An Interposed I/O Scheduling Framework for Latency and Throughput Guarantees

Quan Zhang*, Dan Feng, Fang Wang and Yanwen Xie

Wuhan National Laboratory for Optoelectronic, Wuhan 430074, P.R. China
School of Computer, Huazhong University of Science and Technology, Wuhan 430074, P.R. China

Abstract

Cloud storage system is becoming a trend in production environments for its economic benefits. With such architecture, storage resource is consolidated to provide multiplexing service for concurrent applications. Therefore, storage centers must be able to guarantee multi-dimensional Quality of Service for various applications. However, satisfying performance targets for each workload is challenging, because they compete for storage resource and have various performance targets in terms of throughput or latency. In this paper, we design and implement a novel scheduler, called Arbitrator, to maintain per-application performance no matter in terms of throughput or latency. In our scheduling framework, we introduce a factor to reflect how applications are sensitive to deadline missing. The scheduler employs a feedback mechanism to monitor latency guarantees and throughput allocation for each application, and compute how much applications deviate from their performance targets. Based on the estimation, Arbitrator makes the scheduling decision to achieve latency guarantee and proportional sharing of bandwidth. We implement Arbitrator in Linux kernel and evaluate its effectiveness, and the results show the scheduler has good ability to maintain satisfactory performance for applications.

Key Words: I/O Scheduling, Quality-of-Service, Shared Storage System, Performance Management

1. Introduction

With explosive growth of internet applications and data volumes, storage system becomes a service-oriented paradigm due to its attractive economic benefits of managing massive data and providing multiplexing storage service for applications [1]. Though this paradigm can produce benefits through sharing of underlying storage infrastructure such as disk devices, memory and network, customers may not be able to achieve their performance targets. That is because it will result in competition of storage resource and lead to significantly interference between applications, which may violate service level agreement (SLA) between customers and cloud storage providers. So QoS scheduler should be integrated into storage systems to manage applications’ performance. However, storage systems may face with several critical challenges to provide performance guarantees for various applications.

First, the mechanism should support performance differentiation in terms of both latency and throughput, as applications sharing the storage system usually have different performance targets. For example, applications such as virus scanning usually require high throughput but not sensitive to latency, while other applications such as online transaction processing typically require bounded I/O response time but low throughput. We believe that lack of widely integrating QoS mechanisms into consoli-
dated storage system is mainly due to the lack of supporting both latency and throughput requirement.

Second, different applications may have different requirement on timeliness. Even though time-critical workloads can be specified with low response time, this latency requirement cannot reflect how application is sensitive to deadline missing. For example, one can specify latency requirement for OLTP with 50 ms and 100 ms for multimedia, but these latency requirement can not reflect the fact that multimedia can tolerate timeout for some requests and only requires soft timeliness, while OLTP is sensitive to timeout. Previously schedulers [2,3], based on Earliest Deadline First cannot differentiate applications with different timeliness requirement.

Third, the scheduler should make no assumption about underlying storage system [2]. Currently, various storage devices are used in storage center, and the system architecture is becoming more complicated. The work of integrating QoS mechanisms into various storage systems would become impractical if it heavily depends on implementation of underlying storage system.

Thus, some solutions were proposed to manage storage resource and provide performance-based QoS for applications in shared data centers. Such as PARDA [4] and mclock [5] aim at providing proportional sharing of storage resources for applications, while other schedulers [6,7] focus on meeting per-application latency target. Also, some previous algorithms [8,9] that employ static hierarchical schedulers provide multi-dimensional QoS guarantees. In the hierarchical schedulers, high-level scheduler is responsible for throughput allocation while low layer takes charge of latency guarantee. But these hierarchical schedulers potentially provide fair bandwidth allocation preferentially. Maestro [10] supports absolute performance guarantee either in terms of latency or throughput. Also, it uses a pre-application priority to differentiate importance of applications, but this priority cannot reflect timelessness requirement.

Therefore, we proposed a scheduling framework, called Arbitrator to leverage latency and throughput requirement. Our approach is based on interposed request scheduling, which requires little information about implementation of the underlying storage system. The interposed scheduler intercepts requests from all applications and reorders requests before dispatching them to underlying storage utility. Also, Arbitrator monitors actual latency and throughput for each application class, and computes how much applications deviate from their performance targets in last period, which helps Arbitrator make appropriate scheduling decision and guarantee performance targets for applications.

The remainder of this paper is organized as follows. The next section will describe the system model and define some metrics used in our scheduling algorithm. Section 3 gives overview of our framework, together with detailed implementation of Arbitrator. The evaluation results are presented in section 4. Finally, we will review representative related works and summarize our work.

