Scalable QoS for Distributed Storage Clusters using Dynamic Token Allocation

Yuhan Peng
Department of Computer Science
Rice University
Houston, TX, 77005
Email: yp10@rice.edu

Qingyue Liu*, Peter Varman†
Department of Electrical and Computer Engineering
Rice University
Houston, TX, 77005
Email: *ql9@rice.edu, †pjv@rice.edu

***Abstract*—The paper addresses the problem of providing performance QoS guarantees in a clustered storage system. Multiple related storage objects are grouped into logical containers called buckets, which are distributed over the servers based on the placement policies of the storage system. QoS is provided at the level of buckets. The service credited to a bucket is the aggregate of the IOs received by its objects at all the servers. The service depends on individual time-varying demands and congestion at the servers.**

We present a token-based, coarse-grained approach to providing IO reservations and limits to buckets. We propose pShift, a novel token allocation algorithm that works in conjunction with token-sensitive scheduling at each server to control the aggregate IOs received by each bucket on multiple servers. pShift determines the optimal token distribution based on the estimated bucket demands and server IOPS capacities. Compared to existing approaches, pShift has far smaller overhead, and can be accelerated using parallelization and approximation. Our experimental results show that pShift provides accurate QoS among the buckets with different access patterns, and handles runtime demand changes well.

***Index Terms*—distributed storage, coarse-grained QoS, reservations, limits, token based scheduling**

I. Introduction

Clustered storage systems such as Ceph [1], GlusterFS [2], Amazon's Cloud Storage [3], FAB [4], Kudu [5], Dynamo [6], Cassandra [7], HDFS [8] and vSAN [9], provide a scalable and economical approach for the storage of huge data sets over multiple storage servers. In such systems, multiple related objects are often grouped into logical containers called *buckets*. Objects are replicated on multiple servers for fault tolerance and performance, large objects are sharded for manageability, and decentralized consistency protocols enable highly concurrent, distributed access to data.

With shared storage becoming the norm in datacenter deployments, performance QoS is becoming an increasingly important requirement of storage systems. In our model, QoS is provided at the level of buckets. The creator (owner) of a bucket contracts for storage space and a specified IO rate (lower and upper bounds) for bucket access. IO requests to bucket objects may be generated by multiple independent applications, and the requests follow independent paths from a requestor node to the server holding the object. A bucket's objects are distributed over the cluster, and QoS allocations must account for service received at multiple servers.

There has been considerable past research [10]–[18] on providing QoS controls for virtual machines (VMs) sharing a single server-attached [17] or SAN-attached storage array [18]. All IO requests of a VM go through a hypervisor module, which controls when and in what order these requests are forwarded to the storage backend to enforce QoS guarantees. In previous work on QoS for distributed storage clusters, [15], [17], [19]–[21] considered a model where all requests from a client (the QoS entity) pass through a common ingress point before being dispersed to the servers. This is reasonable in centralized client applications, but is onerous when the bucket is accessed by multiple independent distributed applications. The ingress point collects global information about a client's requests and adds meta-information to a request before forwarding it to the appropriate server. The servers use sophisticated schedulers that use the meta-information to arbitrate requests.

A method for providing bucket QoS using a token-based approach was recently proposed [22]. The framework called *bQueue* models token allocation as a max-flow problem on a graph with servers and buckets. However, since max-flow algorithms have high time complexities and cannot be parallelized, it is not practical to use *bQueue* in storage systems at large scales.

In this paper, we present a token-based, coarse-grained approach to providing IO reservations and limits to buckets. We propose a novel token allocation algorithm, **pShift**, that works in conjunction with token-sensitive scheduling at each server, to control the aggregate IOs received by each bucket on multiple servers. *pShift* determines the optimal token distribution based on estimated bucket demands and server IOPS capacities. It periodically distributes tokens to the storage nodes to shape their local request schedulers. Compared to fine-grained QoS approaches, our scheme does not require any additional overhead for per-request metadata-based tag computations at the servers, and only uses simple token-sensitive, round-robin schedulers. Moreover, it does not require the requests of a bucket to be funneled through a common ingress node. We believe that *pShift* is a novel algorithm for distributed resource allocation that provides coarse-grained

bucket QoS in distributed storage systems.

The rest of the paper is organized as follows. In Section II we provide additional background on the system architecture for bucket QoS. Section III presents the design of the *pShift* algorithm and its usage in QoS allocation. Section IV shows the performance evaluation of *pShift* under different runtime scenarios. Section V discusses related QoS schemes and additional design issues in the *pShift* approach. Finally, Section VI provides conclusions and summarizes future work.

II. Overview

In this section, we give an overview of our token-based coarse-grained QoS architecture. The platform consists of a cluster of *servers* (storage nodes) that collectively store and manage data collections called **buckets**. A bucket is a grouping of logically-related objects (*e.g.* fragments of a file or files in a directory) that are distributed among the servers. A server hosts objects from multiple buckets and the objects in a bucket may be spread over several servers. Figure 1 shows an example of the system architecture. There are 4 buckets indicated by different colors, whose objects are distributed on 3 server nodes. Node 1 hosts red and green buckets; Node 2 hosts red, orange and blue buckets; and Node 3 hosts red, green and orange buckets.

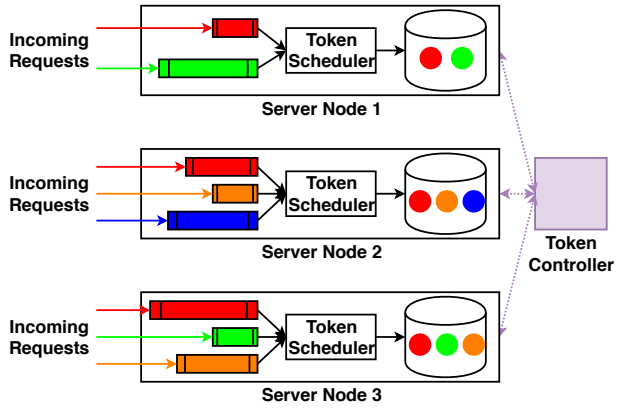


Fig. 1: Token QoS architecture.

Each server receives read and write IO requests for its stored objects. The requests are queued at the server in bucket-specific queues and dispatched to the backend devices by a QoS scheduler called the **Token Scheduler**. A controller process running on a dedicated computing node (or one of the server nodes) periodically receives status information from the storage nodes and pushes dynamic QoS control parameters (encoded as *tokens*) back to them for use by their respective Token Schedulers.

A. QoS Model

QoS is specified at the granularity of buckets using two QoS control inputs **reservation** R_i and **limit** L_i for bucket i. Service time is divided into equal-sized non-overlapping intervals called *QoS periods*. These controls specify that the total number of IOs for bucket i objects done in a QoS period must be *at least* R_i and must *not exceed* L_i. The problem is complicated because bucket objects are distributed across multiple server nodes and requests go directly to the servers without intervention.

The Token QoS framework uses tokens to control the scheduling of requests at individual servers. There are two types of tokens associated with a bucket: reservation tokens (**R-tokens**) and limit tokens (**L-tokens**). An R-token for a bucket implies priority in scheduling the requests of that bucket. A bucket without any R-tokens at a server will only receive service when there are no pending requests for buckets with R-tokens at the server. L-tokens control the total number of IOs for a bucket serviced at a storage node. A bucket without L-tokens will not be scheduled at a server. The controller periodically allocates R-tokens and L-tokens to the servers using the *pShift* algorithm. The algorithm is based on the current status of the system: current service rates (IOPS) of the servers, demand distribution of the buckets at the servers, and the number of unconsumed tokens of a bucket (representing unfulfilled reservation or limit).

