Managing Bufferbloat in Storage Systems

Esmaeil Mirvakili
UC Santa Cruz
Santa Cruz, California, USA
smirvaki@ucsc.edu

Samuel Just
Red Hat, Inc.
Los Angeles, California, USA
sjust@redhat.com

Carlos Maltzahn
UC Santa Cruz
Santa Cruz, California, USA
carlosm@ucsc.edu

Abstract

Today, companies and data centers are moving towards distributed and serverless storage systems instead of traditional file systems. As a result of such transition, allocating sufficient resources to users and parties to satisfy their service level demands has become crucial in distributed storage systems. The Quality of Service (QoS) is a research area that tries to tackle such challenges. The schedulability of system components and requests is of great importance to achieve the QoS goals in a distributed storage. Many QoS solutions are designed and implemented through request scheduling at different levels of system architecture.

However, the bufferbloat phenomenon in storage backends can compromise the request schedulability of the system. In a storage server, bufferbloat happens when the server submits all requests immediately to the storage backend due to a too large buffer in the storage backend. In recent decades, many research works tried to solve the bufferbloat problem for network systems.

Nevertheless, none of these works are suitable for storage system environments and workloads. This paper presents the SF_CoDel algorithm, an adaptive extension of the Controlled Delay (CoDel) algorithm, to mitigate the bufferbloat for different workloads in storage systems. SF_CoDel manages this purpose by controlling the amount of work submitted to the storage backend. The evaluation of our algorithm indicates that SF_CoDel can mitigate the bufferbloat in storage servers.

CCS Concepts: • Information systems → Storage management; Information storage systems; • Software and its engineering → File systems management.

Keywords: storage system, distributed storage, scheduling, bufferbloat, queuing system

1 Introduction

Scalability and availability drive the move to distributed and serverless storage systems. Traditional file systems cannot compete when it comes to high performance storage that can tolerate common storage faults, and can scale across thousands of machines. Examples of such modern storage systems include the Google File System (GFS) [7] and the Hadoop Distributed File system (HDFS) [18]. But distributed storage systems have their own challenges; ensuring Quality of Service (QoS) and fulfilling Service-Level Agreement (SLA) demands in the face of increasing complexity, number of components, and number of services is the focus of this paper.

To identify and understand the challenges of resource management and scheduling in distributed storage systems, one needs to understand such a system’s general architecture and design. Most distributed storage systems follow the Staged Event-driven Architecture (SEDA) [22], which offers a high-performance concurrent distributed system with the ability to control the request streams to achieve better resource management and schedulability. In detail, SEDA defines the system architecture as a network of stages that are connected via event queues. This architecture provides fine-grained control over client requests in the form of event queues. Many distributed storage systems such as Ceph [21], Cassandra [13], Amazon’s Dynamo [3] follow SEDA’s principles.

Distributed and scalable storage systems that follow SEDA principles consist of multiple components that communicate asynchronously via queues. In every storage system, a storage backend exists, at the lowest level, that places the data structure on the storage devices [1]. In order to store the data efficiently, the storage backend buffers the requests in a queue and flushes them on the device in a batch manner. By storing data in a batch, the storage backend amortizes the fixed overheads associated with single write transactions such as positioning, allocation, and metadata creation.

Storage systems can have multiple types of backends with different design assumptions about their underlying storage device technologies. Some storage systems, such as GFS [7], HDFS [18], and XtreemFS [8], use local file systems as their storage backends. On the other hand, other storage systems have implemented their own specialized storage backend to either remedy the local file systems problems, adopt new technologies, or improve the whole system’s performance. An example of such storage backends can be BlueStore [1], one of the popular storage backends in Ceph [21]. In either case, the storage systems, as Figure 1 depicts, consist of two main components, namely the Frontend, which manages high-level and abstract data storage, and the Backend, which handles the storage devices and low-level storage mechanisms.

In a scalable storage server, request scheduling is one of the most crucial aspects of Quality of Service (QoS). The reason is that most of the essential features of QoS can be achieved through request scheduling, such as fair resource
sharing, throughput and latency guarantees, and performance isolation [23]. Such a request scheduling mechanism tries to ensure different levels of service for different classes of requests. For example, requests generated by data scrubbers working in the background generally have a lower priority than requests from an application.

