Fair-EDF: A Latency Fairness Framework for Shared Storage Systems

Yuhan Peng
Rice University

Peter Varman
Rice University

Abstract

We present Fair-EDF, a framework for latency guarantees in shared storage servers. It provides fairness control while supporting latency guarantees. Fair-EDF extends the pure earliest deadline first (EDF) scheduler by adding a controller to shape the workloads. Under overload it selects a minimal number of requests to drop and to choose the dropped requests in a fair manner. The evaluation results show Fair-EDF provides steady fairness control among a set of clients with different runtime behaviors.

1 Introduction

Modern 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 high-throughput storage for huge data sets. When deployed in a datacenter these systems are shared among multiple clients, each representing tens to hundreds of users. Clients require predictable performance typically codified in SLOs such as guaranteed throughput over a specified duration or a maximum response time for a specified fraction of its requests.

In order to make response time QoS guarantees, two-sided SLOs are employed; the client receives the agreed-upon service provided its input meets specified constraints on its arrival rates and the sizes and frequency of its bursts. Admission control is used to limit the clients in the system to a sustainable set, regulators police client traffic for compliance with input SLOs, and schedulers order the requests in a globally expedient manner to meet guarantees. Moreover, in clustered storage, requests belonging to a client are directed to different storage servers based on the placement and replication policies of the system. Clients are typically unaware of the data distribution, making it hard to predict the dynamic runtime demands on any server. Hence, the traffic seen by an individual server is highly dynamic and difficult to control. While SLO policing may be used to regulate the *aggregate* client traffic characteristics, it is unreasonable (and impractical) to require these at the individual server level.

This motivates the problem addressed in this paper: how to provide reasonable differential response time guarantees for clients sharing a server when the traffic cannot be predicted or controlled at ingress to the server? The problem of guaranteeing latencies in storage systems has been extensively studied over the years. Many solutions like [10–20] present techniques to meet explicitly-specified QoS latency bounds, using a combination of workload shaping and scheduling algorithms. Others [21–27] present different system techniques to reduce average or tail latencies. More discussion is presented in Section 2.1.

A recent paper, MittOS [28], describes a novel driver-level model and implementation that predicts whether an arriving request can meet its deadline, and drops (or redirects) the arriving request if it cannot. In our work, we propose an orthogonal dimension to this solution: we describe an optimal algorithm for deciding when to discard a request and which request to drop based on the QoS requirements. We introduce *Fair-EDF*, a framework providing fairness between the clients in a shared storage server when guaranteeing their latencies. Fair-EDF extends the *earliest deadline first* (EDF) [29] scheduling policy by adding a *controller* that selects and drops requests in the scheduling queue when it detects that latency violations will occur. Fair-EDF assumes an OS such as MittOS is already in place, which provides system support for dropped requests.

Figure 1 shows an overview of the system. The controller is optimal in the sense that the total number of requests it drops is the minimum possible, and provides fairness control among the clients such that the proportion of requests being dropped for each client are equalized. Different fairness criteria can be substituted in the controller without changing the framework. A standard EDF scheduler is used to dispatch requests in deadline order. In Fair-EDF, all requests dispatched by the scheduler will meet their deadlines. Fair-EDF extends the idea of the offline *RT-OPT* [30] framework to work in an online situation and add fairness. In practice, Fair-EDF would be especially useful in streaming applications, such as streaming video from object storage.