Algorithm 1: Token Scheduler

next = 0;
while *(TRUE)* **do**
  Step 1a: Search the request queues in round-robin order starting from *next* for the first queue that has both pending requests and R-tokens;
  if *there is no such queue* **then**
    Go to **Step 2**
  Step 1b: Schedule a request from the queue found in **1a**, decrement the number of R-tokens for this bucket by 1, update *next*; **continue**;
  Step 2a: Search the request queues in round-robin order starting from *next* for the first queue that has both pending requests and L-tokens;
  if *there is no such queue* **then**
    Go to **Step 3**
  Step 2b: Schedule a request from the queue found in **2a**, decrement the number of L-tokens for this bucket by 1, update *next*; **continue**;
  Step 3: Delay a small amount; **continue**;

The Token Scheduler at a server uses its current allocation of R-tokens and L-tokens received from the controller in scheduling its requests. Algorithm 1 shows the request scheduler. The scheduler will not idle if there are any requests pending in its queues, unless all buckets with pending requests have reached their limit for the QoS period. It gives priority to buckets with non-zero R-tokens (called reservation requests) over those without any reservation tokens (called normal requests). It chooses requests fairly among pending reservation requests by serving them in round-robin order. If there are no reservation requests it chooses among normal requests that have not exceeded their limit, again in a round-robin manner. If there are no pending requests or if the only pending requests are for

buckets that have reached their limit, the scheduler will wait for a short interval and try again. One R-token (L-token) of a bucket will be consumed for each reservation (normal) request of the bucket that is scheduled.

III. Token Allocation

In this section, we describe the token allocation algorithm to distribute R-tokens and L-tokens to the servers. In Section III-A we formalize the token allocation problem, and we concentrate on the allocation of R-tokens since the procedure is similar for L-tokens. Section III-B describes the *pShift* algorithm, and the optimization approaches are discussed in Section III-C. Section III-D describes how *pShift* algorithm is used in the Token QoS architecture.

A. Problem Statement

We first formalize the token allocation problem as follows. Let $\mathbf{B}$ and $\mathbf{S}$ denote the sets of buckets and servers respectively. In a QoS period, bucket i must meet a reservation R_i and has an estimated aggregate IO demand (total number of requests across all servers) of D_i. Without loss of generality, we assume that $D_i \geq R_i$ for all $i \in \mathbf{B}$, *i.e.* every bucket has sufficient demand to meet its reservation. If not, the best one can do is to match its demand, so we temporarily set $R_i = D_i$. Each server has a current service rate (which may be a workload-dependent estimate) that determines the number of IOs it can do per unit time. C^j indicates the estimated number of IOs that server j can perform in the QoS period.

For each bucket $i \in \mathbf{B}$, the token allocation algorithm will distribute a total of R_i tokens among the servers $j \in \mathbf{S}$. Let d_i^j and a_i^j denote the **demand** and the number of **allocated tokens** for bucket i on server j. Note that $\sum_{\mathbf{j}} d_i^j = D_i$ and $\sum_{\mathbf{j}} a_i^j = R_i$. When a server does an IO for a bucket, it *consumes* one of the bucket's tokens. If all a bucket's tokens are consumed by the end of the QoS period, then the bucket's reservation requirements will have been met.

There are two situations when a server j may have unconsumed tokens. Firstly, if j is allocated more tokens for bucket i than its demand (*i.e.* $d_i^j < a_i^j$), then at most d_i^j tokens of bucket i can be consumed on server j. Secondly, if the total number of tokens allocated to server j (*i.e.* $\sum_{i \in \mathbf{B}} a_i^j$) exceeds its IO capacity C^j, then even with Token Scheduling, some tokens will not be consumed. In the first case we say that server j has **strong excess tokens** for bucket i, and in the second case we say that server j has **weak excess tokens**. A server with weak excess tokens is said to be **overloaded**.

The goal of the token allocation algorithm is to maximize the total number of tokens that will be consumed. If the tokens for all buckets on all servers are consumed, then all reservation requirements are met. In other cases, the distribution of demands on servers may preclude meeting all reservations *irrespective* of the scheduling or token allocation method. For instance, a server may become overloaded if all the demands of several buckets are concentrated on it. One cannot spread the load to other servers since they do not have demand for these buckets. In this case the allocation aims to maximize the number of reservation IOs performed.

	d_1	d_2	a_1	a_2
Red	150	50	75	25
Blue	50	50	50	50

TABLE I: Configuration of Examples 1 and 2. Servers 1 and 2 have capacity 100 each. Red and blue buckets have reservation of 100 each. d_i and a_i show demand and allocation on server i.

We refer to a **configuration** of the system by its demand distribution, server IO capacities, and token allocations: $\{[d_i^j, C^j, a_i^j] : i \in \mathbf{B}, j \in \mathbf{S}\}$. For a given configuration μ, we define the **effective server capacity**, $\phi^j(\mu)$, to be the number of tokens that server j consumes: $\phi^j(\mu) = min(C^j, (\sum_{i \in \mathbf{B}} min(a_i^j, d_i^j))$. The **effective system capacity**, $\Phi(\mu)$, is the sum of the effective capacities of individual servers, *i.e.* $\Phi(\mu) = \sum_{\mathbf{j}} \phi^j(\mu)$. For different token allocations, the effective system capacities Φ may be different, and the goal of token distribution is to find an allocation that *maximizes* Φ, which we call an *optimal* allocation. Note that the optimal allocation may not be unique.

Example 1: Table I shows a system of 2 servers with IO capacity 100, and two buckets (red and blue) with reservations of 100 each. The demands and token allocations are shown in the table. Server 1 receives a total of 125 tokens that exceeds its capacity, so it is overloaded with 25 weak excess tokens. Server 2 has only 75 tokens so is *underloaded*. Note that the allocation of a bucket on any server does not exceed the bucket demand on that server, so there are no strong excess tokens. The effective capacity of server 1 is $min(100, (min(75, 150) + min(50, 50))) = 100$ which is its IO capacity. Similarly, the effective capacity of server 2 is $min(100, (min(25, 50) + min(50, 50))) = 75$. The effective system capacity of the configuration is 175. The blue bucket meets its reservation but the red bucket gets only 75 IOs.

B. pShift Algorithm

To find an optimal allocation we only consider *prudent allocations* in which there are no strong excess tokens at any server, *i.e.* the token allocations satisfy $a_i^j \leq d_i^j$ for all $i \in \mathbf{B}, j \in \mathbf{S}$. A prudent allocation always exists since $\sum_{\mathbf{j}} a_i^j = R_i \leq D_i = \sum_{\mathbf{j}} d_i^j$. Furthermore, it can be shown that optimal allocations are either prudent or can be transformed to a prudent allocation with the same Φ.

The algorithm operates as follows. An initial prudent configuration is obtained by distributing the R_i tokens of bucket i among the servers in proportion to its demand on the server, *i.e.* server j gets an initial allocation of $a_i^j = R_i \times d_i^j / D_i$. Since $D_i \geq R_i$, we always have $a_i^j \leq d_i^j$, so there are no strong excess tokens. Following this allocation some servers may be overloaded, some may be underloaded, and the rest may exactly match their IO capacities.

The algorithm then iteratively attempts to find another prudent allocation with higher Φ by moving tokens from an

	d_1	d_2	d_3	a_1	a_2	a_3
Red	150	50	0	75	25	0
Blue	0	150	50	0	75	25
Green	50	0	50	50	0	50

TABLE II: Configuration of Example 3. All servers have capacity 100 and all buckets have reservation of 100. d_i and a_i are demand and allocation for server i.

overloaded server to an underloaded one. We refer to such a token movement as a *prudent transfer*. Every token moved in this way increases Φ by 1. The algorithm stops when there are no overloaded servers or there are no prudent transfers from an overloaded to an underloaded server possible. The resulting allocation will be optimal.

The prudent transfer of tokens between two servers may be done either directly or indirectly. To effect a *direct transfer*, the donor server must have a sufficient number of tokens of a bucket and the receiver must have high enough demand for that bucket to avoid creating strong excess tokens. Specifically, a prudent transfer of $n \geq 1$ tokens of bucket i from server j to server k requires: (i) $a_i^j \geq n$ and (ii) $d_i^k - a_i^k \geq n$. We refer to $d_i^k - a_i^k$ as the *spare demand* of server k for bucket i, which indicates the number of tokens of bucket i that server k can accept in a prudent transfer.

Example 2: The token distribution of Example 1 is an initial prudent allocation in which tokens are distributed in proportion to the demands. Server 1 has 50 blue tokens that it can donate. However, server 2 cannot accept any blue tokens since it has no spare demand. So no blue tokens can be transferred from server 1 to server 2. On the other hand, server 1 has 75 red tokens it can donate and server 2 has a spare demand of 25 red tokens. So $min\{75, 25\}$ = 25 red tokens can be transferred from server 1 to 2 resulting in both servers having 100 tokens each. After the transfer, Φ increases to 200. The reservations of both buckets are now satisfied.

We now describe a more complicated example where direct token transfer is not possible.

