1-(\frac{r_{disk}}{v+s})^K)}{v-r_{disk}+s},$$

$$\bar{S_{diskN}} = \frac{\bar{N}}{r(1-P_K)},$$

where, $P_i = \frac{(1-u)u^i}{1-u^{K+1}}, i = 0, ..., K$ is the probability of system buffer state, $\bar{N} = \frac{u(1-(K+1)u^K + Ku^{K+1})}{(1-u)(1-u^{K+1})}$.

Then we can calculate the distribution of response latencies for a storage device at the backend tier as same as when $N_{be} = 1$, with the following Equations.

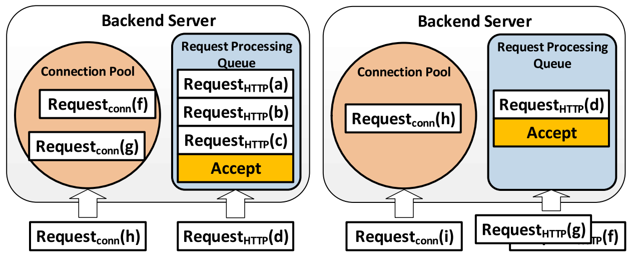
$$index_d(t) = meta_d(t) = data_d(t) = S_{diskN}(t)$$
$$\bar{index} = \bar{meta} = \bar{data} = \bar{S_{diskN}}$$
$$r = r/N_{be}$$

C. Performance Modeling at Frontend Tier

The response latency for a request at the frontend tier contains three main components: queueing latency in the process at the frontend tier, waiting time for being accept()-ed at the backend tier, and 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 is identical to each other. So, the distribution of overall queueing latencies is the same as that of any single process. And the frontend tier of heterogeneous servers can be divided into several sets of homogeneous servers, and the distribution of queueing latencies can be calculated separately. Fig. 3 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, arriving rate for one process P_i at the frontend tier is $r_i = \frac{r}{N_{fe}}$, and $parse_{fe}(t)$ is the distribution of request parsing time for processes in the frontend tier. We could also use the M/G/1 queue to model the queue of one process in the frontend tier [23]. Then, the Laplace transform of queueing latency pdf. is

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



(a) before accept() (b) after accept()

Figure 4. Waiting time for being accept()-ed

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, the connecting requests from the frontend tier have to wait in the connection pool before being accept()-ed by a process for the storage device at the backend server. Since the accept() operation is scheduled as identical as normal operations for processing requests, the accept() operation also has to wait in the request processing queue. Fig. 4a 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. 4b 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 follow the Poisson process. According to the PASTA theorem [26], the lifetime pdf. of accept() operations $A(t)$ is the same as the waiting time pdf. of the request processing queue at the backend server. The connecting requests may arrive at any time point during the lifetime of an accept() operation. Suppose that the waiting time pdf. of connecting requests 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

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

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

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

$$S_{fe}(t) = (S_q * W_a * S_{be})(t) \quad (2)$$

D. 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 corresponding response latencies at the frontend tier is $S_j(t)$. The requests arriving rate for D_j is r_j. Since we have already known the distribution of response latencies for each storage device, we could calculate the cdf. of response latencies for the overall system ($S(t)$) with the following formula.

$$S(t) = \frac{\sum_{D_j \in D}[r_j S_j(t)]}{\sum_{D_j \in D} r_j} \quad (3)$$

IV. Estimating the Model Parameters

In order to predict the distribution of response latencies, our model requires several parameters as inputs. 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.

A. Device Performance Properties

The distribution of disk service times. 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 operation to 1 to avoid operations queueing. Finally, we use the distribution fitting to get the distribution of disk service times for different operations. 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 III. Moreover, the mixture of distributions could be used for fitting the recorded latencies, as long as the mixture 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. And Fig. 5 shows the fitting results with 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 average 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 times.

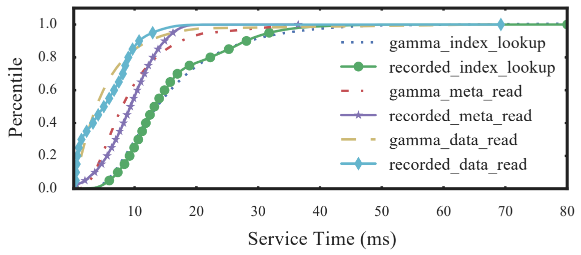


Figure 5. The results of fitting the disk service times

The distribution of request parsing latencies. 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 the 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 receive a request and start responding) and D_{bp} (duration between a backend tier process receive a request and start 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 latencies. In our testbed, the request parsing latency is almost constant (Degenerate distribution).