The remainder of the paper is organized as follows. Sec-

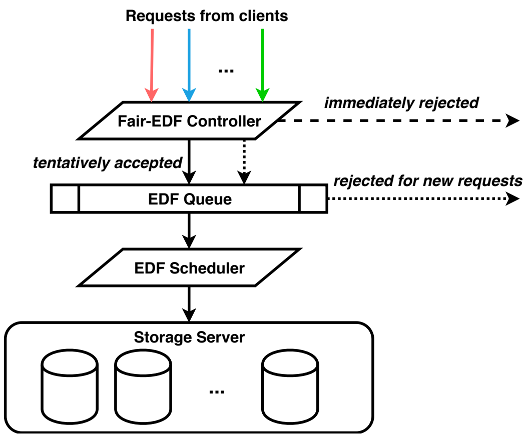


Figure 1: An overview of the Fair-EDF framework.

tion 2 gives additional background on latency QoS and discusses related works. In Section 3 we propose a framework for guaranteeing latencies in a distributed cluster. Section 4 shows the preliminary evaluation results of our proposed algorithm. Finally, Section 5 discusses the related issues and challenges of our design.

2 Overview

The storage server is shared by a number of clients that send I/O requests to the server. All client requests are assumed to be chunked into fixed-size I/O requests that have a fixed service time σ. A request that arrives at time t_r is assigned a *deadline* d_r equal to the sum of its arrival time and latency bound. A request is *successful* if it completes before its deadline, and is *wasted* if it is dropped (not scheduled) or completes after its deadline. Each request can have its own latency bound based on some classification. We assume that $t_r + \sigma \leq d_r$ is always true, *i.e.* a request will meet its deadline if there are no queueing delays and it is scheduled immediately upon arrival.

Example 1: We motivate the latency fairness framework with the following example. Suppose a server of 100 IOPS capacity is shared by two clients A and B, which send uniform-sized I/O requests with a service time of 10ms each. At time 0, A sends a burst of 100 requests with deadlines all at 1s. Client B sends I/O requests at a uniform rate every 10ms apart and all request have a latency bound of 40ms, *i.e.* B sends requests at times $0, 10, 20, \ldots$, with deadlines of $40, 50, 60, \ldots$ respectively.

There are a total of 200 combined I/O requests in one second of operation, which is double the server IOPS capacity. An EDF [29] scheduler, which is commonly used for supporting latency requirements, will do almost all of B's requests first, followed by those of A. Most requests of B will meet their deadlines and almost no requests of A will do so, resulting in poor fairness. Furthermore, all new requests arriving in the next second will also miss their deadlines.

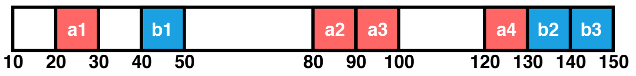
2.1 Related Work

For latency guarantees, most of the existing QoS frameworks, such as [11, 13, 14, 20, 24, 26], compute scheduling metadata (usually referred as *tags*) for each request. The tags are used for the request scheduler to determine the order of scheduling. Usually, a sophisticated scheduling algorithm is required in these frameworks, which can affect their scalability. Another set of approaches, such as [10, 16, 17], use *dynamic token buckets* which guarantee the latency by controlling the rate at which requests are dispatched to the schedulers. Some frameworks [13,15,20] also supports the bandwidth allocation or proportional sharing at the same time. Existing works use a combination of input SLOs, admission control, and traffic policing, either explicitly or implicitly, to guarantee response times if the load exceeds server capacity. The controller either drops the overflow requests or degrades their priorities. An offline algorithm, *RT-OPT* [30], was developed to guarantee timely playback of VBR video with the minimum number of dropped frames. Fair-EDF extends the idea of RT-OPT to an online algorithm, and also provides fairness guarantees.

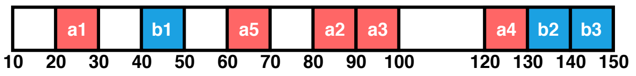
Existing solutions do a good job of meeting latency guarantees when the client traffic can be predicted or regulated to meet server capacity limits. Some solutions [11] delay requests from clients that violate their SLO in order to keep the aggregate request rate below system capacity; this isolates well-behaved workloads from misbehaving ones, but can lead to a large increase of the average response time for even small violations. Other solutions use a token-bucket regulator and drop requests that violate their SLO [10], while others [16] reschedule overflowing requests in secondary queues with relaxed deadlines, or dispatch them to replica servers [6, 22, 27]. In contrast to these frameworks that require per-client arrival-rate parameters to *avoid* server overload, we dynamically *detect* overload and shape the workloads accordingly.