Example 3: Consider three servers of capacity 100 each and three buckets: red, blue and green. The demands and initial token allocations of the buckets are shown in Table II. Server 1 is overloaded (125), server 3 is underloaded (75) and server 2 is full (100). The tokens on server 1 and 3 are not compatible: server 1 can donate red and green tokens but server 3 has no spare demand for either bucket, and so cannot accept them in a prudent transfer. On the other hand, server 3 has a spare demand of 25 for blue tokens but server 1 has no blue tokens to donate. Hence direct transfer of tokens from 1 to 3 is not possible. We therefore look for an indirect transfer using intermediate servers to act as *token brokers* to create a path of compatible token transfers. In this example, server 1 can donate 25 *red* tokens to server 2 in a prudent transfer; this would make server 2 overloaded, but it can get rid of these 25 weak excess tokens by transferring 25 *blue* tokens to server 3. After this brokered transfer, there are no weak or strong excess tokens and all buckets meet their reservations.

When servers j and k do not have compatible donor and receptor tokens an *indirect transfer* may be possible. In an *indirect transfer* of tokens from server j to k, the transfer is effected with the help of intermediate servers $u_1, u_2, \cdots u_s$ (called brokers). A broker accepts a token of some bucket and sends out a token of another bucket. For each of the n tokens transferred from server j to k: j moves a token of bucket b_0 to server u_1 that in turn moves a token of another bucket b_1 to server u_2. Each server u_i, $2 \leq i < s$, accepts a token of a bucket b_{i-1} from u_{i-1} and sends a token of bucket b_i to u_{i+1}. Finally, u_s sends a token of bucket b_s to server k. If the source server j is overloaded by at least n tokens and the sink server k is underloaded by at least n then the above transfer will increase Φ by n.

The pseudo-code in Algorithm 2 shows the overview of *pShift* algorithm. There are three major steps: initial prudent allocation of tokens to the servers (Algorithm 3); creation of the initial Token Movement Map (Algorithm 4); iterative improvement of Φ by prudent transfers of tokens from overloaded to underloaded servers (Algorithm 5).

Algorithm 2: *pShift* Algorithm

```
InitialTokenAllocation();
MakeTokenMovementMap();
ProgressiveShifting();
```

Algorithm 3: InitialTokenAllocation Function

```
for each bucket i ∈ B do
    D_i = Σ_{j∈S} d_i^j;
    A_i = min(D_i, R_i);
for each bucket i ∈ B do
    a_i^j = A_i × d_i^j / D_i;
```

Algorithm 4: MakeTokenMovementMap Function

```
// Create a token map vector for each pair of servers
for each j ∈ S do
    for each k ∈ S, k ≠ j, do
        for each i ∈ B do
            TM_{j,k}[i] = min(a_i^j, d_i^k − a_i^k);
        TMS_{j,k} = Σ_{i∈B} TM_{j,k}[i];
```

1) Token Movement Map: For each pair of servers j and k, we define a *token movement vector* $\mathbf{TM}_{j,k}$ that specifies for each bucket the number of tokens that can be moved between the servers in a prudent transfer. From the discussion in the previous section, for bucket i this is bounded by the number of tokens of bucket i at the source server and the spare demand for i at the destination server. That is: $\mathbf{TM}_{j,k}[i] = min(a_i^j, d_i^k -$

a_i^k). We also define $\mathbf{TMS}_{j,k}$ as the total number of tokens of all buckets that can be moved between the servers in a prudent transfer. Algorithm 4 shows the initialization of the *token movement map*, the collection of the token movement vectors $\mathbf{TM}_{j,k}$ and $\mathbf{TMS}_{j,k}$, for all pairs of servers $j, k \in \mathbf{S}$.

Table III shows the **TM** vectors and **TMS** between each pair of servers for the configuration of Example 3. For instance, server 1 can only transfer 25 *red* tokens to server 2 and cannot transfer any tokens to server 3, hence $\mathbf{TM}_{1,2} = [25, 0, 0]$ and $\mathbf{TMS}_{1,2} = 25$.

	1	2	3
1	-	[25, 0, 0] / **25**	[0, 0, 0] / **0**
2	[25, 0, 0] / **25**	-	[0, 25, 0] / **25**
3	[0, 0, 0] / **0**	[0, 25, 0] / **25**	-

TABLE III: Token Movement Map for configuration of Table II. Each entry is $\mathbf{TM}_{j,k}$ vector / $\mathbf{TMS}_{j,k}$ from server j (row) to server k (column).

2) Progressive Shifting: The token transfer problem will be modeled by a weighted directed graph (called the *token transfer graph*) whose vertices are the servers. There is a directed edge with weight $\mathbf{TMS}_{j,k}$ from vertex j to vertex k if the weight is greater than zero. An edge of weight w between two servers signifies that it is possible to move w tokens from the source to the destination server without creating strong excess tokens (*i.e.* in a direct prudent transfer).

We extend this observation to a path of length $l \geq 2$ from server j to server k going through intermediate vertices $u_1, u_2, \cdots u_{l-1}$. Let the weights of the edges in this path be $w_1, w_2 \cdots w_l$. Denote the *smallest weight* on this path by w_{min}. Then it is always possible to construct a prudent transfer that moves w_{min} tokens from j to the k, by moving w_{min} tokens across each edge in the path *i.e.* from j to u_1, from u_i to u_{i+1}, $i = 2, \cdots l-2$, and from u_{l-1} to k. We call such a path a *shift path*, and an illustration is shown in Figure 2. Shift paths can be found using *breadth-first search (BFS)* on the token transfer graph.

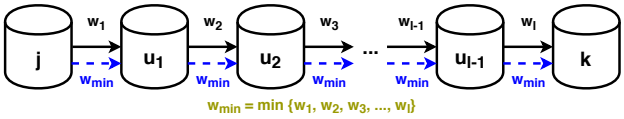


Fig. 2: Illustration of shift path.

Note that the number of tokens at any intermediate server u_i does not change by the prudent transfer, so no weak excess tokens are created at u_i. Also, since the transfer is prudent no strong excess tokens are created. Hence the effective server capacity ϕ^{u_i} of intermediate server u_i does not change.

The effective capacity ϕ^j of the source server j will not decrease if it does not become underloaded due to the tokens moved from it. Correspondingly, the effective capacity ϕ^k of the sink server k will increase if it does not become overloaded due to the tokens transferred into it. As a consequence, the effective system capacity Φ will increase if j is an overloaded server and k is an underloaded server, and the number of tokens transferred does not cause either j to become underloaded or k to become overloaded.

The iterative step of the algorithm identifies a pair of servers, overloaded server j and underloaded server k. Let the amount of overload (i.e. the number of weak excess tokens) on j be denoted by γ^j, and let the amount of underload on k (i.e. $C^k - \sum_{i \in \mathbf{B}} a_i^k$) be denoted by δ^k. Let w_{min} denote the weight of the smallest edge in a chosen shift path between j and k. We define the *shift amount* $\Omega = \min\{\gamma^j, w_{min}, \delta^k\}$. Then the progressive shifting function will move Ω tokens along the shift path. By construction, the resulting allocation will be prudent and Φ will increase by Ω.

Moving tokens along the shift path changes the configuration. Specifically each server w in the path may have a change in the allocation a_i^w for one or more buckets $i \in \mathbf{B}$. This requires recalculating the token movement vectors of all outgoing and incoming edges from and to w for those buckets $i \in \mathbf{B}$ whose token allocations have changed, *i.e.* $\mathbf{TM}_{u,k}[i]$ and $\mathbf{TM}_{k,u}[i]$, $k \in \mathbf{S}$.

Algorithm 5 gives a pseudo-code of the progressive shifting function. It iteratively moves tokens between overloaded and underloaded servers as described above, by exploring shift paths in increasing order of length. The algorithm terminates either when there are no overloaded servers or when there is no shift path. Since moving tokens along a shift path will increase Φ, it is obvious that a globally optimal allocation should not contain any shift path in its token transfer graph. On the other hand, it can be shown that the converse is also true: *i.e.* an allocation with no shift path is optimal. The formal proof is given in [23]. Therefore, the progressive shifting function will terminate with a globally optimal solution that maximizes Φ.