2. System Model

2.1 System Model

The storage system, as shown in Figure 1, provides shared storage service to all arrival requests. In this model, each incoming request is classified into corresponding application based on its performance-based service level objective (SLO). Customers should specify performance targets for their applications in terms of throughput and latency. Also, customers should specify timelessness to reflect how applications are sensitive on latency.

All requests sent to storage server are intercepted by the interposed scheduler, and then dispatched to the storage utility according to its internal policies to share re-
sources fairly among the applications according to their assigned ratios. So our system architecture maintains a separate request queue for each application class, shown as Figure 1. Once requests arrive, they are added to the corresponding request queue firstly before sending to backend storage. Then the Arbitrator selects a request from the right request queue in FIFO order and dispatches it to storage server. In our system model, we make no assumption about the architecture of underlying storage server. It may be a disk array or Solid State Drives, so the underlying systems can employ other scheduling algorithm (SCAN for hard disk or FIFO for SSD) to optimize its efficiency.

2.2 Metric Definition

In shared storage system, the performance received by one application not just depends on its workload characterization or performance target, but also rely on performance requirement of other applications and the capability of storage system. Different from reservation-based scheduler [11] just providing minimum reservations for applications through over-provisioned resource, proportional-share schedulers should allocate storage resource to applications in proportion to their weights no matter the server is under-loaded or overloaded. This means applications can gain extra performance proportionally from over-provisioned resource, or violate performance target equitably when the server gets overloaded. For this reason, it is important to measure windfall or violation value for each application. Violation refers to how much the application deviates from its service level objectives. Similar to the definition of windfall metric, we normalize violation value for applications.

**Definition 1:** Let $T_{\text{goal}}$ denotes the throughput requirement for an application, and the actual throughput of the application observed in the interval $(t_1, t_2)$ is $t$. Then throughput windfall gained by the application in the interval $(t_1, t_2)$ can be defined as:

$$w_i(t_1, t_2) = \begin{cases} 
(t - T_{\text{goal}}) / T_{\text{goal}}, & t \geq T_{\text{goal}} \\
0, & t < T_{\text{goal}}
\end{cases}$$

Similarly, $L_{\text{goal}}$ and $l$ denote latency requirement and actual average response time of the application observed in the interval $(t_1, t_2)$. Then latency windfall is:

$$v_i(t_1, t_2) = \begin{cases} 
(l - L_{\text{goal}}) / L_{\text{goal}}, & l \leq L_{\text{goal}} \\
0, & l > L_{\text{goal}}
\end{cases}$$

We next define violation value to quantificationally measure how much application deviates from its service level objectives. Similar to the definition of windfall metric, we normalize violation value for applications.

**Definition 2:** Assume $T_{\text{goal}}$ and $t$ denote the throughput requirement and actual throughput of the application. Then throughput violation can be defined as:

$$v_i(t_1, t_2) = \begin{cases} 
(T_{\text{goal}} - t) / T_{\text{goal}}, & t \leq T_{\text{goal}} \\
0, & t > T_{\text{goal}}
\end{cases}$$

Similarly, when storage system is overloaded, latency violation can be defined as:

$$v_i(t_1, t_2) = \begin{cases} 
(l - L_{\text{goal}}) / L_{\text{goal}}, & l \geq L_{\text{goal}} \\
0, & l < L_{\text{goal}}
\end{cases}$$

In order to provide per-application performance guarantees for IO-intensive application or time-critical applications, we compute violation value through combining $v_i(t_1, t_2)$ and $v_i(t_1, t_2)$. To differentiate how applications are sensitive on latency and throughput, two tunable weights, denoted as $\alpha$ and $\beta$ are used in the definition and violation value can be defined as:

$$V(t_1, t_2) = v_i(t_1, t_2)\alpha + v_i(t_1, t_2)\beta$$

Simply, we can assume $\alpha$ to be a relative value to $\beta$,
(specified with 1), then $\alpha$ can be assigned a value between 0 and 1 for IO-sensitive workload and a value between 1 and 2 for time-critical workload. Then we can simplify violation function as:

$$V(t_1,t_2) = v_i(t_1,t_2)\alpha + v_i(t_1,t_2)$$ (6)

The violation function presents three important properties:

**Property 1:** Smaller value of $V(t)$ means performance of application $app_i$ get closer to its performance target. Especially, when $app_i$ has meet its performance requirement, $V(t)=0$.

**Property 2:** For each application, if its throughput is below the limit or the latency exceeds its upper bound, $V(t)>0$, and we declare any violation on throughput or latency as SLO violation for application.