B. System Online Metrics

Generally, the arriving rate of requests 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 latency threshold to distinguish cache hit and 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. And in our testbed, we use 0.015ms as the latency threshold. Linux only provides the summary value for disk service time. And in order to obtain the average service times of different type operations, we exploit the proportion of service times from Section IV-A. Suppose that r is the request arriving rate, and r_d is the arriving rates for 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.

$$b_i/p_i = b_m/p_m = b_d/p_d$$

$$m_i b_i r + m_m b_m r + m_d b_d r_d = (m_i r + m_m r + m_d r_d)b$$

V. 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.

A. Experimental Setup

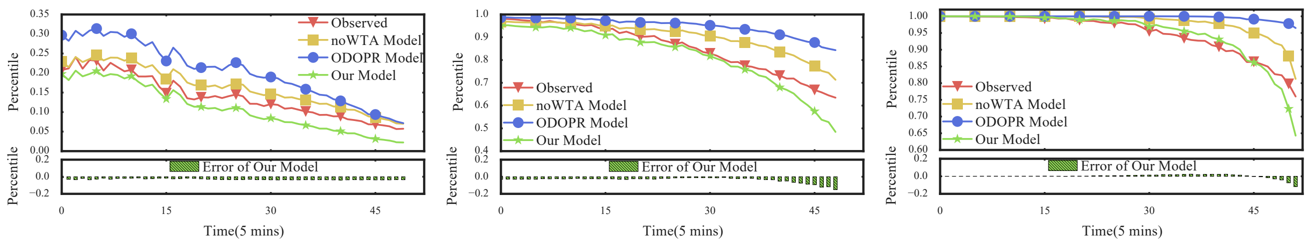
Our testbed is a 7-nodes OpenStack Swift cluster, including 3 frontend servers and 4 backend servers, and we use 1Gbps Ethernet to connect the frontend and backend servers. There is a 1TB HDD disk attached to each backend server. Data objects are mapped to 1,024 partitions based on hashing, and each 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 40Gbps Infiniband. Such configuration prevents workload generators from being the bottleneck of the whole system. Each node has four 2.4GHz Intel E5620 quad-core CPUs, 24GB of memory, and runs Centos 7. Except that we limit the memory of the backend servers to 5GB. 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 [16]).

We generate the workload based on a 50 hours trace of media objects accessing from Wikipedia. This trace is extracted from the trace provided along with wikibench [15] (the URL of media request 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 32KB. The average size of requests is about 10KB.

We use the SwiftStack Benchmark Suite (ssbench) [27] 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 the frontend server instead of the ssbench worker. 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 predicting the response latency of the cloud object storage system. We control the rate of generating requests at the ssbench master.