Algorithm 5: ProgressiveShifting Function

```
for l from 1 to (|S| − 1) do
    for each overloaded server P do
        while P is still overloaded do
            Find a shift path (u_0, u_1, ··· u_{l−1}, u_l) from
              server P = u_0 to an underloaded server
              u_l = Q with length at most l using BFS;
            if no shift paths found then
                break;
            Ω = min{γ^P, δ^Q};
            for each pair of adjacent servers (u_i, u_{i+1})
              on the shift path do
                Ω = min(Ω, w_i);
            for each pair of adjacent servers (u_i, u_{i+1})
              on the shift path do
                Select Ω tokens on u_i to move to u_{i+1};
                Update Token Movement Map for all
                  edges into and out of u_i and u_{i+1};
```

C. Performance Optimizations

1) Parallelizing pShift: *pShift* has the opportunity to achieve better scalability by parallelization. The two most time-consuming functions are *MakeTokenMovementMap* and *ProgressiveShifting*, and both can be parallelized. The function *MakeTokenMovementMap* shown in Algorithm 4, has execution time complexity of $O(|\mathbf{S}|^2|\mathbf{B}|)$. Since this function is simply initializing a 2D array of vectors whose entries are independent, this step can be fully parallelized. For instance, we can evenly divide one dimension of the array into several parts and have different threads working on different subarrays.

In the function *ProgressiveShifting* shown in Algorithm 5, the most time-consuming step is the update of the Token Movement Map for each affected server. The update has a complexity of $O(|\mathbf{S}||\mathbf{B}|)$. However, the updates can be parallelized by evenly partitioning the servers on the path, and letting different threads work on updating entries in its partition.

2) Approximation Approach: Another opportunity to accelerate the *pShift* algorithm is to use an approximation approach. This is based on the observation that not all buckets contribute equally in causing overload or reducing underload. In particular, buckets with a small number of tokens and those with only a small amount of spare demand do not contribute much to increasing Φ.

Therefore, instead of maintaining a vector of size $|\mathbf{B}|$ for each edge, *pShift* need only consider the buckets with the top M (or fraction f) token counts and spare demands. Then the controller only considers the reduced information for the prudent transfers. In practice, we found that in most cases we only need a small fraction of the buckets to achieve the optimal or near-optimal Φ (see Section IV-D).

D. Runtime Usage of pShift

In practice, a QoS period is evenly divided into several *redistribution intervals*. At the start of each interval, the controller receives from the servers a projection of their status till the end of the QoS period; specifically it receives the number of unconsumed tokens (unfulfilled reservations and limits), the estimated bucket demands, and the estimated remaining server capacities. The server estimates the demands of a bucket from the size of its pending queue and the number of its arrivals in the past interval, and estimates remaining capacity by extrapolating from its service rate in the past interval. Note that determining the best estimator is an orthogonal issue that will be explored in future work.

If there is no significant change from the previous interval the estimates need not be sent to the controller. The controller computes the number of R-tokens to be allocated for each bucket on each server using *pShift* with the received parameter values. The new token allocations are sent back to the servers. The servers then use the new R-tokens for request scheduling during the next interval. Servers continue to serve requests with the current token allocations while the new token allocations for the next redistribution interval are being computed at the controller. L-tokens are handled in a similar manner on the residual demands and capacities after allocating R-tokens.

IV. Performance Evaluation

A. Experimental Setup

To evaluate the performance of *pShift* algorithm, we implemented the QoS framework using both simulation and direct evaluation on a small Linux cluster. For the former, we create a set of concurrent processes to simulate the storage servers and the token controller, and use a request generator process at each server to create the dynamic workload. The communication overhead is simulated by a built-in delay function. IO service times are randomly drawn from a uniform distribution with mean equal to the reciprocal of the server IOPS capacity and limited variance.

For the actual implementation, we built a prototype on a small cluster of 9 Linux servers connected using QDR InfiniBand (40 Gb/s). Each server node is equipped with an *Intel Xeon Processor E5-2640 v4* [24] CPU with 10 two-way hyper-threaded cores. In our implementation, one thread on each server is responsible for inserting the generated requests to the bucket queues. A second thread at the server runs the Token Scheduler that implements the round-robin scheduling policy. Finally, each storage node uses an independent thread to communicate with the controller node. We use the *send* and *recv* primitives from the socket programming library to handle the communication between the controller and the storage nodes, and *OpenMP interface* to implement the parallel threads.

Two backend servers were used in the evaluation. The first is the well-known distributed memory caching system *memcached* [25] and the second is a conventional block-based Linux storage server. In the first case, each server runs a memcached daemon that is pre-populated with $10,000$ objects of size 4KB each. The requests on each server are generated by an independent *YCSB* workload generator [26], which generates the *core workload A* [27] that gives a $50-50$ mix of *gets* and *puts*. An initial profile run was used to determine that each server had an average throughput of roughly $50,000$ requests per second (RPS). All buckets are continuously backlogged on their active servers with 5 outstanding requests. The scheduler chooses requests from the bucket queues and invokes the memcahced server with a *get* or *put* command. For the storage backend, requests consist of random 4KB reads from a 1GB file created on the server. Using 5 concurrent request threads, each server is initially profiled and found to have an average throughput of roughly 1000 IOPS.

We describe our experimental results below. In Sections IV-B we show that *pShift* can meet reservations and enforce limits in the face of dynamically changing workloads and large numbers of buckets. In Section IV-C we report the measured run times of the parallelized controller algorithm on the Linux server for different numbers of threads and buckets. In Section IV-D results on the tradeoff between run times and accuracy for the approximation controller algorithm on the Linux server are reported. Finally in Section IV-E we report

the results of the evaluation on Linux scheduler to show the workings of the *pShift* approach in a real system.

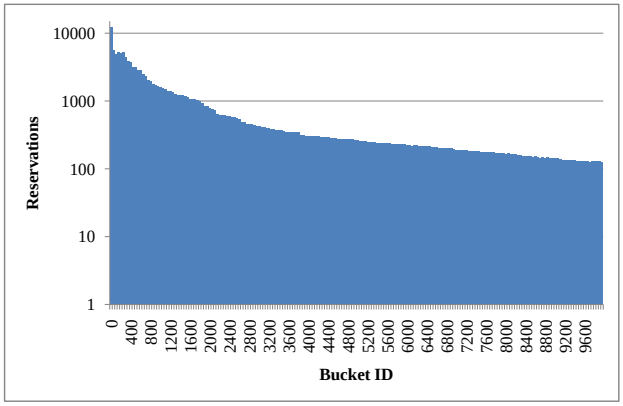
B. QoS Evaluation

Experiment 1: We use the simulator to show how *pShift* handles reservation QoS with a large number of buckets and a dynamically changing workload. There are 64 servers and $10,000$ buckets. Each server has an average capacity of $20,000$ IOPS, and we run the *pShift* algorithm for a full QoS period of 5sec; the throughput per QoS period is therefore roughly $100,000$ IOs. We divide each QoS period into 5 token redistribution intervals *i.e.* a token redistribution is triggered every 1sec using statistics gathered for the last interval.

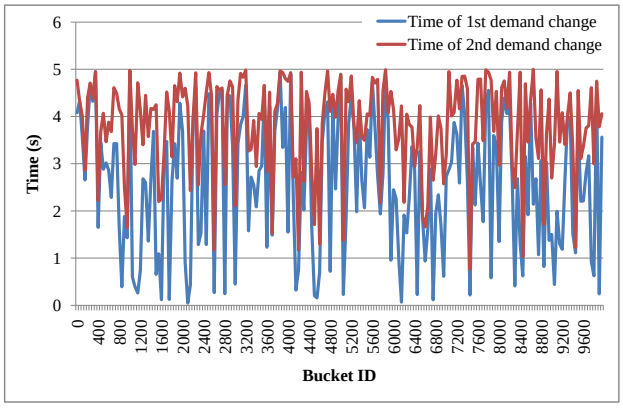
Each server's throughput of $100,000$ IOs (per QoS period) is fully reserved by all the buckets. This causes the greatest stress on the scheduler since there is no spare capacity that can compensate for errors in the token distribution. The reservation of the buckets follows a *Zipf distribution* with exponent factor $s = 0.5$, as shown in Figure 3(a). The *Zipf distribution* simulates a common scenario in practice where most buckets have a relatively low reservation while a few highly-accessed buckets have a much higher reservation. In the figure, there is a factor of 100 between the highest and lowest reservations. All buckets are assumed to have unbounded limits. Each bucket is active on 8 servers at any time. The total demand of a bucket is set to 1.5 times to its reservation. The average request arrival rate for the bucket over all servers is the total demand divided by the length of the QoS period. To stress the *pShift* algorithm, the demands of each bucket are also distributed among the eight servers using a *Zipf distribution* with exponent factor of $s = 0.5$. IO requests for a bucket on a server are generated at a uniform rate proportional to the demand on the server. Buckets may change their demands at random times within the QoS period. When a demand change is triggered, the bucket randomly selects eight servers (which may or may not intersect with the current set), and redistributes the total demand to them based on a *Zipf distribution*. In the initial experiment, a bucket may change its demands up to 2 times in the QoS period. Figure 3(b) shows the times of demand changes of the sample of the buckets. For instance, bucket 6100 has its first demand change early in the 1st interval (around 0.065s).