One of the design decisions in storage servers is where the request scheduling should reside, on the frontend or the backend. We can find the answer in the four design principles of software-defined storage (SDS) [14]. The first principle suggests that the storage system design should decouple the storage mechanisms from the control policies over data. Based on this principle, the SDS architecture consists of two decoupled layers: the Control Plane and the Data Plane. The control plane holds the system-wide control building blocks, and the data plane holds the controller-defined storage operation stages. Based on SDS architecture, the best place for request scheduling is the frontend component—which represents the control plane in SDS architecture.

Moreover, most storage systems have multiple backends with different design that can work with different storage devices. As a result, placing the request scheduling in the backend means that we need to implement different scheduling mechanisms to target unique characteristics of different storage backends.

However, employing the request scheduling on the frontend presents a new problem. For effective request scheduling, the frontend needs to hold enough requests in its queue. On the other hand, the backend has to receive enough requests to work efficiently with acceptable performance. This raises the question: what is enough for the frontend and the backend? There are two problematic cases regarding balancing the number of requests in the frontend and the backend:

1. There are too many requests on the frontend and a few on the backend.
   In this case, since the scheduling mechanism has access to most of the current in-flight requests, it can achieve its full objectives. However, the backend cannot perform efficiently since it is starving.
2. There are a few requests on the frontend and too many on the backend.
   This case results from having a huge or infinite queue on the backend without any back-pressure mechanism. This case results in an ideal performance since the backend is saturated with requests. On the other hand, the frontend scheduling mechanism cannot fulfill its objectives on a few requests. The problem of buffering too much data to a downstream is known as Bufferbloat [6].

Most storage systems exhibit bufferbloat, resulting in their suffering from schedulability and QoS issues. To address this issue, the storage server needs to adopt an admission control mechanism on the backend—thus balancing the requests between the frontend and backend (Figure 1).

![Figure 1. The storage server architecture with Frontend and Backend.](image)

The bufferbloat is one of common problems in network systems where excessively large buffers in the system cause undesired latency. The network experts have provided multiple solutions to mitigate the bufferbloat such as Random Early detection (RED) [5], Controlled Delay (CoDel) [15], and Proportional Integral controller Enhanced (PIE) [16]. The bufferbloat solutions in network can be adopted to implement the admission control as the storage systems can be treated as network systems. There are multiple works that adopt network abstractions and solutions in storage systems [19, 20].

To address this issue in storage systems, we introduce the SlowFast_CoDel (SF_CoDel), a specialized adaptive algorithm for storage systems for different workloads and storage devices. We adopt the CoDel algorithm, an existing solution for bufferbloat in network systems, as the base algorithm and design a new algorithm called SlowFast_CoDel. The SlowFast_CoDel can address storage systems’ issues and requirements related to the bufferbloat. Furthermore, we implement our algorithm in BlueStore, Ceph’s storage backends [1], and evaluate it against different workloads. Our main contributions in this paper are as follows:

1. We analyze the storage backend system for the bufferbloat problem and show that this problem exists.
2. We design a new adaptive CoDel algorithm, SlowFast_CoDel, that can adapt to different workloads and provide enough requests for the frontend scheduling.
3. We implement our algorithm for BlueStore, one of Ceph’s storage backends [1], and evaluate it against different workloads.

The rest of this paper is structured as follows. Section 2 describes our algorithm design in detail. In section 3, we evaluate our implementation of our algorithm in BlueStore. Finally, we discuss the related works in Section 4 and conclude in Section 5.
2 Design

2.1 CoDel algorithm for Storage Systems

The Controlled Delay algorithm monitors the queuing delay of the requests in defined intervals and decides how to handle the incoming flow. This algorithm has two key parameters, namely target delay and interval. The CoDel compares the minimum queue delay observed in every interval with the target delay. If the minimum delay exceeds the target, the CoDel initiates packet loss to make the upstream component decrease the TCP window and limit the packet flow. Moreover, with every violation, CoDel shortens the interval to adapt to the latency changes faster.

However, we cannot use the CoDel algorithm as it is for the storage systems for the following reasons:

1. The CoDel uses deliberate packet loss as a signal to upstream to control the packet flow. However, storage systems require a different signal.
2. The storage backend internals can be very complicated. As a result, tracking and measuring the queuing delay is not always feasible.