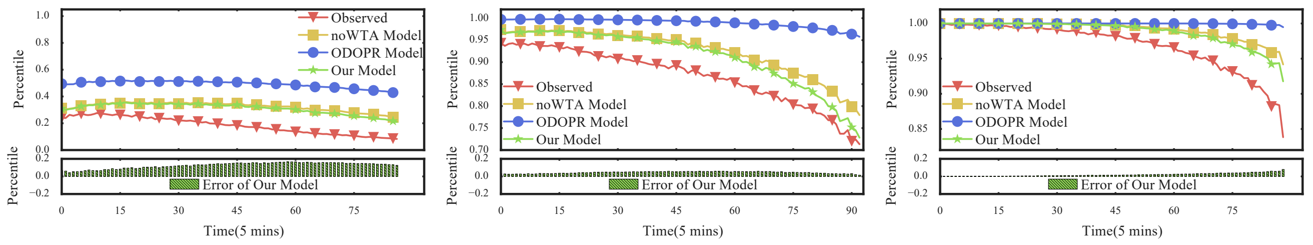
B. The Accuracy of the Model

We conduct a set of experiments to evaluate the accuracy of our model on predicting the response latency percentiles of cloud object storage systems. We perform the evaluation in two scenarios, *S1* and *S16*. At the backend tier, we use the configuration of 1 process per storage device for the scenario *S1* and the configuration of 16 processes per storage device for the scenario *S16*. For each scenario, we carry out the experiments with 3 different SLAs (response latency requirements of 10ms, 50ms, and 100ms). 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 both frontend and backend tiers for each minute. We calculate and predict the percentile of requests meeting SLA using the average value of 5 minutes of the same arriving rate of requests. As discussed in Section III-A, 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 [3]–[6], [11] predict the average performance metrics rather than response latency percentiles; 2). Existing



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

Figure 6. Prediction results of different SLAs for scenario *S1* (single process per storage device). The x-axis shows the execution time.



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

Figure 7. Prediction results of different SLAs for scenario *S16* (16 processes per storage device). The x-axis shows the execution time.

models [28], [29] rely on simulation-based technique for prediction; 3). Existing models [11]–[13] focus on modeling different factors (e.g. data striping) in the system instead of the factors addressed by our model.

We generate the workloads by changing the request arriving rate of the trace described in Section V-A. In order to control the arriving rate of requests, we change the timestamp field of each request in the trace. With our modification, the workload contains 3 phases: *warmup phase* lasts 3 hours with a fixed arriving rate, *transition phase* lasts 1 hour with a fixed arriving rate, and *benchmarking phase* with a varying arriving rate and each arriving rate lasts 5 minutes. 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 are different for scenario *S1* and *S16* 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 both *S1* and *S16* during the transition phase. During the benchmarking phase, the arriving rate starts at 10 requests per second and ends at 350 for *S1* (600 for *S16*), with the increase of 5.

Fig. 6 and Fig. 7 show the observed percentiles of requests meeting SLAs (10ms, 50ms, 100ms), and the predicted percentiles of our model for the scenario *S1* and scenario *S16* respectively. And Fig. 6 and Fig. 7 also display the prediction errors of our model (the difference between predicted and observed percentiles) at the bottom of each graph. In Fig. 6 and Fig. 7, 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. 6, so are the graphs in Fig. 7. The reason is that timeouts do not occur at the exactly identical time point in different experiments (randomness exists in the replica choosing scheme of OpenStack Swift). For all figures in Section V, we use identical scales of y-axis for prediction errors to enable comparability.

For the scenario *S1* (Fig. 6), we observe that the prediction accuracy of our model decreases as the workload increases. And the reasons are 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 III-C), 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.

For the scenario *S16* (Fig. 7), we observe that the prediction errors are relatively larger than the prediction errors in the scenario *S1*. The reason for lower accuracy is that there exist systematic errors as we use M/M/1/K to approximate M/G/1/K for calculating the distribution of disk response latencies, when $N_{be} > 1$ (discussed in Section III-B). Moreover, while our model almost always underestimates the percentiles of requests meeting SLA for the scenario *S1*, it always overestimates the percentiles for the scenario *S16*. It is because our model assumes that the requests of a storage device are uniformly distributed among its corresponding processes. And load imbalance occurs in practice, because

Table I

THE SUMMARY OF PREDICTION ERRORS FOR OUR MODEL

Scenario	SLA	Best Case	Worst Case	Mean
S1	10ms	1.01%	3.82%	2.91%
	50ms	0.86%	15.04%	3.47%
	100ms	0.02%	11.70%	1.26%
S16	10ms	4.29%	16.61%	12.57%
	50ms	1.48%	5.85%	4.48%
	100ms	0.08%	7.94%	1.96%

each process may batch accept() requests or be blocked by long processing requests. The load imbalance leads to the increased response latencies and the decreased percentiles of requests meeting SLA.

Table I summarizes the prediction errors (absolute value) of our model for the different scenarios and SLAs. For all cases, the average prediction error of our model is 4.44% and the worst case error is 16.61%.

C. The Contribution of Core Components

We reveal the contributions of the core components to the overall accuracy of our model 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.

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 architecture. Compared to the ODOPR model, which does not consider these factors, our model reduce the average prediction errors by 36% to 73% (relative percentage), Fig. 6 and Fig. 7 show the prediction results of the ODOPR model and our model.

