A Study on Performance Isolation Approaches for Consolidated Storage

Adrian Popescu and Saeed Ghanbari
Department of Electrical and Computer Engineering
University of Toronto
{adrian,saeed}@eecg.toronto.edu

Abstract

Due to maintenance and operational cost, production lines tend to consolidate storage servers which serve several workloads. In this context, it is essential to deploy mechanisms that isolate the performance of each individual workload. In this paper, we study several non-intrusive isolation approaches that are interposed between applications and the storage server. We evaluate the isolation offered by each scheme based on QoS guarantees, latency predictability and head seek overhead.

1 Introduction

In production environments, due to maintenance costs and power concerns, demand for consolidated storage is increasing. As a result, instead of using several storages for each individual application, a consolidated storage server can be shared among multiple workloads.

In this context, it is also essential that the performance of each individual workload to be isolated by other workloads such that they do not affect each other. Specifically, the goal of performance isolation is to provide each application with a virtualized view of the storage device in a transparent way such that the application has the illusion that it uses a dedicated peripheral.

However, due to mechanical arrangements in magnetic disks, providing performance isolation is non-trivial. For instance, I/O latency is highly unpredictable, which makes scheduling I/O requests inaccurate. As another instance of isolation implications, head seek overhead increases when multiple workloads are served, and as a consequence, workloads implicitly impact each other's performance.

In the literature, there are two classes of approaches for performance isolation in consolidated storage servers. One class of approaches integrates isolation targets into the server's resource scheduler. Integrated schemes make better utilization of the disk, but they are optimized for specific underlying disk controller, which make them inflexible for general purpose storage servers. On the other hand, the other class of isolation techniques interpose a scheduler between applications and the storage server. These are not intrusive and make little assumptions about the storage server. Such approaches have the advantage that they have more flexiblity and portablity. In this paper, we implement and evaluate interposed approaches for the consolidated storage problem, and compare the isolation provided by each approach based on QoS guarantees, latency predictability and seek overhead.

We first study QoS guarantees of each scheme. We start our analysis with schedulers that aim to ensure bandwidth isolation, and then we continue with approaches that aim to ensure latency. We also evaluate each scheme based on the latency predictability, i.e. how is the head seek time minimized so that a workload is minimally affected by other workloads. We finally evaluate seek overhead for each approach, i.e. the portion of I/O latency that is due to seek time.

We study Quanta-based, Lottery, Start-time Fair Queuing (SFQ) and leaky bucket as approaches that guarantee a fixed portion of disk bandwidth to each

workload, and Earliest Deadline First (EDF) and Façade as schemes that guarantee I/O latency.

The rest of the paper is organized as follows. Section 2 introduces the approaches that we implement and analyze in this paper. Section 3 presents the architecture of the system. Section 4 presents our methodology for analysis and evaluation. Section 5 evaluates and compare the scheduling schemes. A discussion follows in Section 6. We briefly share our experience of failed attempts in Section 7. Section 8 presents the related work. Finally, Section 9 concludes the paper.

2 Implemented Approaches

In this section we describe each isolation scheme that we implement and evaluate.

- **Quanta-based**: Quanta-based scheduler allocates a portion of disk bandwidth to each workload. Specifically, it gives each workload a specified quantum time during which it has the resource only for its dedicated usage. We implemented the quanta-based scheduler in two flavors: non work-conserving (Quanta) and work-conserving (Quanta-WC). The work conserving implementation (Quanta-WC) attempts to keep disk busy as long as there is outstanding I/Os. Therefore, if during its quantum a workload has no outstanding I/O, the scheduler switches to the next workload with outstanding I/Os.

- **Start-time Fair Queuing (SFQ)**: SFQ [1] schedules I/O requests of a workload based on its share of the disk. In effect, bandwidth is allocated proportionally to each workload in a work conserving manner. The main idea of SFQ(D)[2] is to set start-tags and end-tags for each request based on the I/O cost, on the share corresponding to the issuing application, and on the current virtual time. [1] In our implementation, we use a priority queue to store the requests sorted in the proper order. The I/O cost is dynamically approximated, by profiling. It is also worth mentioning that D represents depth or the maximum number of requests that can be dispatched to the disk at a time.