**Property 3:** If two applications have equal violation value for latency and throughput respectively, time-critical applications will get larger violation value based on violation function. That is because a larger value $\alpha$ is assigned to applications that are more sensitive to latency.

Similarly, we can define windfall function as:

$$W(t_1,t_2) = w_i(t_1,t_2)(2-\alpha) + w_i(t_1,t_2)$$ (7)

Contrary to windfall metric, higher value is better for this metric since it means the application receives better performance than its desired requirement. So the SLO for the $i$th ($1 \leq i \leq m$) application can be described by a three-dimensional tuple $<T_i,L_i,\alpha>$, where $T_i$ is the throughput target (I/Os per second) and $L_i$ is the latency target for application $app_i$, $\alpha_i$ ($0 \leq \alpha_i \leq 2$) represents the degree of timelessness required.

3. Scheduling Algorithm

In this section we will present the scheduling algorithm that control request accesses to the storage server so as to maintain proportional sharing of throughput among applications while providing latency guarantees for requests. To provide per-application performance guarantee, our system architecture maintains a separate request queue for each application class. All incoming requests are put into the application-specific queue in order of their arrival times and wait to be dispatched into back-end storage device.

Instinctively, requests belonging to the application which has missed its SLO should be dispatched to underlying storage device preferentially, since this can increase bandwidth and decrease average latency for the application. So Arbitrator dispatch requests based on the performance that each application has received, which is different from traditional schedulers [8,12] that attach timestamp-based flag to each request and dispatch requests in the order of flag values.

In our scheduling framework, a monitor is employed to continuously observe the bandwidth and latency performance actually received by each application. Based on the instantaneous performance of each application, the scheduler can compute the windfall and violation value for each application based on the functions defined in previous section. For each application, if its violation value is larger than 0, which means its throughput is below to its target or the latency exceeds its upper bound, the QoS scheduler perform corrective actions to bridge the gap between actually performance and the SLO.

If several applications have not met their performance target, the scheduler will dispatch requests from application that has the largest violation value preferentially. When these requests have completed, the violation value for the application will decrease. And in the long term, all applications have the same violation value and share storage resource proportionally.

Different from the leaky bucket mechanism [5] that uses an absolute rate to control accesses and cannot take full use of spare storage resource, Arbitrator allocates spare resource to applications proportionally based on windfall value if all applications have met their performance target. It dispatches requests from application which has the smallest windfall value. After completion of requests, the proportion for the application sharing the server increases and all application share storage resource proportionally in the long term.

After completion of each request, the scheduler framework should update violation value and windfall
value. To avoid dispatching requests from the same application all the time, $V(t)$ and $W(t)$ should be recomputed based on the latency and throughput statistical information of last period using the metrics defined in previous section. Actually, the time interval $t$ in above equations is a slide window. When each I/O request has completed, the end of the slide window should be moved to completion time of the request. Through this short slide window, Arbitrator can make decision based on I/O statistics of latest period and adjust scheduling decision to fit variability of workload.

4. Evaluation

Our prototype is implemented in the Linux kernel as a loadable module. The module creates several pseudo devices (entries in /dev), which can be backed up by a physical block device or disk arrays. To avoid interference from operating system, all pseudo devices are installed on an extra hard disk, which is only accessed by our module. In our experiments, different applications with different service level objectives, access different pseudo devices. This allows QoS scheduler to recognize each request belongs to which application, monitor I/O statistics and compute violation value for each pseudo device. Our scheduler module intercepts all requests and dispatches them to corresponding pseudo devices. When requests are passed to underlying disk device, the disk driver dispatches requests using FCFS algorithm to obey Arbitrator scheduler.

All experiments are set up on an Inspur node, which is composed of 2.4 GHz Intel Xeon Dual-Core CPU and 12 GB memory. A 1 TB SATA hard drive (15000 RPM) is used for operating system and another disk is used for our prototype. A workload generator is implemented to generate synthetic workloads and evaluate Arbitrator. In real environment, various applications or users access storage server at different time. To demonstrate that the scheduler can guarantee proportional sharing of storage resource even in such dynamic environment, two generators were started at the beginning of the test and another generator was activated in 100 seconds later. Each generator employs 20 threads to generate requests, and each request is 64 KB. All requests are random access and the read/write ratio is 1:1. The capability of the storage device when running this workload in isolation is about 210 IOPS.