3 Fair-EDF Framework

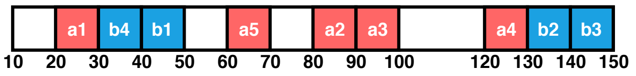
Each incoming request is tagged with its deadline indicating the time by which it must complete service. When it arrives at the server, the request may be immediately *rejected* or tentatively *accepted* for scheduling. A rejected request may be discarded, re-directed to an alternate server, or moved to a secondary queue for best-effort scheduling. An accepted request will be enqueued and wait for its turn to be dispatched; during its wait, the request may still be rejected as new requests arrive. The *client success ratio*, f_i, is the fraction of client i's requests that succeed, and $(1 - f_i)$ is called the *client miss ratio*. The *system success ratio* F is the fraction of the total number of requests that succeed. The goal of our framework is to shape the f_i subject to maximizing F. In this paper we choose a *min-max* criterion: that is, among all schedules that maximize F we choose one that simultaneously minimizes the highest client miss ratio. Other QoS criteria can be



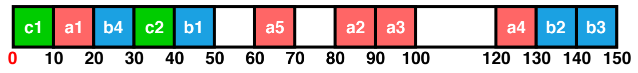
(a) The initial occupancy chart in Example 2.



(b) The occupancy chart in Example 3 after accepting a_5.



(c) The occupancy chart in Example 3 after accepting b_4.



(d) The occupancy chart in Example 4 after accepting c_1 and c_2.

Figure 2: An example of the occupancy chart in Fair-EDF.

implemented within our framework as discussed in Section 5.

We now discuss the idea behind the algorithm followed by the details of the implementation. Let d_i denoted the deadline of request i and σ its service time. The current set of accepted requests are conceptually placed on a timeline called an *occupancy chart*. Request i occupies the interval $[t_i,\ t_i+\sigma]$ on the timeline where t_i is the latest time it can begin execution while ensuring that it and all requests with deadlines later than d_i are successful. The occupancy chart is naturally partitioned into alternating *busy* and *idle* segments as discussed below. As long as no interval crosses the current time T_{now}, all requests will meet their deadline when scheduled in EDF order.

Example 2: Consider two clients A and B. Each request has a service time of 10ms. Client A has 4 requests a_1, a_2, a_3, a_4 with deadlines 30, 100, 100, and 140ms respectively, and Client B has 3 requests b_1, b_2, b_3 with deadlines 50, 145 and 150ms. Suppose the current time is 10ms. Figure 2a shows the occupancy chart. Request b_3 occupies the interval $[140, 150]$; b_2 cannot be scheduled at 135 as it would conflict with b_3, and occupies the interval $[130, 140]$. Similarly a_4 must occupy the interval $[120, 130]$, resulting in the busy segment $[120, 150]$ comprising requests $\{a_4, b_2, b_3\}$. In the same way $\{a_2, a_3\}$ comprises busy segment $[80, 100]$, $\{a_1\}$ forms segment $[20, 30]$, and $\{b_1\}$ forms segment $[40, 50]$.

Example 3: Continuing Example 2, suppose there is a new request a_5 with deadline 70ms. The request can be assigned the interval $[60, 70]$, since it will not conflict with existing requests (see Figure 2b). Next a new request b_4 with deadline 45ms arrives. It will be assigned the interval $[30, 40]$ to avoid conflicting with b_1. The two adjacent segments $[20, 30]$ and $[40, 50]$ will now merge into a single segment $[20, 50]$, as shown in Figure 2c.