- **Earliest Deadline First (EDF)**: EDF guarantees latency for each workload. The basic idea is to prioritize I/Os with the shortest deadline. In order to implement EDF, we require the deadlines in terms of I/O request latencies for each application. Based on this information we compute the latency deadlines for each request. Then, we use a priority queue to sort requests based on their deadlines. Requests with the smaller deadlines are issued first. We implemented EDF in two flavors: non work-conserving (EDF - the default implementation) and work-conserving (EDF-WC). The first one defers issuing the request with the current smallest deadline if there is still sufficient time to be satisfied. In this way, other new requests with smaller deadlines can be prioritized even though they arrive later, at lower rates. The second one issues the request with the smallest deadline immediately.

- **Leaky Bucket**: Leaky Bucket uses a constant rate for dispatching requests from each workload. We implemented a simple throttling mechanism by specifying the active periods for the threads responsible for dispatching requests within each application; basically each application has a corresponding thread with a specified service rate.

- **Lottery-based**: The lottery-based scheduler is a simple implementation of the traditional lottery mechanism. Specifically, we give each application a share proportion for the disk resource and we are using a random number generator in order to select the next request to be dispatched.

- **Façade**: Façade [3] provides latency guarantees. The idea is to throttle individual I/O requests from multiple workloads so that the disk do not saturate, and each workload enjoy guaranteed latency. Façade consists of a controller which decides the maximum number of outstanding I/Os: D, and a non work-conserving EDF scheduler. Upon arrival, a request is scheduled to be issued to the

[1] Virtual time is a monotonically increasing function which reflects chronological order of events. In SFQ, virtual time is always the maximum of the start-tag of the latest dispatched I/O to disk, and the finish-time of the latest served I/O.

disk based on earliest deadline first. The scheduler then issues up to *D* requests to the disk. In this way, Façade controls the length of queue, I/Os waiting to be served by the disk, which determines the latency.

3 Architecture

The architecture on which we built the scheduling schemes is illustrated in Figure 1. As it can be seen, we used the Network Block Device (NBD) [2] module from the Linux kernel to map virtual disks on the client machines to the Gemini storage server. The storage server is situated on the machine hosting the physical disks and it is implemented within our research group. We implemented the scheduling schemes as Gemini modules and we interconnected them with the other modules as shown in the figure. In the following we describe the communication mechanism and the modules that we use, in more details.

3.1 Interfaces and Communication

Gemini clients, such as any multi-threaded application, use NBD-client for reading and writing logical blocks while the Linux virtual disk driver uses the NBD protocol to communicate with the storage server. As it can be seen in Figure 1 the client applications issue I/O requests to a nbd device (e.g. /dev/nbd1), which are then forwarded to the NBD module in the Linux machine. Next, the NBD module will forward the requests to the NBD processing-module on the storage server. When a request is received at the storage server, the *NBD module* converts the request into an internal representation format which is used further between Gemini modules. Finally, when the reply is received from the disk by the Gemini modules, before it is prepared to be sent to the client machine it is reconverted into the NBD format by the same *NBD module*. Then the reply is sent back to the client machine, which at its turn sends the reply back to the client application. The communication along the Gemini modules is explained in the next subsection.

[2]NBD is a standard storage access protocol similar to iSCSI which provides a method to communicate with a storage server over the network.