1) Throughput allocation: Our first experiment was conducted to demonstrate that Arbitrator can guarantee per-application performance in metric of throughput. So all applications are specified with different throughput targets, and the same latency target and timelessness factor (their latency targets are specified with 1s and factor with 0). Throughput of application 1 and application 2 are set to 60 IOPS and 120 IOPS respectively. Show as Figure 2, the two applications achieve their throughput targets at the beginning of this experiment. When another application, whose throughput target is 40 IOPS, starts to access the storage system, it receives around 40 IOPS based on its proportion, and meanwhile throughput of application 1 and 2 drops to 60 IOPS and 110 IOPS respectively. Additionally, with Arbitrator scheduling, applications can achieve proportional sharing of throughput very quickly in such dynamic environment, seen from Figure 2. Similarly, when application 4 with 75 IOPS throughput requirement becomes active, it receives 55 IOPS, and performance of other applications has dropped correspondingly. Because that total throughput requirement of applications exceeded capability of underlying disk, all applications miss their throughput targets at last, but they proportionally share disk bandwidth based on their desired throughput.

2) Latency guarantee: Next, we evaluate that Arbitrator has ability of enforcing per-application latency guarantees. In this test, we also use the workloads de-

![Figure 2. Throughput allocation of Arbitrator.](image)
scribed as before, but we specified them with different latency requirement. Throughput of all applications was set to 210 IOPS and timelessness factor was set to 1. Latency for the four applications was set to 30 ms, 45 ms, 60 ms, and 90 ms respectively. For simplifying the presentation, we just present average response times obtained by each workload using Arbitrator algorithm, shown as Figure 3. In this experiment, we can get similar result as before. The first two applications achieve their latency targets at the beginning. Once last application started to access disk, latency of front applications increased and missed their deadline. As storage utility should provide fair service for all applications even it has limited service ability, latency for all applications has increased correspondingly.

(3) Latency and throughput guarantees: In this experiment, we validate that Arbitrator can guarantee per-application performance either in terms of latency or throughput even in overload situation, so different target metrics are set for these applications to validate that the scheduler is adaptive to various applications. The application 1 has throughput requirement with 50 IOPS and latency target of application 2 is 25 ms. The result presented in Figure 4 shows that application 1 obtained around 80 IOPS and average response times of application 2 is about 25 ms. To clearly present performance received by applications with different metric, Figure 5 shows the normalized violation values for these applications. These two applications meet their own performance target at the beginning, and violation value is very small. When application 3 with latency target of 40 ms is started, violation value of first two applications increased. However, these applications acquire nearly the same degree of deviating from their performance targets even they have different target metrics. Additionally, application 2 and 3 receive nearly proportional latency to their latency targets. When application 4 with requirement of 100 IOPS began to access the storage, violation value increased a lot for the limited service ability of device. During most time of the test, all running applications have nearly similar violation value, which means they can achieve proportional sharing of storage resource even in such dynamic environment.

(4) Timelessness effect: To better understand how timelessness requirement affects requests scheduling in our framework, we evaluate the scheduler with different timelessness factor values. The first two appli-

![Figure 3. Latency guarantees of Arbitrator.](image)

![Figure 4. Performance with different metrics.](image)

![Figure 5. The degree of SLO violation with Arbitrator.](image)
cations are specified with the same throughput and latency requirement (70 IOPS for throughput and 30 ms for latency targets). Their timelessness factors are specified with 0.1 and 0.9 respectively. Similarly, another two workloads started subsequently are specified with 100 IOPS for throughput and 60 ms for latency, and timelessness factors of the two applications are set to 0.6 and 0.4. Their average response times and throughput allocation are presented in Figure 6. The first two applications receive nearly the same average response times during first 100 second, and meet their latency targets. When another application began accessing the disk, the average response times of the first two applications increased. But response times of application 1 has exceeded 50 ms, while the application 2 still nearly maintain its latency targets, which is much lower than the first application even though they have the same latency targets and acquire the same response time at the beginning of this experiment.

Moreover, we see contrary result in throughput allocation in Figure 6(b). That is because application 2 has much larger timelessness factor and is much more sensitive to missing deadlines than application 1. Similarly, when the last workload sends requests to the disk, it raises latency of all other applications, and additionally, the gap of latency between last two applications is much smaller than first two workloads.

5. Related Work

The popularity of software-as-a-service (SaaS) has attracted significant research interests in providing latency guarantees and proportional throughput allocation in shared storage centers. So, this work can be classified into two categories: scheduling algorithms for proportional throughput allocation and latency guarantee.