3.1 Algorithm

Let $S_1, S_2, \cdots S_n$ denote the current set of busy segments and let S_i span the time range $[L_i, R_i]$ (L_i and R_i are the left and right time boundaries). When a new request r arrives, we must identify the segment its interval will occupy, followed by updating its boundaries. This may cause some segments to now overlap, leading to a cascade of boundary changes and segment merges. If the start time of the earliest (leftmost) segment becomes less than the current time, then the current set of requests cannot all meet their deadlines, and one or more will need to be dropped so that the remaining requests can be successful.

Let d_r denote the deadline of new request r.

Case 1: d_r lies *between* two consecutive segments S_k and S_{k+1} *i.e.* $R_k < d_r < L_{k+1}$. Request r is assigned the interval $[d_r - \sigma, d_r]$. If $d_r - \sigma > R_k$ then create a new segment $[d_r - \sigma, d_r]$ that lies between S_k and S_{k+1}. Otherwise, merge the interval $[d_r - \sigma, d_r]$ with the segment S_k: L_k is reduced by $\sigma - (d_r - R_k)$ (the amount of overlap) and R_k is changed to d_r. Note that changing L_k may cause S_k to overlap segment S_{k-1}, potentially causing a cascade of merges of adjacent segments.

Case 2: d_r lies *within* a segment S_k *i.e.* $L_k \leq d_r \leq R_k$. In this case, L_k is reduced by σ. Once again, reducing L_k may cause Segments S_k and S_{k-1} to overlap, potentially triggering a cascade of merges of adjacent segments.

In general, when a request arrives it is placed on the timeline using case 1 or 2 above. If the first segment now begins at a time less than the current time (this means the left boundary of the first segment S_1 is changed) then a request needs to be dropped. Any request in the first segment, including the new request that caused the potential deadline violation, are candidates for dropping. For our min-max fairness criterion we must choose a request from the candidate set that belongs to a client with the smallest miss ratio. Finally, the request scheduler simply dispatches requests in EDF order, which will guarantee that the current set of requests will meet their deadlines.

Example 4: In Figure 2c, suppose two new requests c_1 and c_2 from client C, with deadlines 25 and 40, arrive at time 10. From case 2 above, c_1 will be assigned the interval $[10, 20]$. Since the current time is 10 this is a feasible schedule in which all requests will meet their deadlines if they are executed in deadline order (their order in the occupancy chart). Next c_2 will be assigned the interval $[30, 40]$, moving the requests b_4, a_1 and c_1 one slot to the left to $[20, 30]$, $[10, 20]$ and $[0, 10]$ respectively (see Figure 2d). After placing c_2, segment 1 starts at time 0, which is smaller than the current time 10, which means that at least one of the four requests c_1, a_1, b_4 or c_2 must miss its deadline. We can choose to reject any one of these requests, and move the other requests on its left to the right to the vacant slot. The choice of which of the requests to drop is governed by the QoS policy.

4 Preliminary Evaluation

4.1 Experimental Setup

In the evaluation, we compare Fair-EDF with a standard EDF scheduler, as well as a variant of EDF we call *Prudent-EDF*. The problem with standard EDF is that a capacity overload will cause the deadlines of future requests, those following the overloaded period, to be missed as well. By contrast, Prudent-EDF drops requests that it recognizes will miss their deadline, thereby preventing them from affecting future request deadlines. Prudent EDF drops a request at the latest possible time and consequently, like the immediate-drop policy, cannot shape the QoS profile by selecting an appropriate victim request to discard. However, like Fair-EDF, it will also drop the minimal number of requests.