Figure 3(c) shows the number of requests being completed by all buckets in each of the redistribution intervals. From the figure we can see that, as expected, the number of requests completed by a bucket at the end of the last interval is highly consistent with its reservation. Quantitatively, 99.5% of the buckets meet at least 95% of their reservation. Furthermore, since the servers perform reservation requests in a round-robin fashion, buckets with smaller reservations will complete earlier and free up server capacity for use by buckets with larger reservations. The effect can be seen clearly in the figure, where during the last two intervals, the servers are mainly processing requests of the buckets with smaller indexes (*i.e.* those with higher reservations).

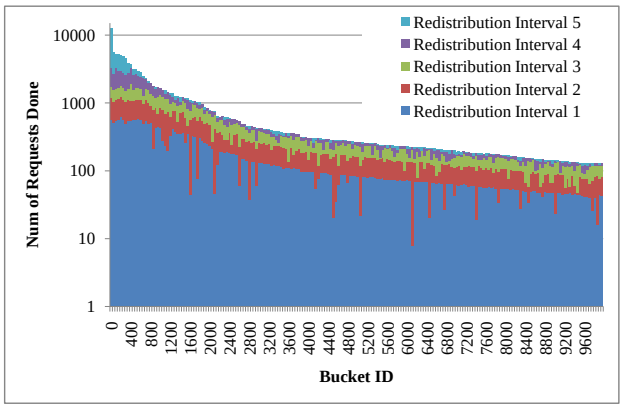
Figure 3(c) also shows the adaptivity of the algorithm to sharp demand changes. For instance, several red needles



(a) The *Zipf distribution* of buckets' reservation requirements with the exponent factor $s = 0.5$, sorted from high to low. Each bucket is assigned a weight $w_i = 1/i^{0.5}$ from the set of weights $\{w_j : j = 1, \cdots |B|\}$ with probability $(1/i^{0.5})/(\sum_i 1/i^{0.5})$. Then the aggregate capacity of the servers is distributed to the buckets in proportional to their weights as their reservations. The bucket reservations are roughly in the range of $(120, 13000)$, and the y-axis has a logarithmic scale.



(b) The time that demand changes of the buckets.



(c) The number of requests completed for each bucket in the 5 redistribution intervals. Each bucket is active (having non-zero demand) on a set of 8 servers.

Fig. 3: The result of the simulator-based QoS evaluation: reservations, demand change times, and IOs for a sample of the buckets. The figures show the results for every 50^{th} bucket.

(representing IOs in the 2nd interval) can be seen in the blue (1st interval) region. This means that these buckets received less service than their peers in the first interval. This happens because of a sudden drop in the demands of this bucket at some servers and an increase in others because of locality changes. Servers with reduced demand will not be able to consume their reservation tokens in this interval. The unused capacity however is not wasted and will be used by buckets which have both demand and tokens on the server. Even if all tokens with demand at the server have been consumed, the additional capacity is used to serve requests without tokens. These opportunistic requests will still be counted towards the reservation requirements of the corresponding bucket, and the controller will correct for these additional IOs by reducing its target remaining reservation. In the next interval, the coordinator will allocate additional tokens to the buckets that were underserved in the first interval and direct them to the new server, and reduce the total number of tokens to buckets which received opportunistic IOs. For instance, bucket 6100 that has its first demand change early in the 1st redistribution interval, receives less service in the first interval but catches up by the end of the second one.

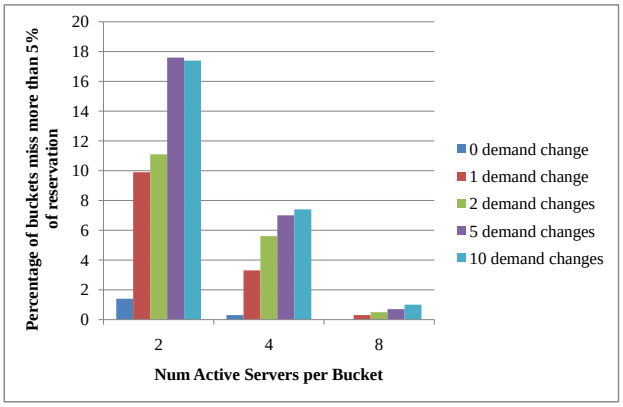


Fig. 4: The average error of *pShift* with different number of demand changes and different number of active servers of each bucket.

Experiment 2: In this experiment we study the effect of two parameters on QoS accuracy: the number of servers on which the buckets are active (N_A) and the number of times the demand of a buckets changes in a QoS period (N_D). The accuracy measure is the fraction of buckets that miss 5% or more of their reservation in the QoS period.

Inaccuracy in the reservations achieved may arise due to *intrinsic error*. The intrinsic error arises because for certain data distributions it is *impossible* to meet the reservations irrespective of the scheduling strategy. For example, consider a situation where two buckets with reservations of 100 IOPS each have high demand on a single server and no demand on any of the other servers. If the capacity of the server is 100 IOPS then clearly at most one of the buckets can meet its reservation, and the intrinsic error is greater than 0. Intrinsic errors are manifested in *pShift* as weak excess tokens on some overloaded server(s) that cannot be moved to any underloaded server because of lack of demand for the buckets on the latter. When there is no runtime demand change, *pShift* guarantees that all reservations will be met whenever such an allocation is possible. Hence, any error in this situation is an intrinsic error that cannot be avoided.

Figure 4 also shows the average measured error for $N_A = \{2, 4, 8\}$ and $N_D = \{0, 1, 2, 5, 10\}$. Each bar is the average of 5 runs; the variation was less than 10%. We found the error is almost always 0 for $N_A > 8$ and so are not reported. The bar for $N_D = 0$ represents intrinsic errors; in this case the optimal allocation determined by *pShift* is still insufficient to meet all reservations. For a given number of active servers, the error grows initially as the number of demand changes increases, but levels off and becomes insensitive to additional demand changes. This is because demand changes that occur within a reallocation interval tend to average out or in any case have no worse an effect than one large demand change early in the interval. On the other hand, as the demands of a bucket get spread out over several servers, albeit in a skewed Zipf-like distribution, the error decreases. Note the errors also include intrinsic error, which no scheduling algorithm can avoid. It can be seen that a higher N_A reduces the intrinsic error. Characterization and bounds on intrinsic errors are deferred to future work.

C. Parallelization Evaluation

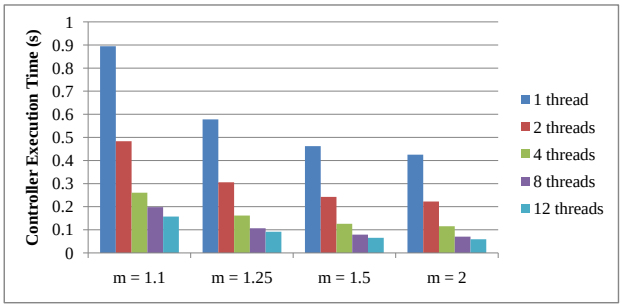
In this section, we show the speedup of the parallelization optimization for the *pShift* algorithm. We used the **parallel for** primitive in *OpenMP* to parallelize the two hot-spot regions discussed in Section III-C1.