3.2 Gemini Modules

Gemini has a modular design which allows us to build many storage configurations by connecting modules together. All the modules are interconnected through a set of in-memory buffers and each module has several threads to process requests. Figure 1 shows a potential configuration for the Gemini modules. The bolded rectangle emphasizes the modules that we implemented. In this scenario, NBD processing-modules 1 and 2 receive requests from Application 1 and Application 2, respectively. The requests are forwarded to the statistics-module which keeps track of workload characteristics (incoming rate, throughput, latency etc), and then it sends the requests further to the scheduler module. Finally, the scheduler uses one of the scheduling scheme to dispatch the requests to disk. Next, we describe the role of each Gemini module in greater details:

- **NBD Processing module**: Is an entry point module which receives the I/O requests sent by the NBD client from the client machine. It is also the module that converts NBD packets into the Gemini internal protocol packets. This ensures the communication interoperability between different Gemini modules.

- **Disk module**: This is the module that is situated at the lowest level of the Gemini module hierarchy as it is providing the interface with the underlying physical disk. Basically, it is translating the application I/O requests to the virtual disk into the `pread()/pwrite()` system calls and reads & writes the underlying physical data. It is worth mentioning that the disk module does not implement any optimizations such as data caching; the OS buffer cache is also disabled by using direct IO.

- **Statistics module**: As already stated this module tracks the workload characteristics along the I/O request path. We modified the module to measure metrics of interests such as average throughput/response time or worst case throughput/response time.

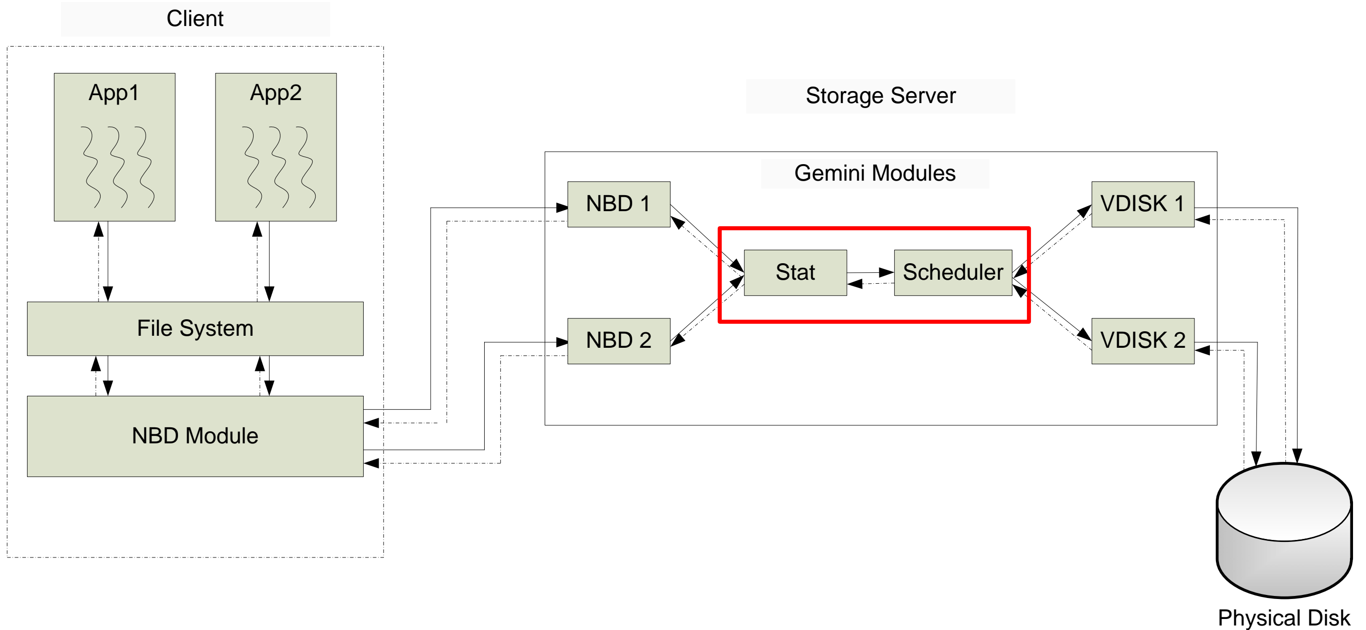

Figure 1. The System Architecture

- **Scheduler module**: The scheduler is implemented as a separate module interposed between the NBD and disk modules. Each scheduling scheme that we implement can be seen as a separate scheduling module which is interconnected with the other modules in a similar fashion as illustrated by Figure 1.