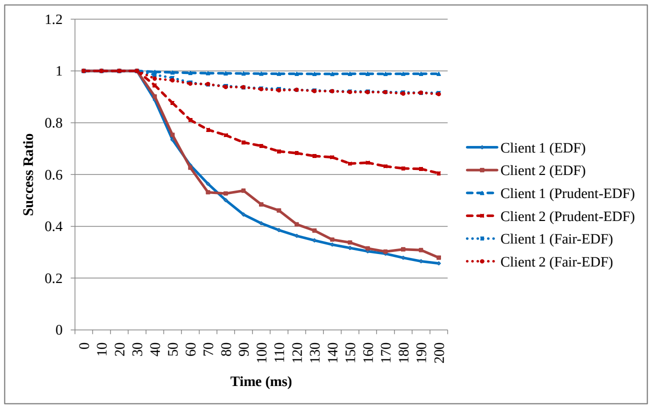
We implemented a prototype of the Fair-EDF framework in *OpenMP*. We use different threads to handle the request controller and request scheduler and keep them pinned on different cores. The arrival pattern and deadlines of the clients are explicitly specified using external input files. The controller generates requests at desired times. For each request, if using Fair-EDF, it uses the algorithm described in Section 3 to place it in the EDF request queue, and if a deadline violation is predicted drops an allowed request based on the QoS policy. With Prudent-EDF, the controller will drop the request whose deadline has already been missed. The scheduler simply dispatches the next request with the earliest deadline.

The workload in our evaluation consists of random 4KB direct reads from a 1GB file. We run our experiments on a standard Linux server. The server node is equipped with an Intel® SSD DC S3700 hard disk [31] and an Intel® Xeon® E5-2697 CPU [32]. We profiled the service time using the standard EDF scheduler and a closed loop client which keeps the server always busy. The average profiled service time for each request is 137μs, *i.e.* the server has an average throughput of around 7300 IOPS.

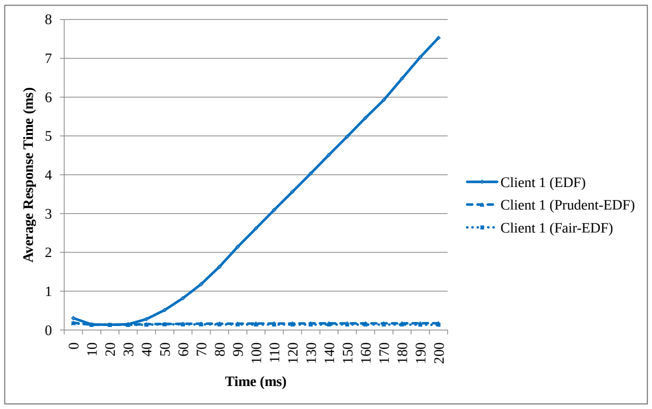
4.2 Evaluation Results

Experiment 1: In this evaluation, we have 2 clients. Client 1 sends one request every 0.15ms, with a latency bound of 0.5ms. Client 2 sends a burst of 15 requests every 10ms, with a latency bound of 25ms for each request. Client 1 has a request rate of 6667 IOPS, and Client 2 a rate of 1500 IOPS. The load on the server is 8167 IOPS which exceeds its capacity of 7300 IOPS.

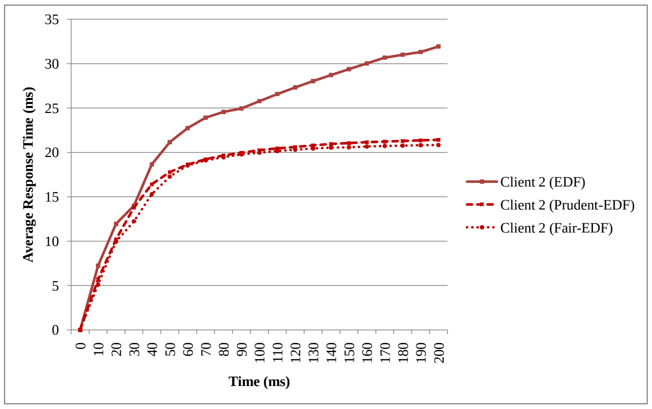
We run the system for a second and show the first 200ms of execution in Figure 3a. It shows the success ratio of both clients using the three policies. Figure 3b and 3c show the average response time for both clients. We can see that EDF gradually misses deadlines of both clients, and the response times start to increase. Due to the chain effect of missed deadlines for standard EDF scheduling under overload, it results in a poor success ratio for both clients. For Prudent-EDF, technically no deadlines will be missed (since they are proactively dropped). However, for this workload it keeps doing requests of client 1 and drops most of the requests of bursty client 2, resulting in poor fairness between clients. Finally, Fair-EDF also guarantees that no deadlines will be missed and will drop the same (minimal) number of requests