We use 64 servers and $10,000$ buckets with the same *Zipf* demand and active server specification as in Section IV-B. Each server's throughput in the QoS period is $100,000$ and each bucket is active on 8 servers. However, we vary the following two variables:

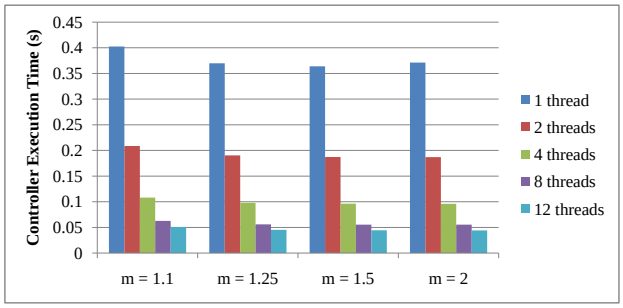
- r : the fraction of the total cluster capacity being reserved, *i.e.* $(\sum_{i \in \mathbf{B}} R_i)/(\sum_{j \in \mathbf{S}} C^j)$. Note that $0 \leq r \leq 1$.
- m : the ratio of the total demand of each bucket to its reservation, *i.e.* D_i/R_i. Note that $m \geq 1$.

During the experiments, we found that the execution time of *pShift* increases with higher r and smaller m. If m is small there will be less spare demands at a server reducing the number of tokens that can be moved along an edge. Similarly, if r is high, servers will overload more easily and underloaded servers will have less spare capacity to accept tokens. The execution times of *pShift* with $r = 1.0$ and $r = 0.9$, with different number of working threads, and different m values are shown in Figure 5(a) and Figure 5(b). From the figures we can see by using 12 threads, we can achieve up to $5\times$ speedup and absolute runtimes in tens of milliseconds.

As a comparison, for the problem size of this scale, *bQueue* takes 2-3 minutes to complete, an overhead that renders it unusable for the QoS period of the order of seconds. Figure 7 shows the comparison of the controller's execution time of *pShift* and *bQueue* with a smaller problem size of 100 to 1000

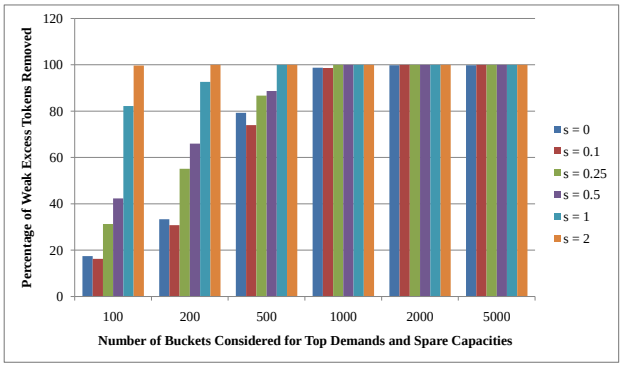


(a) Variation of the execution time of *pShift* with m for $r = 1.0$.

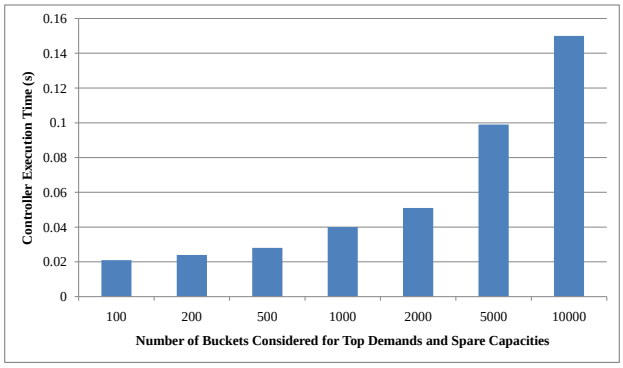


(b) Variation of the execution time of *pShift* with m for $r = 0.9$.

Fig. 5: Execution time of *pShift* with parallel threads.



(a) Percentage of weak excess tokens removed with different M.



(b) Execution time with 12 threads and different M.

Fig. 6: Error and execution time with approximation.

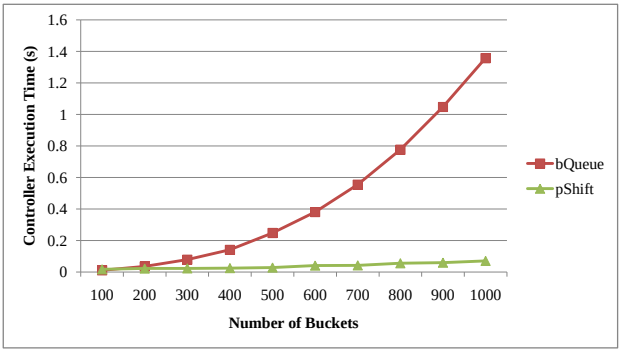


Fig. 7: The controller execution time of *pShift* comparing against *bQueue*, with 100 to 1000 buckets, 64 servers, $r = 1.0$ and $m = 1.1$.

buckets while keeping the number of servers to be 64, and $r = 1.0$, $m = 1.1$. From the figure, we can see with a fixed number of servers, *bQueue*'s execution time grows quadratically with the number of buckets, while *pShift* only grows linearly. This is the main reason that *pShift* runs faster than *bQueue*. Additional discussions are provided in Section V.

D. Approximation Evaluation

In this section, we evaluate the efficiency and accuracy of the approximation approach. To make a comparison, we maintain the configuration of Section IV-C. We choose $r = 1.0$ and $m = 1.1$ since this is the stress case that takes most execution time, and use 12 threads as well. Before beginning to shift the tokens we filter the data and retain only the top M buckets with most tokens and the top M buckets with the highest spare demands. We choose $M = \{100, 200, 500, 1000, 2000, 5000\}$. Moreover, we tried different reservation distributions by using different exponent factors s in generating the *Zipf* distribution. We tried $s = \{0, 0.1, 0.25, 0.5, 1, 2\}$, where a smaller s implies less variation in the reservations of different buckets. In particular, $s = 0$ denotes a uniform distribution.

Recall that the goal of the shift steps is to reduce the number of weak excess tokens. Figure 6(a) shows the percentage of weak excess tokens removed by using different M and s. Form the figure, we can see that though there are 10000 buckets, considering only the top 2000 (20%) buckets is enough to remove all the weak excess tokens, even with the uniform reservation distribution. Moreover, we can still remove more than 70% weak excess tokens using only the top 500 (5%) buckets. One can also see that for a fixed M we remove more weak excess tokens when the skew in the reservation distribution increases, reaching 100% success with just $1-5\%$

of the buckets. Since skewed distributions are more likely in practice, the approximation will be especially useful in these cases.

The benefit of the approximation approach is that it can further accelerate the execution time of the *pShift* algorithm. Figure 6(b) shows the running time of the *pShift* algorithm with different M. It can be seen that we can achieve another $5\times$ speedup on top of the parallelization approach while still keeping the accuracy at a reasonable threshold.

E. Linux Evaluation

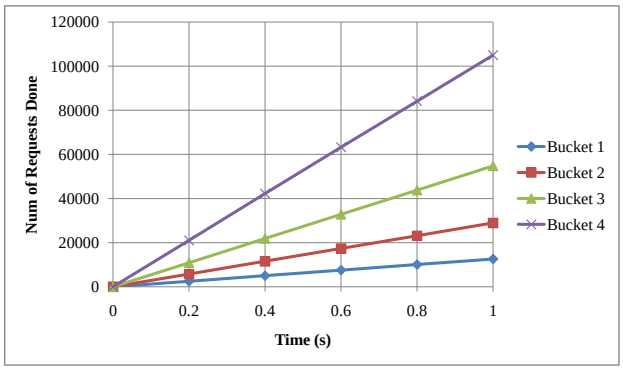
In this section, we describe results on the Linux cluster in three experiments. Experiment 3 uses a small configuration with static demands to show how *pShift* meets reservations and limits. We then consider dynamically varying demands in Experiment 4. These experiments are done using memcahced as the backend server. Finally, Experiment 5 shows the results using file IO.