The contribution of modeling waiting time for being accept()-ed: Compared to the noWTA model, which does not consider the WTA, our model reduces the average prediction errors by 9% to 61% (relative percentage) for different scenarios and SLAs, except the 10ms SLA in the scenario *S1*. As a matter of fact, our model increases the average prediction errors by 0.46% for the 10ms SLA in the scenario *S1*. This is because the overestimation of the WTA introduces more errors than overlooking the WTA. The 10ms SLA is an extreme case, and less than 25% of requests return within 10ms even under the lightest workload. It is worth to mention that the WTA itself decreases in the scenario *S16* compared to the scenario *S1*. Because there are 16 processes accept()-ing connecting requests in the scenario *S16*, 16 times the scenario *S1*. Fig. 6 and Fig. 7 show the prediction results of the noWTA model and our model.

Table II

THE MEAN PREDICTION ERRORS OF DIFFERENT MODELS

Scenario	SLA	Our Model	ODOPR Model	noWTA Model
SW1	10ms	2.91%	6.54%	2.45%
	50ms	3.47%	9.41%	5.18%
	100ms	1.26%	4.80%	3.26%
SW16	10ms	12.57%	30.74%	13.87%
	50ms	4.48%	12.10%	5.69%
	100ms	1.96%	3.11%	2.28%

Table II compares the mean prediction errors (absolute value) of different models for different scenarios and SLAs.

VI. RELATED WORK

Queueing Network: General queue networks fail on modeling 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. As a matter of fact, the cloud object storage system could be modeled by Layered Queueing Network (LQN). 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 response latencies for the LQN using FCFS queueing discipline. The state-of-the-art LQN solver DiffLQN [30] focus on calculating the mean values, and Line [31] calculates the distribution of response latencies for LQN using PS discipline. Some LQN solvers support estimating response latency distribution with simulation, which is high time consumption.

Multi-tiered Web Application Performance Modeling: The early models, e.g. Yaksha [3], 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. Different from our model that predicts response latency percentiles, the recent models generally predict the average performance metrics (e.g. throughput, average response latency) for a particular scenario. For example, Urgaonkar et al. [4] use closed queueing networks to model session-based web applications, Calheiros et al. [5] rely on queueing networks for modeling applications running in virtualized environments, and Han et al. [6] build a performance model for latency-critical applications in the context of sharing resources with offline batch jobs. Nguyen et al. [32] 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 distribution of response latencies. Watson et al. [28] build a simulation-based model for predicting the distribution of response latencies, and we focus on building an analytic-based model.

Storage System Performance Modeling: The focuses of modeling different types of storage systems are different. Wu et al. [11] 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 [12], [13] generally exploit fork-join queue for modeling data striping, where an IO request is split into several sub-requests of different storage devices. Spotify also builds a performance model [29] to predict the distribution of response latencies for their "cloud object storage system", which only works as a cache tier of Amazon S3. However, their model is simulation-based model due to relying on workload-dependent benchmarking for prediction.

VII. Conclusion

In this paper, we present an analytical performance model that predicts the percentile of requests meeting SLA for the cloud object storage system using event-driven architecture. 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 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. Moreover, our model is available as open-source software from https://github.com/ysu-hust/cosmodel.

Acknowledgment

This work was supported by the National High Technology Research and Development Program (863 Program) No.2015AA015301, the National Key Research and Development Program of China under Grant 2016YFB1000202; NSFC No.61472153, No. 61502191; State Key Laboratory of Computer Architecture, 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] "Openstack swift," https://docs.openstack.org/developer/swift/, 2016-10-19.

[3] A. Kamra, V. Misra, and E. M. Nahum, "Yaksha: A self-tuning controller for managing the performance of 3-tiered Web sites," in *IWQOS '04*, Jun. 2004, pp. 47–56.

[4] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An Analytical Model for Multi-tier Internet Services and Its Applications," in *SIGMETRICS '05*. New York, NY, USA: ACM, 2005, pp. 291–302.

[5] R. N. Calheiros, R. Ranjan, and R. Buyya, "Virtual machine provisioning based on analytical performance and qos in cloud computing environments," in *ICPP '2011*, Sept 2011, pp. 295–304.