(a) The success ratio of both clients using three policies.



(b) The average response time for Client 1 using three policies.



(c) The average response time for Client 2 using three policies.

Figure 3: Evaluation result for Experiment 1.

as Prudent-EDF. However, it tries to fairly distribute the pain of dropped requests and obtains success ratios of around 0.9 for both clients.

Experiment 2: In this experiment, we have 10 clients sharing the server. The arrival patterns and deadline specifications are shown in Table 1. Clients 1 to 8 send requests at different fixed rates, and clients 9, 10 are very bursty. The total load is 8000 IOPS exceeding the server capacity of 7300 IOPS.

Client	Inter-Arrival Time (ms)	Burst Size	Demand (IOPS)	Deadline Time (ms)
1	0.4	1	2500	0.5
2	1	1	1000	1
3	1	1	1000	1
4	2	1	500	2
5	2	1	500	5
6	2	1	500	5
7	2	1	500	10
8	4	2	500	10
9	40	20	500	40
10	50	25	500	50

Table 1: The arrival pattern and deadline specifications of the clients in Experiment 2.

We run the three schedulers for a second, and the success ratio for all clients as well as the system is shown in Figure 4. From the figure, we can see that EDF has poor success ratio for all clients since it gradually misses all deadlines if the capacity is not enough. Both Prudent-EDF and Fair-EDF have a good system success ratio. However, in Prudent-EDF, most requests of bursty clients 9 and 10 are dropped, making their success ratios below the other clients and the system's ratio. In contrast, by using Fair-EDF, the success ratios of clients 9 and 10 go up to roughly equal to the system success ratio, while the success ratios of other clients do not decrease significantly, resulting in better fairness. Moreover, Fair-EDF achieves a similar system throughput as Prudent-EDF. This indicates Fair-EDF does not introduce much runtime overhead comparing with Prudent-EDF.

Experiment 3: Finally, we show the performance of Fair-EDF with a different QoS policy. The specifications of the storage system and the clients remain the same as Experiment 2. However, instead of using the *min-max* criterion, we group the clients into different groups with different priorities. In this experiment, we have two groups *gold* and *silver*. The *gold* group has a desired success ratio of 0.9 and the *silver* group has a desired success ratio of 0.8. The QoS policy for all clients becomes equalizing the relative success ratio divided by the desired success ratio of the group.

We show the result of two grouping policies. Policy 1 groups clients 1 to 8 as *gold* and the rest as *silver*, and policy 2 does the opposite, *i.e.* grouping clients 1 to 8 as *silver* and the rest as *gold*. Figure 5 shows the evaluation results of both

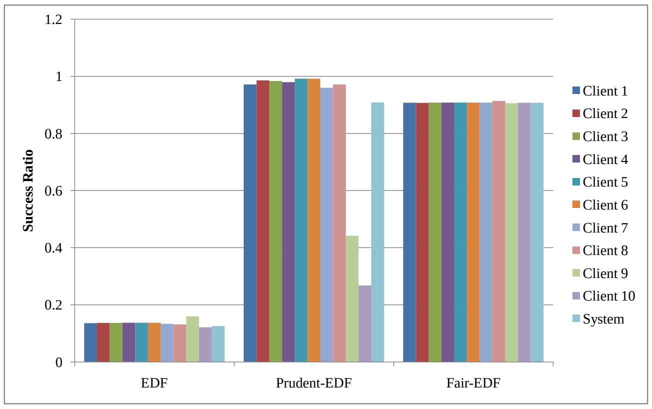


Figure 4: Evaluation result for Experiment 2.

policies, where the desired success ratios of the *gold* and *silver* groups are marked as dotted and solid lines respectively, respectively. From the figure, we can see Fair-EDF provides reasonable QoS for clients with different grouping policies. The framework can be easily modified to support different QoS policies by only changing the controller, and the details are being studied in our ongoing work.