For adopting CoDel in storage systems, we apply some changes in the CoDel algorithm and propose Algorithm 1 for admission control in storage systems.

First, we define a Queuing Budget for the submitted (in-flight) requests in the backend. Every request has a cost according to its size and nature. After a request enters the backend, the queuing budget will decrease by the cost of the request. If the backend’s queuing budget reaches zero, the admission control will prevent the frontend from submitting any requests. The rest of the requests should wait in the frontend until the budget increases or some requests exit the backend and release their used queuing budgets. The Queuing Budget Adjusting CoDel in Algorithm 1 controls the backend queuing budget directly. In case of a latency violation, the algorithm decreases the queuing budget based on the difference between minimum latency and the target.

Another change to the CoDel is that we measure the backend’s total latency instead of queuing delay for two reasons:

1. As mentioned before, measuring the queuing delay in the backend might not be feasible due to the complexity of its design. The storage backend usually holds multiple stages and queues based on their design objective.
   For example, in BlueStore, the requests are queued to be stored by the asynchronous IO library provided by the OS. After that, the metadata associated with requests is stored in RocksDB [4]. Figure 2 shows a simplified version of BlueStore architecture and the requests flow path and states. In such systems, it is difficult to track the queuing delay.
2. One of our design goals is the generality of the solution for different storage backend so that it can be applied to different storage backend with the minimum amount of change. We can achieve this generality by considering the backend as a black box by only measuring the total latency.

![Figure 2. Requests path and states in the BlueStore from OSD (frontend) to the storage device.](image)

Algorithm 1: The Queuing Budget Adjusting CoDel algorithm for Storage Systems

```plaintext
violation_count = 0;
INTERVAL = initial_low_interval;
in every step
begin
    if min_latency > TARGET then
        Decrease backend_queuing_budget according to |min_latency − target|;
        violation_count++;
        INTERVAL = INTERVAL / violation_count;
    else
        Increase backend_queuing_budget by budget_increment;
        INTERVAL = initial_interval;
        violation_count = 0;
end
sleep for INTERVAL;
```

The Queuing Budget Adjusting CoDel can reduce the backend latency and increase frontend latency. This shows that the algorithm effectively alleviates the bufferbloat by limiting the submitted requests to the backend. However, the results also show that the algorithm reduces the throughput. This is quite natural that by limiting the request flow to the backend, the backend throughput reduces.

The most important challenge to employ the Queuing Budget Adjusting CoDel algorithm in a storage system is that the parameter Target is completely workload-dependent, meaning that different workloads need a different value of Target to operate under acceptable performance. This means that the algorithm needs parameter tuning for every different workload to work at the desired performance state. However,
when the workload is dynamic and unstable, having predefined and fixed CoDel parameters is proved to be difficult. Consequently, the storage system requires an auto-tuning and adaptive algorithm that can overcome these issues. For this matter, we propose the SlowFast CoDel, an adaptive version of CoDel, in the next section.

2.2 SlowFast CoDel

To make the CoDel parameter (target) adaptive, we designed SlowFast CoDel (SF_CoDel) algorithm. This algorithm consists of two separate optimization loops, namely fast or inner and slow or outer Loop. Figure 3 shows the structure of this algorithm.

![Figure 3. SlowFast CoDel algorithm consists of Queuing Budget Adjusting CoDel algorithm (Fast loop) and Target Adjusting Algorithm (Slow loop).](image)

The fast loop is a high-frequency loop that monitors the backend latency and minimizes the latency by controlling the backend queuing budget based on the Target parameter. In other words, the Fast loop is the Queuing Budget Adjusting CoDel algorithm (Algorithm 1). The responsibility of this part is to decrease the backend latency using a given Target. For this purpose, measuring and controlling the latency in high-frequency (short intervals) can help the backend react to latency spikes quickly and reduce the excess latency. We address this fast loop as the Queuing Budget Adjusting CoDel in the rest of the paper.

The Slow loop (Target Adjusting Algorithm) is a low-frequency loop that monitors the backend throughput and tries to balance the throughput loss and latency reduction by controlling the Queuing Budget Adjusting CoDel’s Target parameter. In fact, the Target Adjusting Algorithm can detect any workload change and optimize the Target parameter with respect to the throughput-latency trade-off. Since the throughput of the backend is usually unstable due to the effect of external sources such as compaction or device behavior, a low-frequency sampling over a longer interval is the best approach to keep the optimization stable. We address this slow loop as the Target Adjusting Algorithm in the rest of the paper.