[6] R. Han, J. Wang, S. Huang, C. Shao, S. Zhan, J. Zhan, and J. L. Vazquez-Poletti, "Pcs: Predictive component-level scheduling for reducing tail latency in cloud online services," in *ICPP '15*, Sept 2015, pp. 490–499.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *SOSP '07*. New York, NY, USA: ACM, 2007, pp. 205–220.

[8] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a Needle in Haystack: Facebook's Photo Storage," in *OSDI '10*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[9] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An Analysis of Facebook Photo Caching," in *SOSP '13*. New York, NY, USA: ACM, 2013, pp. 167–181.

[10] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell, "Ambry: LinkedIn's Scalable Geo-Distributed Object Store," in *SIGMOD '16*. New York, NY, USA: ACM, 2016, pp. 253–265.

[11] Y. Wu, F. Ye, K. Chen, and W. Zheng, "Modeling of Distributed File Systems for Practical Performance Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 156–166, Jan. 2014.

[12] E. Varki, "Response time analysis of parallel computer and storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 11, pp. 1146–1161, Nov. 2001.

[13] A. S. Lebrecht, N. J. Dingle, and W. J. Knottenbelt, "Analytical and Simulation Modelling of Zoned RAID Systems," *The Computer Journal*, p. bxq053, Jun. 2010.

[14] T. Brecht, D. Pariag, and L. Gammo, "Acceptable Strategies for Improving Web Server Performance," in *ATEC '04*. Berkeley, CA, USA: USENIX Association, 2004, pp. 20–20.

[15] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia Workload Analysis for Decentralized Hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, Jul. 2009.

[16] "Swift eqiad cluster report," https://ganglia.wikimedia.org/latest/?r=week&cs=&ce=&c=Swift+eqiad, 2016-10-19.

[17] "Wikipedia openstack swift cluster live status," https://grafana.wikimedia.org/dashboard/file/swift.json?var-DC=eqiad, 2016-10-19.

[18] "Openstack swift and many small files," http://engineering.spilgames.com/openstack-swift-lots-small-files/, 2016-10-19.

[19] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "f4: Facebook's Warm BLOB Storage System," in *OSDI '14*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 383–398.

[20] "Openstack swift use cases," https://www.swiftstack.com/customers, 2016-10-19.

[21] "Ceph use cases," http://ceph.com/users/, 2016-10-19.

[22] Q. Fan and Q. Wang, "Performance Comparison of Web Servers with Different Architectures: A Case Study Using High Concurrency Workload," in *HotWeb '15*, Nov. 2015, pp. 37–42.

[23] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power Management of Online Data-intensive Services," in *ISCA '11*. New York, NY, USA: ACM, 2011, pp. 319–330.

[24] J. Sztrik, "Basic queueing theory," *University of Debrecen, Faculty of Informatics*, vol. 193, 2012.

[25] J. M. Smith, "Optimal design and performance modelling of m/g/1/k queueing systems," *Mathematical and Computer Modelling*, vol. 39, no. 910, pp. 1049 – 1081, 2004.

[26] R. W. Wolff, "Poisson arrivals see time averages," *Operations Research*, vol. 30, no. 2, pp. 223–231, 1982.

[27] "Swiftstack benchmark suite (ssbench)," https://github.com/swiftstack/ssbench, 2016-10-19.

[28] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang, "Probabilistic Performance Modeling of Virtualized Resource Allocation," in *ICAC '10*. New York, NY, USA: ACM, 2010, pp. 99–108.

[29] R. Yanggratoke, G. Kreitz, M. Goldmann, and R. Stadler, "Predicting response times for the Spotify backend," in *CNSM '12*, Oct. 2012, pp. 117–125.

[30] T. Waizmann and M. Tribastone, "DiffLQN: Differential Equation Analysis of Layered Queuing Networks," in *ICPE '16 Companion*. New York, NY, USA: ACM, 2016, pp. 63–68.

[31] J. F. Pérez and G. Casale, "Assessing SLA Compliance from Palladio Component Models," in *SYNASC '13*, Sep. 2013, pp. 409–416.

[32] M. Nguyen, Z. Li, F. Duan, H. Che, and H. Jiang, "The Tail at Scale: How to Predict It?" in *HotCloud '16*. Denver, CO: USENIX Association, Jun. 2016.