5.1 Throughput Allocation

In order to provide fair sharing of storage bandwidth, existing research of QoS guarantees [12,13] in network area has been extended to storage systems. Most of these solutions are based on WFQ [12] or SFQ [13]. For example, SFQ(D) and FSFQ(D) [14] use virtual time based tagging to select I/Os and dispatch requests to underlying devices, and control the number of requests outstanding on devices to improve disk performance. Wang and Merchant [15] had extended SFQ(D) algorithm to guarantee proportional sharing of storage resources among clients in distributed environment, but it requires cooperation between storage servers and clients. Similarly, PARDA [4] employs SFQ(D) as local scheduler to allocate throughput proportionally for VMs within each individual host, and adjusts per-host queue lengths to guarantee proportional sharing of storage resource between hosts. Since storage throughput varies significantly with high-variance I/O characteristics, mClock [5] algorithm enforces proportional sharing of throughput while providing minimum throughput reservations and maximum throughput limits for VMs.

Different from above solutions that based on fair queuing algorithm, Fahrrad [11] and Argon [16] allocate disk head time to applications fairly to ensure fair sharing among applications and reduce interference among multiple access streams. In Argon [16], I/O requests from each application can only be dispatched during the appli-

![Figure 6. Performance with different timelessness factor.](image-url)
cation’s time slice. One issue behind of this approach is the potential of latency violation. The application that has not completed its requests during its timeslice needs to wait for next timeslice, which may result in timeout for the uncompleted requests.

5.2 Latency Guarantee

Instead of providing proportional sharing of throughput, some other algorithms are proposed to guarantee bounded response times for requests. These algorithms are usually based on Earliest Deadline First (EDF) algorithm. Façade [2] enforces latency guarantees for applications through combination of EDF scheduling and feedback controller scheme. It monitors request arrival rate and response time and adjust storage device queue length to achieve latency guarantees. AVATAR uses hierarchical I/O scheduling framework, where the high level takes charge of controlling requests dispatching rate and low-level scheduler is employed to guarantee latencies requirement. However, this scheme only guarantees latency requirement for applications whose request arrival rates do not exceed their specifications, and is liable to violate latency requirement for time-critical applications.

Rather than using EDF algorithm, SLEDS [7] enforces per-application latency bounds through heuristically throttling requests from overly-demanding applications. It maintains a leaky bucket for each application class, but this solution cannot make full use of spare storage resource since the storage service capacity varies dynamically with workload characteristics. To address this problem, Triage [3] employs an adaptive controller to throttles requests based on their latency observation and requirement. Also, some other solutions [1,17] are presented to provide latency guarantee for applications that are characterized by unpredictable high-variance burst requests. Decomposition algorithm [1] dynamically decomposes arrival stream into two portions based on the response time requirement and capacity parameters. One portion is guaranteed with bounded response times, while others are served in a best-effort fashion. Similar to SLEDS [7], this scheduling framework depends on the prior knowledge of system service capability, which fluctuates significantly and is difficult to pre-estimate.

Since above algorithms mainly focus on constraining I/O response times and ignore mechanical nature of hard disk drives, some research [18] studies the inherent trade-off between disk I/O efficiency and QoS guarantees. So, several real-time schedulers [19,6] are designed to provide latency guarantee while optimizing disk I/O efficiency via combination of EDF and SCAN algorithms.

6. Conclusions

In this paper, we have designed a novel scheduling framework that has the ability of providing per-application performance guarantees in terms of both latency and throughput. For supporting multi-dimensional QoS guarantees in consolidated storage system, we unify latency and throughput metrics. Also, to differentiate applications that have different timelessness requirement, a factor that reflects how much an application is sensitive to deadline missing is introduced in the framework. At last, we have proposed an approach to quantitatively evaluate how much applications deviate from their service level objectives when the shared storage system is overloaded, or how much applications gain from over-provisioned storage resource. Based on the analysis, we present and implement an interposed scheduling framework, which can accommodate several workloads with various characteristic. Instead of requiring prior knowledge of storage system capability, Arbitrator monitors the actual performance obtained by each application. Based on the degree of performance violation or performance windfall, the Arbitrator makes scheduling decision for all requests. Benefit from this approach, applications can make full use of spare storage resource to improve their performance, while achieving their desired performance requirements.

Acknowledgment

We would like to thank anonymous reviewers for their useful feedback. The work described in this paper was supported by the National Basic Research 973 Program of China (Grant No. 2011CB302301), the National
High Technology Research and Development Program of China (Grant No. 2013AA013203), the National Key Technology R&D Program (Grant No. 2011BAH04B02) and the National Science Fund for Distinguished Young Scholars (Grant No. 61025008).

References


Manuscript Received: Feb. 18, 2013
Accepted: Apr. 19, 2014