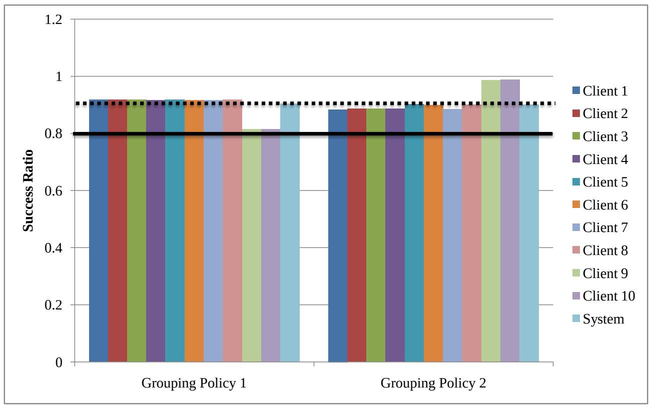


Figure 5: Evaluation result for Experiment 3.

5 Discussion

In this section, we discuss some issues and challenges for Fair-EDF.

QoS policy: Currently, we are mainly using a min-max QoS policy for fairness so all clients have the same miss ratio. In our ongoing work, we are studying supporting other policies that can differentiate between clients. For instance, each client can specify its own SLO as a bound on drop percentage, or the drop rate can be simply weighted to reflect client priorities. We also plan to guarantee the fairness in coarse-grained QoS periods where we reset the counters at the beginning of each period as in our previous work [33].

Request cost estimation: An orthogonal issue to our framework is estimating the cost of the requests. MittOS [28] studies predicting the requests costs for better handling requests latencies. In our future work, we will incorporate this framework or other related request cost estimations with Fair-EDF.

Variable request service times: Even with known service times, in practice, the service times may vary among different requests. The current algorithm will need to be generalized to handle this situation. For instance, in the substitution, one big request may cause several small requests to be dropped, and one substitution may involve multiple clients. Thus, there are more choices in the substitution, which potentially introduces extra runtime overhead. Furthermore, with variable cost requests, we may need a different policy for fairness. For instance, we may want to give different weights to requests of different costs, otherwise requests of higher costs are more likely to be dropped. In the ongoing work, we are extending Fair-EDF to efficiently support requests with a known upper bound of the service times. We are also introducing a work stealing mechanism for the dropped request as best-effort scheduling. The difference between the actual service time and the upper bound provides opportunities to do the dropped requests.

Scalability: We need to worry about scheduler execution time as the number of clients and requests increase and the storage system processes requests faster. We need better data structures as well as implementations that use more efficient synchronization and timer management. In ongoing work, we will examine implementations using better data structures such as priority queues or balanced binary search trees (e.g. AVL tree and RB tree) which handle insertion, modification and deletion in worst case logarithm time. In addition, efficient data structures for QoS selection on subsets of the requests need to be designed.

Distributed fairness: The original motivation for the work was the difficulty of shaping the workload at individual servers of a clustered storage system. In ongoing work, we want to combine the individual server schedulers with a global QoS policy where the performance of a client in aggregate across all servers is optimized. Our previous work [33] studies handling skewed demand distributions and runtime demand changes in QoS. We are planning to extend the idea to Fair-EDF in distributed scenarios.