4 Methodology

In this section, we briefly review the metrics that we use to compare the studied isolation schemes. We evaluate and compare each scheme at the low-level by using micro-benchmarks. The evaluation is performed on three dimensions: *QoS guarantees*, *latency predictability* and *seek overhead*.

- **QoS guarantees**: This metric shows how well the bandwidth allocations or the guaranteed latencies are respected when multiple workloads are run concurrently on a shared physical disk. An ideal bandwidth/latency isolation ensures that the bandwidth allocation or latency for a workload is not affected when running other workloads on the same physical disk.

- **Latency predictability**: This metric shows how much the fluctuation in latency is imposed when multiple workloads concurrently share the same physical disk. Head seek time mainly account for latency fluctuation. For example, a sequential workload has low fluctuation because most of blocks are sequential and lay adjacent to each other on a same track. On the other hand, for random workload the head needs to switch between tracks frequently, and thus introduces unpredictable spikes in I/O latency. An ideal latency isolation ensures that a workload with mostly sequential requests, and hence low latency fluctuation, remains unaffected when it is sharing the bandwidth with other workloads.

  To measure latency predictability(fluctuation), we measure the coefficient of variation of latency[3] for a sequential workload (A) when it is running alone, and when it is competing with another se-

[3] Coefficient of variation is the standard deviation normalized by average: $\gamma(x) = \frac{stdev(x)}{mean(x)}$

quential workload, while both workloads have the same QoS guarantees.

- **Seek Overhead**: Head seek time accounts for the major performance overhead of disk latency. Operating systems and disk controllers have policies to optimize head seek time. Basically, when multiple I/O's are issued simultaneously, a disk controller sorts the requests in such a way that head seek time is minimized. We define the *relative seek overhead* (ρ) as the ratio of average I/O delay when a sequential workload shares the device with another running workload over average I/O delay when it is not sharing bandwidth with any other workloads. ρ indicates the efficiency of a scheme in terms of disk seek overhead (smaller ρ is desirable). In the ideal case $\rho = 1$, which indicates no seek overhead.

$$\rho = \frac{c^{shared}}{c^{dedicated}} \tag{1}$$

4.1 Benchmarks

We used two types of benchmarks to evaluate the scheduling schemes:

- **Sequential workload**: We used *dd* command from Linux to simulate sequential workloads. We configure the IO request size to 16K. In order to stress the disk, we used several instances of *dd* commands to generate a sequential workload.
- **Random workload**: To simulate random workloads, we used *load-generator*, a benchmark developed within our research group capable to stress the disk by issuing multiple simultaneous random IO requests to the disk. We configure the IO request size to 16K, and we setup the number of threads to the order of hundreds to ensure that we fully utilize the disk's capacity.

5 Evaluation

In this section, we compare the isolation schemes according to: how closely they meet their QoS guarantees, how well they minimize interference with regards to latency predictability, and finally we quantify each individual scheme based on the relative seek overhead.

5.1 Evaluation Setup

In the evaluation process we used a Linux Ubuntu SMP machine with 4 processors, 2GB of memory and a disk size of 8GB. The disk was modeled by a regular file having the specified size. The storage server and the nbd-clients were running on separate machines. As already mentioned in Section 4 we used *dd* and *load-generator* in performing our experiments. Specifically, we used 100 instances of *dd* that were reading directly the nbd-devices to simulate sequential workloads and we used the load-generator to simulate random workloads. We configured the load-generator to run a number of 379 simultaneous threads which were set to read IO requests of 16K in size from the nbd-devices. The idea for running multiple instances of dd and multiple threads for load-generator was to saturate the disk as much as possible. For the clarity of evaluation, in all the experiments we run only a maximum of 2 simultaneous workloads.