Experiment 3:

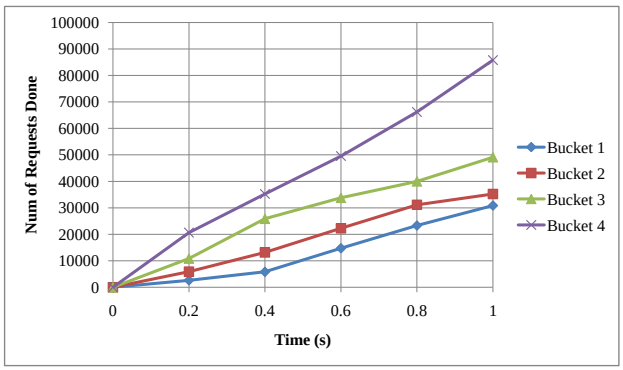
In this experiment, we show how *pShift* handles reservations and limits with a simple and steady configuration of 4 buckets and 4 servers. We initially focus on reservations; each bucket has a reservation of $30,000$ RPS and an essentially unbounded limit ($200,000$ RPS was used in the experiment). Buckets 1, 2, 3, 4 are continuously backlogged on servers $\{1\}$, $\{1,2\}$, $\{1,2,3\}$ and $\{1,2,3,4\}$ respectively with 5 outstanding requests. We choose a QoS period of 1 second. For the token allocation, we redistribute tokens every 200ms *i.e.* we have 5 token redistributions in each QoS period. Theoretically, if no QoS controls are applied, the round-robin scheduler will give buckets 1, 2, 3, 4 throughputs in the ratio of $(1/4) : (1/4+1/3) : (1/4+1/3+1/2) : (1/4+1/3+1/2+1) = 3 : 7 : 13 : 25$, which results in average throughputs of 12500, 29166, 54166, and 104166 RPS, respectively. On the other hand, when QoS controls are applied, the reservations of all buckets are expected to be met.

Figures 8(a) and 8(b) show the results of the execution, which matches our theoretical analysis. Figure 8(a) shows the throughput of the buckets without QoS controls and the results match the predicted throughputs closely. In Figure 8(b) the throughputs with reservation controls are shown. Bucket 1's throughput, which was well below its reservation, now increases to match the required $30,000$ RPS. Looking at Figure 8(b) we can see that bucket 4 reaches its reservation first (at roughly 400ms) since it gets service on all servers. At that point it loses priority in scheduling and the other buckets get increased service; this can be seen most dramatically by the change in the slope of bucket 1 at that time.

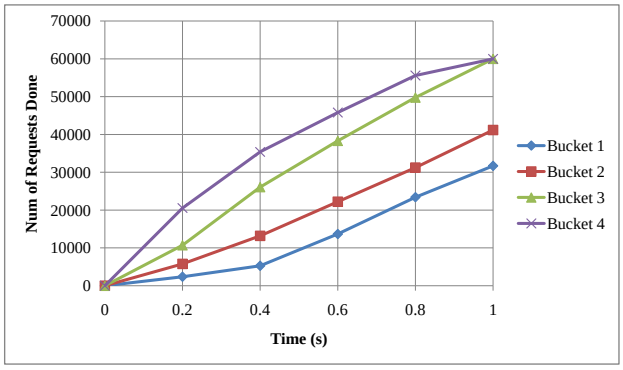
Finally we set the limit of each bucket to be $60,000$ RPS. Figure 8(c) shows the execution results. We can see that both reservations and limits are met by all the buckets. Figure 8(c) looks similar to Figure 8(b) in the interval 0 to 0.6s, where the servers are basically doing reservation requests. Beyond this time bucket 4 gradually slows down as it meets its limit threshold in each sub-interval, yielding to buckets that are further away from their limit.



(a) The number of requests being completed with simple round-robin scheduler.



(b) The number of requests being completed with *pShift* scheduler with only reservations.



(c) The number of requests being completed with *pShift* scheduler with reservations and limits.

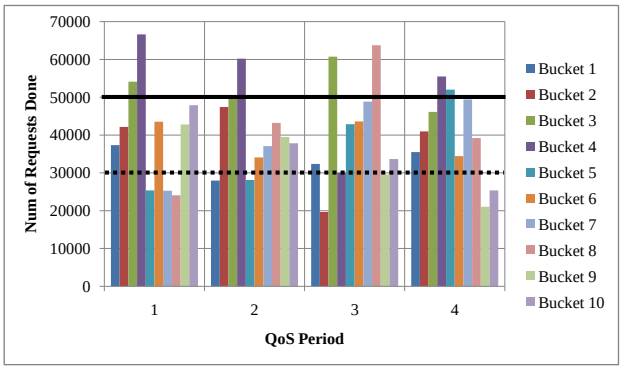
Fig. 8: The number of requests done for *pShift* and simple round robin schedulers in Experiment 3.

Experiment 4:

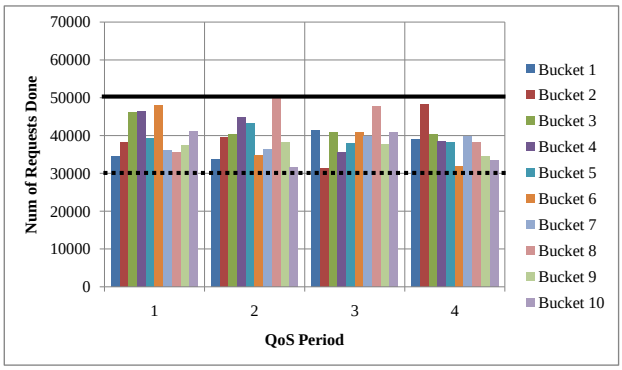
We now show how *pShift* guarantees QoS on the cluster when demands change dynamically. We use a larger configuration of 8 servers and 10 buckets. Each bucket has a reservation of $30,000$ RPS and a limit of $50,000$ RPS. Each bucket randomly chooses a number between 2 to 6 active servers; at every demand change instant, a fresh set of active servers (between 2 and 6) is chosen. We allow 5 demand changes for each bucket. We run 4 consecutive QoS periods with 5 redistributions in each period.

Figures 9(a) and 9(b) show the dynamic throughput for both scenarios. From the figures, we can see with the simple round robin scheduler, the QoS is not guaranteed, as some buckets (*e.g.* buckets $5, 7$ and 8) in QoS period 1 did not meet their reservations, and some buckets (*e.g.* buckets 3 and 4) received throughput exceeding their limits. In contrast, by using the *pShift* scheduler, both reservation and limit QoS was guaranteed in all intervals for all buckets.

Finally, we tested a bigger problem size using 200 buckets. On average without QoS controls, 68.5% of the buckets met their reservations and 83.5% did not go beyond their limits. In contrast, when using *pShift*, 99.6% of the buckets met their reservations and none exceeded their limits. By running the QoS controller for 15 minutes, we found on average, the communication in every 1s QoS period takes 9ms, *i.e.* the average communication overhead is around 0.9%.



(a) The number of requests being completed with simple round-robin scheduler.

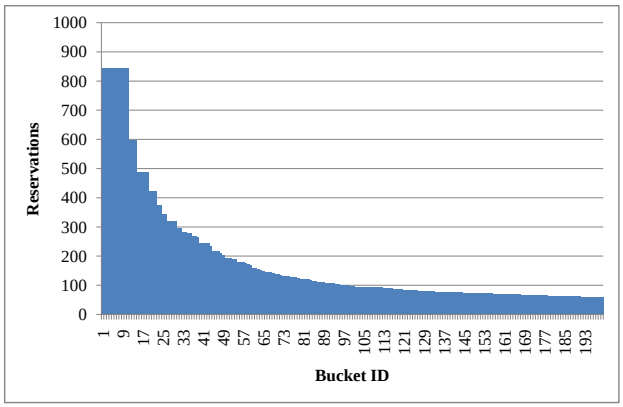


(b) The number of requests being completed with *pShift* scheduler.

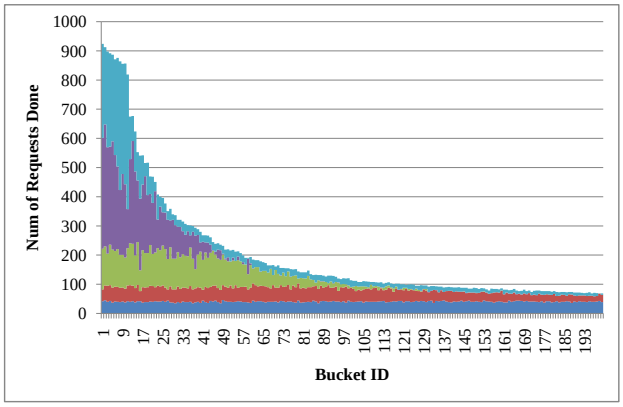
Fig. 9: Total number of request completed for *pShift* and simple round robin scheduler in Experiment 4.