2.2.1 Throughput Latency Trade-off. As mentioned in Section 2.1, the important challenge to employ the Queuing Budget Adjusting CoDel algorithm in the backend is how to choose a suitable Target parameter to both bufferbloat mitigation and preserving the throughput at an acceptable level.

Figure 4 shows a typical Throughput-Latency curve for storage backends (and many other systems). This curve indicates that by having a high latency, the backend can achieve high throughput. However, the backend will reach its maximum capacity at some point, but latency keeps rising due to requests queuing delay. In other words, latency and throughput increase by increasing the request rate until the backend is saturated with requests. After that point, the latency will increase but not the throughput. Regarding the schedulability of the frontend, requests are immediately submitted to the backend in all cases. As a result, the schedulability of the frontend is still compromised.

![Figure 4. The throughput-latency curve in storage backend. By increasing the latency, throughput increases in logarithmic rate until it reaches the system saturation point.](image)

The Queuing Budget Adjusting CoDel algorithm can decrease the latency and solve the schedulability issues of the frontend. However, the cost will be the throughput. As a result, we implement the Target Adjusting outer loop to control the Target based on the Throughput-Latency curve to preserve the desired trade-off between throughput and latency. The Queuing Budget Adjusting CoDel algorithm achieves this by finding a latency where the slope of the tangent line to the throughput-latency curve is equal to a TargetSlope parameter.

The slope of the tangent line to the throughput-latency curve at a point is an excellent indication of throughput loss in respect to the latency reduction. A value of zero means no throughput loss for latency reduction, and a value of 5 means that by reducing the latency by one unit, the throughput can
drop by five units. The Target Adjusting Algorithm always chooses a Target where the slope of the tangent line to the throughput-latency curve is equal to TargetSlope. By doing this, it can achieve a consistent and same throughput latency trade-off for different workloads.

2.2.2 Target Adjusting Algorithm (Slow Loop). The Target Adjusting Algorithm needs to converge the Target to a value where the slope of the tangent line to the throughput-latency curve is TargetSlope. A typical method for solving such problems is using a gradient descent-based algorithm. In such methods, the algorithm tries to minimize an objective function in iterative steps. In every step, the algorithm updates the controlling parameter by calculating the gradient of the objective function at that moment and estimate the amount of necessary change. There are many successful examples of using gradient descent, such as Kalanat et al.’s works to find optimized actions in social networks [11, 12].

However, using gradient descent in an unstable system with a huge amount of variation is not suggested. Storage system behavior can be very noisy due to different factors such as background tasks, type of storage device, faults in devices, etc. For example, Figure 5 shows the latency of BlueStore during five minutes of execution. As the Figure indicates, the behavior of the backend is very noisy due. In such systems, gradient descent-based optimizations are not helpful.

$$\frac{df(x)}{dx} \propto \frac{1}{x} \quad (1)$$

One of examples for this function is a logarithmic function in form of \( f(x) = a + b \times \ln(x) \). By applying a regression on a limited sampled throughput-latency points, the Target Adjusting algorithm can find the a and b for the estimated function. Figure 6 shows such a regression result. After that, the algorithm can use the derivative of \( f(x) \), Equation 2, to find the x where the slope of the tangent line to the curve is equal to TargetSlope (Equation 3).

$$f'(x) = \frac{df(x)}{dx} = \frac{b}{x} \quad (2)$$

$$\text{Target} = \frac{b}{\text{TargetSlope}} \quad (3)$$

However, to keep a stable regression against system variation and instability, the sampled data points should be distributed through a long enough latency range. Figure 6a shows well-distributed sampled data points in the latency range of 0ms to 100ms. However, the sampled data points in 6b are concentrated in a short range of latency (40ms to 60ms). As a result, the variation in the system have a greater impact on regression in 6b than 6a.