5.2 QoS Guarantees

As mentioned earlier, some isolation approaches guarantee bandwidth while others ensure latency bounds. In this section we compare approaches in each of these categories.

Table 1. Relative throughput for the share of 80%:20%

Approach	Sequential	Random
Quanta	79:21	79:21
Quanta-WC	**50:50**	**50:50**
Lottery	80:20	80:20
SFQ(D=1)	80:20	**51:49**
SFQ(D=32)	**54:46**	**50:50**

5.2.1 Bandwidth Guarantees

To evaluate bandwidth guarantees, we stress disk with two similar competing workloads, where one workload

Table 2. Absolute throughput values for the share of 80%:20% (I/Os per second)

Approach	Throughput (Sequential)	Throughput (Random)
Quanta	6850:1810	71:19
Quanta-WC	1140:1160	45:45
Lottery	7480:1880	72:18
SFQ(D=1)	478:119	46:44
SFQ(D=32)	677:582	46:45

has share of 80% of bandwidth, and the other workload has 20% of the share. Table 1 shows the relative bandwidth obtained for Quanta, Quanta-WC, Lottery and SFQ(D) schemes for both sequential and random workloads while Table 2 shows the absolute bandwidth values obtained for the same approaches. We see that Quanta and Lottery meet the bandwidth share guarantees. However, Quanta-WC and SFQ(D) fail to allocate the required bandwidth. Quanta-WC scheduler fails to provide fair bandwidth proportions because it resorts into a plain round-robin scheduling: it serves one workload and, as soon as the queue is empty, it starts serving the other workload even if the time allocated to the first workload is not exhausted. Because of the closed loop nature of workloads (individual requests are synchronized I/Os), within the short period of time passed from the beginning of the quantum of a workload, all requests in the respective queue will be issued to the disk, and the queue becomes empty. The workload would not issue any new I/O until some of the requests are served by disk. In such a case, Quanta-WC would switch to other workloads. The effect of that would be a worload can not use all its allocated quantum. Although, the work-conserving decreases idleness of the disk device, it fails to provide fairness and share guarantees. Moreover, the big difference in performance between the non-work conserving and work conserving can be explained in the unefficient resource utilization due to seek overhead as more explained in Section 5.4.

SFQ(D) provides poor bandwidth isolation for a different reason. SFQ(D) depends on an accurate estimation of each I/O latency even though this is highly unpredictable for random workloads. Although the average estimation is pretty stable, it has high variation,

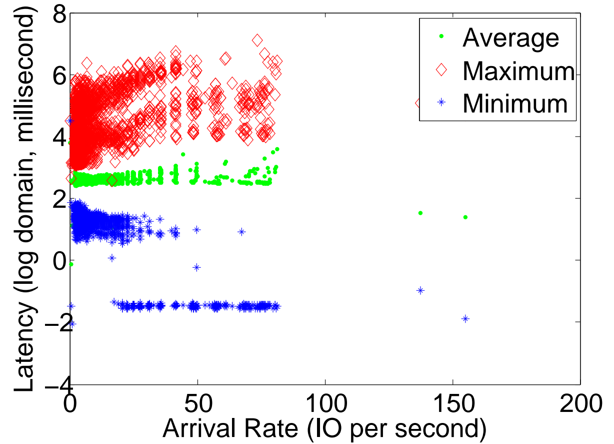

Figure 2. Random I/O latency has high variation, although average latency is stable (latency is plotted in natural Log domain)

see Figure 2, which results into the miscalculation of start-tag and finish-tag for each request within a flow. Hence, it determines inaccurate proportion allocation of requests per workload flow. We see that for sequential workloads which have low variation in latency, SFQ(D=1) respects the proportions, while for random workloads it fails to respect the share. For the case of SFQ(D=32), sequentiality of the workloads is lost because SFQ(D=32) issues up to 32 requests simultaneously. Thus the effective pattern of I/Os observed by the disk is a random workload containing requests from both workloads.