Experiment 5:

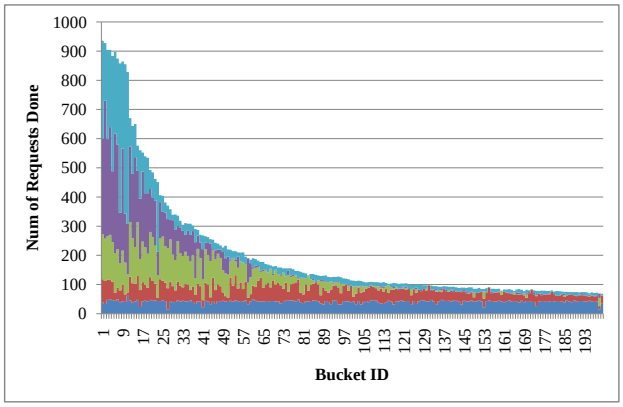
Finally, we show how our storage prototype with *pShift* guarantees reservation and limit QoS when doing block file IO. In this experiment, we have 8 servers and 200 buckets, and 80% of the total server capacities are being reserved, *i.e.* $r = (\sum_{i \in \mathbf{B}} R_i)/(\sum_{j \in \mathbf{S}} C^j) = 0.8$. The bucket reservations (shown in Figure 10(a)) and demands follow the same *Zipf distribution* that we used in the simulation, and each bucket allows 2 demand changes in the QoS period as before. Here we also assign each bucket a *limit* equal to 1.5 times its reservation.



(a) The *Zipf distribution* of buckets' reservation requirements.



(b) The number of requests being completed in the first QoS period.



(c) The number of requests being completed in the second QoS period.

Fig. 10: The reservation distribution and the number of requests being completed in Linux QoS evaluation.

We set the QoS periods to be 5 seconds, and in each QoS period we do 5 token allocations, *i.e.* each redistribution interval is 1 second. We run the server for two QoS periods (*i.e.* 10 seconds) and record the QoS result as the number of requests being completed in each redistribution interval. Servers continue to serve IOs as the controller is computing new token allocations.

Figure 10(b) and Figure 10(c) shows the number of requests being done in both QoS periods. From the figures, we can see *pShift* provides the reservation and limit QoS in a reasonable manner. The needles, similar as those shown in the simulation, indicate how *pShift* handles demand changes. Quantitatively, only one bucket (0.5%) missed its reservation by more than 1%, and no bucket exceeded its limit.

V. Discussion

In this section, we compare *pShift* with the related existing approaches and show the advantages of *pShift*. We also discuss two common design issues in practice and give our suggested solutions.

Coarse-grained Approaches: The token allocation problem has been modeled previously using max-flow and linear programming [22]. Both models have the same goal as *pShift*: maximizing the number of reservations that can be met. However, *pShift* has a much smaller overhead, both in actual runtime and in theoretical complexity. *pShift* achieves its scalability and speed by modeling the token allocation problem as a two-level problem. It first works on the token transfer graph which only includes servers (whose number of vertices is $O(|\mathbf{S}|)$) to find out the shift path, and then use the token-movement map to figure out which buckets to move their tokens. In contrast, both max-flow and linear programming are working on graphs or inequality constraints with per-server, per-bucket information simultaneously, which have a very much higher complexity of $O(|\mathbf{S}||\mathbf{B}|)$.

Fine-grained Approaches: In principle one can extend the fine-grained distributed client QoS algorithms based on tags [17], [19] to provide bucket QoS by forwarding each request to a dedicated bucket controller, which then tags the requests so that it can be scheduled properly. However, forwarding each request through an additional server increases request latencies. Furthermore, the tradeoff for achieving accurate QoS at a fine granularity is to limit the number of requests the controller can keep outstanding at the servers, so that tags accurately reflect IO completion rates. This can potentially limit the system throughput as well.

Local Priorities: In many situations, buckets may have different priorities, which could be quantified by bucket weights. In such cases, our approach is to incorporate the priorities at a local level in each server during scheduling. Therefore we adopt the round-robin scheduler to act like a *weighted round-robin scheduler*. The scheduler continues to prioritize requests with reservation tokens over any requests not backed by a token. Within this umbrella, it gives priority to the requests in proportion to their weights. The reservation guarantees are unaffected by this change but high priority buckets would tend to receive lower latency for their requests.

Demand and Capacity Estimation: In practice, the demand of most buckets may not change too frequently. In such cases, instead of having evenly divided redistribution periods, we can use a dynamic sliding window to estimate the demand changes, and trigger token reallocation only if demands change beyond a threshold. Similar approaches can be applied for capacity estimation. By using dynamic estimation, we can reduce the communication overhead as well. The detailed studies are left for future work.

VI. Conclusions

In this paper, we present *pShift*, a scalable token allocation algorithm for distributed storage clusters. *pShift* uses a novel token shifting approach to handle the resource balancing between different servers. *pShift* has a smaller runtime overhead than existing approaches and can be directly integrated to coarse-grained token-based QoS frameworks. Furthermore, *pShift* can be further accelerated using parallelization and approximation. Our performance results show *pShift* is able to provide distributed bucket QoS on a large scale with good accuracy, and is robust to several dynamic workload behaviors. In future work, we will focus on studying the effect of using dynamic redistribution intervals, as well as distributing the token control algorithm to further improve the scalability.

Acknowledgment

The authors would like to thank Huawei for supporting this work under the Huawei Innovation Research Program.

References

[1] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[2] Bansal Sakshi. Glusterfs : A dependable distributed file system. http://opensourceforu.com/2017/01/glusterfs-a-dependable-distributed-file-system/, 2017.

[3] Mayur R Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64. ACM, 2008.

[4] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. In *ACM SIGARCH Computer Architecture News*, volume 32, pages 48–58. ACM, 2004.

[5] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. Kudu: storage for fast analytics on fast data. *Cloudera, inc*, 28, 2015.

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[7] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[8] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[9] VMWare. vsan hyper-converged infrastructure software. https://www.vmware.com/products/vsan.html.

[10] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[11] Christopher R Lumb, Arif Merchant, and Guillermo A Alvarez. Façade: Virtual storage devices with performance guarantees. In *FAST*, volume 3, pages 131–144, 2003.

[12] Lan Huang, Gang Peng, and Tzi-cker Chiueh. Multi-dimensional storage virtualization. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 14–24. ACM, 2004.

[13] Joel C Wu and Scott A Brandt. Qos support in object-based storage devices. In *Proceedings of the 3rd international workshop on storage network architecture and parallel I/Os (SNAPI05)*, volume 329, pages 41–48, 2005.

[14] Theodore M Wong, Richard A Golding, Caixue Lin, and Ralph A Becker-Szendy. Zygaria: Storage performance as a managed resource. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 125–134. IEEE, 2006.

[15] Ajay Gulati, Arif Merchant, and Peter Varman. d-clock: Distributed qos in heterogeneous resource environments. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 330–331. ACM, 2007.

[16] Anna Povzner, Darren Sawyer, and Scott Brandt. Horizon: efficient deadline-driven disk i/o management for distributed storage systems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 1–12. ACM, 2010.

[17] Ajay Gulati, Arif Merchant, and Peter J Varman. mclock: handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 437–450. USENIX Association, 2010.

[18] Ajay Gulati, Ganesha Shanmuganathan, Xuechen Zhang, and Peter J Varman. Demand based hierarchical qos using storage resource pools. In *USENIX Annual Technical Conference*, pages 1–13, 2012.

[19] Yin Wang and Arif Merchant. Proportional-share scheduling for distributed storage systems. In *FAST*, volume 7, pages 4–4, 2007.

[20] Code that implements the dmclock distributed quality of service algorithm. https://github.com/ceph/dmclock.

[21] Eom Jugwan, Kim Taewoong, and Park Byungsu. Implementing distributed mclock in ceph. https://www.slideshare.net/ssusercee823/implementing-distributed-mclock-in-ceph.

[22] Yuhan Peng and Peter Varman. bqueue: A coarse-grained bucket qos scheduler. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 93–102. IEEE, 2018.

[23] Appendix: Proof of the optimality of pshift algorithm. https://sites.google.com/site/pyh119/MSST2019_Appendix.pdf.

[24] Intel. Intel xeon processor e5-2640 v4 (25m cache, 2.40 ghz) product specifications. https://ark.intel.com/products/92984/Intel-Xeon-Processor-E5-2640-v4-25M-Cache-2_40-GHz.

[25] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[26] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[27] Ycsb core workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.