![Figure 5. BlueStore latency](image)

For implementing the Target Adjusting Algorithm, we need a stable algorithm that can estimate the throughput-latency curve of the system at any time and find the optimal Target. For this purpose, in every iteration of the loop, we store the average throughput and the Target in Queuing Budget Adjusting CoDel during the last interval in a limited size history. At the end of every iteration of the Target Adjusting Algorithm, we calculate an estimation of the throughput-latency curve function using regression. Using the estimated function, the algorithm can easily calculate the Target where the slope of the tangent line to the throughput-latency curve is equal to TargetSlope.

We need to use a non-linear function \( f(x) \) in which:

![Figure 6. The sampled data points in both graph 6a and 6b are noisy points from curve \( f(x) = 1 + 3ln(x) \). The estimated curve using well distributed data points in 6a is much more accurate.](image)
target so that the regression can have sampled data points in a longer range of latency.

However, the added noise to Target should not impact the Queuing Budget Adjusting CoDel and backend throughput. To achieve this, the Target Adjusting algorithm selects the Target randomly from a log-normal distribution with a mode of 2, mean of 1, and standard deviation of 0.55. The log-normal distribution’s important characteristic is that the occurrence of values close to the mode has the highest probability, and the probability of high values is much lower than low values. As a result, in a long period of time, the selected noisy Target by the algorithm is mostly close to the selected optimal point. By applying such a noise, the algorithm can have a better distributed data point samples and achieve the convergence of Target to the optimal value at the same time.

Algorithm 2 shows the complete version of the Target Adjusting algorithm.

2.3 Workload Dependent vs Workload Independent Parameters
A workload-dependent parameter needs to be tuned and defined for the target workload. On the other hand, a workload-independent parameter performs its desirable effect on the system independent of what kind of workload the system operates on. An adaptive and auto-tuning system should be free of any workload-dependent parameters. Instead, it should provide some workload-independent parameters so that the user/operator can customize it based on demands and preferences.

In the Target Adjusting Algorithm in SlowFast CoDel, the TARGET_SLOPE is a workload-independent parameter that provides control over throughput latency trade-off. As a result, the system can be deployed based on different throughput latency trade-off preferences for any workloads.

| Algorithm 2: Target Adjusting algorithm for adapting the target parameter in Queuing Budget Adjusting CoDel |
|---------------------------------------------------------------|
| 1 INTERVAL = initial_high_interval;                           |
| 2 throughput_target_history = [];                            |
| 3 in every step                                               |
| 4 begin                                                       |
| 5 throughput = backend throughput over the passed INTERVAL;   |
| 6 Add tuple(TARGET, throughput) to                           |
|    throughput_target_history;                                 |
| 7 if length(throughput_target_history) >                     |
|    history_len_threshold then                                  |
|    Remove the oldest data from                               |
|    throughput_target_history;                                 |
| 8 end                                                         |
| 9 a, b = find the logarithmic curve                            |
| 10 f(x) = a + b(ln(x)) by regression over                      |
|    throughput_target_history;                                 |
| 11 optimal_target = TARGET_SLOPE;                             |
| 12 TARGET = lognormal_noise(mode =                           |
|    optimal_target);                                           |
| 13 sleep for INTERVAL;                                        |
| 14 end                                                        |

3 Evaluation
In this section, we evaluate our algorithm by comparing it to the BlueStore without any admission control. We implement the SF_CoDel algorithm in BlueStore and evaluate the implementations against two different workloads (4KB and 64KB writes with queue depth of 1024) generated by FIO [2] on an SSD storage device.

We use workloads with a queue depth of 1024 to make sure that the bufferbloat is happening in the system. Moreover, to have the BlueStore without any admission control as the baseline, we disable the BlueStoreThrottle (a static budget-based admission control in BlueStore). We talk about BlueStoreThrottle in Section 4 (Related Works) and why it is not enough to mitigate bufferbloat in BlueStore.

3.1 Target Adjustment and Convergence
Figure 8 shows Target adjustment by Target Adjusting Algorithm (Slow loop) for 4KB and 64KB writes workloads. The Target Adjusting Algorithm adjust the Target according to the workload and log-normal noise model. As Figure 8 depicts, range and convergence of adjustment is according to the workload. For example, range of Target adjustments for 64KB writes (8b) are significantly greater than 4KB writes workload (8a). The algorithm uses the same Target_Slope = 5 to adapt the Target for both workload.
Moreover, in Figure 9, the workload changes from 4KB writes to 64KB writes at the time of 300 seconds. The figure shows that the Target Adjusting Algorithm detects the workload change and adjusts the Target and the log-normal model according to the workload.