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. *Retrieved June from http://getkudu. io/kudu. pdf. Pages,, and*, 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 *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[9] VMWare. vSAN Hyper-Converged Infrastructure Software. https://www.vmware.com/products/vsan.html.

[10] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage performance virtualization via throughput and latency control. *ACM Transactions on Storage (TOS)*, 2(3):283–308, 2006.

[11] Ajay Gulati, Arif Merchant, and Peter J Varman. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 13–24. ACM, 2007.

[12] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: quality-of-service in large disk arrays. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 245–254. ACM, 2011.

[13] Xiao Ling, Hai Jin, Shadi Ibrahim, Wenzhi Cao, and Song Wu. Efficient disk I/O scheduling with qos guarantee for xen-based hosting platforms. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 81–89. IEEE Computer Society, 2012.

[14] Pedro Eugênio Rocha and Luis CE Bona. A QoS aware non-work-conserving disk scheduler. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2012.

[15] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: enabling high-level SLOs on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 14. ACM, 2012.

[16] Hui Wang, Kshitij Doshi, and Peter Varman. Nested qos: Adaptive burst decomposition for slo guarantees in virtualized servers. *Intel Technology Journal*, 16(2), 2012.

[17] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[18] Nitisha Jain and J Lakshmi. PriDyn: enabling differentiated I/O services in cloud using dynamic priorities. *IEEE Transactions on Services Computing*, 8(2):212–224, 2015.

[19] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. vFair: latency-aware fair storage scheduling via per-IO cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 125–138. ACM, 2015.

[20] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: enforcing the X th percentile latency and throughput SLOs for consolidated VM storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 28. ACM, 2016.

[21] Carl Staelin, Gidi Amir, David Ben-Ovadia, Ram Dagan, Michael Melamed, and Dave Staas. CSched: Real-time disk scheduling with concurrent I/O requests. Technical report, Citeseer, 2011.

[22] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, 2015.

[23] Zeqi Lai, Yong Cui, Minming Li, Zhenhua Li, Ningwei Dai, and Yuchi Chen. TailCutter: Wisely cutting tail latency in cloud CDN under cost constraints. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[24] Jonathan Mace, Peter Bodik, Madanlal Musuvathi, Rodrigo Fonseca, and Krishnan Varadarajan. 2dfq: Two-dimensional fair queuing for multi-tenant cloud services. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 144–159. ACM, 2016.

[25] Haishan Zhu and Mattan Erez. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News*, 44(2):33–47, 2016.

[26] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostić, and Sean Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 95–110. ACM, 2017.

[27] Vikas Jaiman, Sonia Ben Mokhtar, Vivien Quéma, Lydia Y Chen, and Etienne Rivìere. Héron: Taming Tail Latencies in Key-Value Stores under Heterogeneous Workloads. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 191–200. IEEE, 2018.

[28] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O Suminto, Cesar A Stuardo, Andrew A Chien, and Haryadi S Gunawi. MittOS: Supporting millisecond tail tolerance with fast rejecting SLO-aware OS interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 168–183. ACM, 2017.

[29] Chenyang Lu, John A Stankovic, Gang Tao, and Sang Hyuk Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*, pages 56–67. IEEE, 1999.

[30] Ozg ur Ertug, Mahesh Kallahalla, and Peter J Varman. Real-Time Parallel I/O Stream Scheduling. In *Proceedings 2nd Intl. Workshop. on Compiler and Architecture Support for Embedded Systems*, 1999.

[31] Intel. Intel® SSD DC S3700 Series. https://ark.intel.com/content/www/us/en/ark/products/71915/intel-ssd-dc-s3700-series-400gb-2-5in-sata-6gb-s-25nm-mlc.html.

[32] Intel. Intel® Xeon® Processor E5-2697 v2. https://ark.intel.com/content/www/us/en/ark/products/75283/intel-xeon-processor-e5-2697-v2-30m-cache-2-70-ghz.html.

[33] 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.