However, all of the Target adjustments results show some unsuitable drastic changes in Target. For example, in Figure 8b between time 50 and 100 seconds, there is temporary increase in Target value. These changes are usually due to factors such as compaction, background tasks, and device faults. However, after some time, the Target adjustment goes back to its normal trend.

Figure 8. The adjustment of parameter Target by Target Adjusting Algorithm (with Target_Slope of 5) for two different workload.

3.2 Throughput Latency Trade-off and Bufferbloat Mitigation

Figures 10 and 11 show the BlueStore latency and throughput comparison between BlueStore without admission control and BlueStore with SlowFast CoDel for 4KB and 64KB writes workloads. Figure 10a shows that the tail (95 and 99 percentile) latency reduces more than 50%, and the average latency decreases over 60% for 4KB writes. Moreover, Figure 11a shows that the tail (95 and 99 percentile) latency and the average decreases about 90% for 64KB writes. From these results, it is obvious that the SlowFast CoDel blocks the excess requests from entering the BlueStore by adjusting the Backend Queuing Budget. As a result, the BlueStore latency of requests decreases drastically. In other words, by using SlowFast CoDel, the requests spend most of their time in OSD (frontend) queue. This shows that the SlowFast CoDel can mitigate the bufferbloat issue in the BlueStore effectively.

Figures 10b and 11b show the throughput loss in the BlueStore with SlowFast CoDel in comparison to BlueStore without any admission control. For both 4KB and 64KB write workloads, throughput loss is about 20%. The parameter Target_Slope controls the throughput loss. In both cases, Target_Slope value is five.

Figures 12 and 13 show the impact of different values of Target_Slope (0.1, 0.5, 1, 5, 10, and 20) on latency and throughput for both 4KB and 64KB write workloads. These figures show that how Target_Slope affect and control the throughput and latency changes in the BlueStore. High values of Target_Slope favor the latency over throughput and achieve a stable and very low latency in cost of huge throughput loss. On the other hand, low values of Target_Slope result in a better throughput but high latency.

By comparing the Figures 12a and 13a more closely, we can observe that high values of Target_Slope can achieve a very similar latency distribution across different workloads (4KB and 64KB writes). This can lead to a very predictable and stable backend which can have valuable advantages for the schedulability of the system. For example, the frontend scheduling algorithm can use this predictable latency for latency guarantees under dynamic and unstable workloads.

Figure 9. The adjustment of parameter Target by the Target Adjusting Algorithm in case of workload change. The workload at first 300 second is 4KB writes, but at time 300 second, the workload changes to 64KB writes.
Related Work

In the last few decades, many works [5, 9, 10, 15, 16] have extensively studied the bufferbloat as a network latency problem. However, managing bufferbloat in storage systems requires different solutions and analyses due to the different environments and workloads. There are not many dedicated works on solving bufferbloat in storage systems to the best of our knowledge.

[17] introduces a control knob, called key-frame Sim, to their architecture to remedy the bufferbloat problem for specialized datastore for latency-critical machine vision. Key-frame Sim discards a number of closest matching video key-frame from their buffers if the video stream transmission bufferbloat exceeds a threshold. However, key-frame Sim is designed for video stream flows for machine vision applications only.

In BlueStore, a budget-based admission control mechanism, called BlueStoreThrottle, controls the amount of queued work in the BlueStore by blocking the OSD from submitting more work. However, currently, this mechanism works with a static and constant maximum budget that needs to be tuned and configured before deployment. As a result, BlueStoreThrottle cannot adapt to different workloads.

In the context of adaptive queue management algorithms, there are some works on making these algorithms adaptive to handle different workloads and environments. PIE [16] is a self-tuning algorithm that controls queuing delay by calculating the probability of packet drop over time. However, adopting the PIE algorithm for storage systems can be difficult since the packet loss mechanism is the core of the PIE algorithm. Changing this mechanism to queue size control requires excessive latency study.
The difference between our approach and other works is that we design the SlowFast CoDel algorithm specialized for the storage systems’ workloads and environments that can automatically adapt to different workloads. For such systems, we need easy-tuning algorithms with few parameters and assumptions about other parts of the systems. Most previous adaptive algorithms have many parameters to tune and assumptions about the system and network that are not completely true or unavailable for storage systems. For example, ACoDel-IT and ACoDel-TIT [24] require an assumption about bottleneck link capacity and TCP window size.

5 Conclusion

We adopt the CoDel algorithm for bufferbloat mitigation in the storage backend. However, based on our experiments in Section 3, different workloads in storage systems require different Target parameters to have the desired performance trade-off. Then, we introduce the SlowFast CoDel, an adaptive extension of CoDel, that can adapt the Target parameter of CoDel to different workloads. SlowFast CoDel achieves this goal by using a slower loop that adjusts the Target parameter based on the desired throughput latency trade-off and a workload-independent parameter called Target_Slope.

The results in Section 3 show that our algorithm can mitigate the bufferbloat by decreasing the backend latency and a controlled throughput loss.

Acknowledgments

This work has been made possible by the Center for Research in Open Source Software at UC Santa Cruz (cross.ucsc.edu) and supported by the National Science Foundation under Cooperative Agreement OAC-1836650.

References

[1] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In Proceedings of the 27th ACM Symposium on Operating Systems Principles. 353–369.
[2] Jens Axboe. 2014. Fio-flexible io tester. URL http://freecode.com/projects/fio (2014).
[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon’s highly available key-value store. ACM SIGOPS operating systems review 41, 6 (2007), 205–220.
[4] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In CIDR, Vol. 3. 3.
[5] Sally Floyd and Van Jacobson. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking* 1, 4 (1993), 397–413.

[6] Jim Gettys. 2011. Bufferbloat: Dark buffers in the internet. *IEEE Internet Computing* 15, 3 (2011), 96–96.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.

[8] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. 2008. The XtreemFS architecture—a case for object-based file systems in Grids. *Concurrency and computation: Practice and experience* 20, 17 (2008), 2049–2060.

[9] Haiqing Jiang, Zeyu Liu, Yaogong Wang, Kyunghan Lee, and Injong Rhee. 2012. Understanding bufferbloat in cellular networks. In *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*. 1–6.

[10] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. 2012. Tackling bufferbloat in 3G/4G networks. In *Proceedings of the 2012 Internet Measurement Conference*. 329–342.

[11] Nasrin Kalanat and Eynollah Khanjari. 2020. Action extraction from social networks. *Journal of Intelligent Information Systems* 54, 2 (2020).

[12] Nasrin Kalanat and Eynollah Khanjari. 2020. Extracting actionable knowledge from social networks with node attributes. *Expert Systems with Applications* 152 (2020), 113382.

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

[14] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. 2020. A Survey and Classification of Software-Defined Storage Systems. *Comput. Surveys* 53, 3 (2020). https://doi.org/10.1145/3385896

[15] Kathleen Nichols and Van Jacobson. 2012. Controlling queue delay. *Commun. ACM* 55, 7 (2012), 42–50.

[16] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili Suryanarayana Prabhu, Vijay Subramanian, Fred Baker, and Bill VerSteeg. 2013. PIE: A lightweight control scheme to address the bufferbloat problem. In *2013 IEEE 14th international conference on high performance switching and routing (HPSR)*. IEEE, 148–155.

[17] Arun Ravindran and Anjus George. 2018. An edge datastore architecture for latency-critical distributed machine vision applications. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*.

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

[19] Joan Stefanovic, Bianca Schroeder, Greg O’Shea, and Eno Thereska. 2019. Srout: Treating the storage stack like a network. *Proceedings of the 14th USENIX Conference on File and Storage Technologies, FAST 2016* (2019), 197–212.

[20] Eno Thereska, Hitesh Ballani, Greg O’Shea, Thomas Karagiannis, Antony Rowston, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow. (2013). 182–196. https://doi.org/10.1145/2517349.2522723

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

[22] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 230–243.

[23] Suli Yang, Jing Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2018. Principled schedulability analysis for distributed storage systems using thread architecture models. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 161–176.

[24] Jiancheng Ye and Ka-Cheong Leung. 2019. Adaptive and stable delay control for combating bufferbloat: Theory and algorithms. *IEEE Systems Journal* 14, 1 (2019), 1285–1296.