5.2.2 Latency Guarantees

To evaluate latency guarantees, we stress the disk with two random workloads with the same intensity. Work-

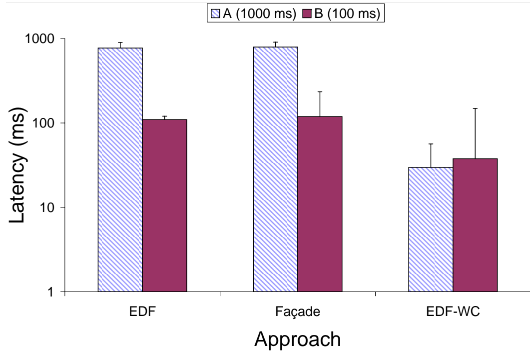

Figure 3. Latency guarantees

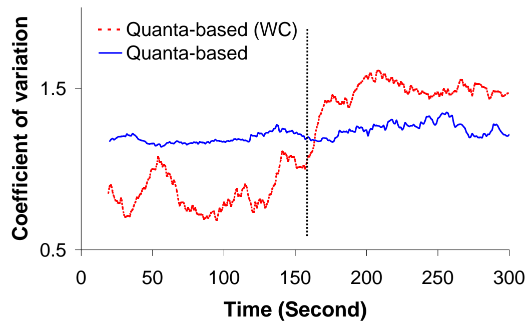

Figure 4. Latency predictability

load A has guarantee of 100ms and workload B has guarantee of 1000ms. We see in Figure 3 that latency delivered by EDF and Façade exactly matches the guaranteed latency. EDF-WC provides lower latency but does not respect the latency guarantees. In fact, EDF-WC does not provide fairness but makes better utilization of the disk.

We noticed that Façade resorts into EDF scheduling most of the time. In fact, the real time controller of Façade shrinks the device queue size to zero which effectively results in scheduling I/O requests based on their deadlines and issuing them to disk only when their deadline is passed. When the device queue size (the size is determined by the controller) is non-zero, Façade behaves just like EDF-WC as most I/Os are issued to the disk before their deadlines are actually passed. However, sporadic spikes in latency makes the controller shrink the device queue size to zero, resorting to the plain EDF scheduling.

5.3 Latency Predictability

We compare isolation schemes in regards to the degree that competing workloads affect each other in terms of fluctuations in latency. An ideal latency isolation ensures that the latency fluctuation of a workload is not affected by other workloads on the same physical disk.

In order to measure the latency predictability we computed the coefficient of variation of latency, as specified in Section 4. We quantify how is the average coefficient of variation modified for a workload that runs in isolation, when a second workload is started. Table 3 shows these values for all the studied approaches. We observe that except Quanta scheduling which maintains its coefficient of variation constant, other approaches have a high degree of fluctuation when two competing workloads share the same physical disk.

As an illustration, Figure 4 shows the coeficient of variation of a sequential workload using Quanta and Quanta-WC. A competing sequential workload is started after 150 seconds from the beginning of the experiment. As we can see, latency variation increased using Quanta-WC, while with Quanta approach it remains unaffected. This is due to the strong isolation properties of the Quanta approach, where only a single workload is served during a particular quanta.

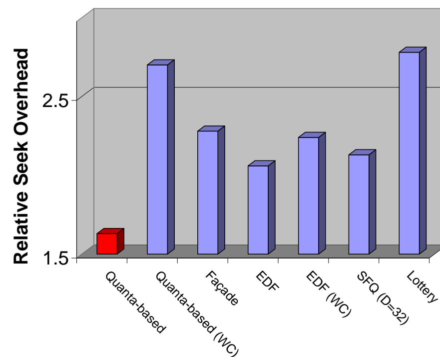

Figure 5. Relative Seek Overhead

Table 3. Coefficient of variation of latency

Approach	Coefficient of variation (Isolation)	Coefficient of variation (Interference)
Quanta	1.18	1.25
Quanta-WC	0.86	1.49
EDF	0.14	1.19
EDF-WC	0.08	2.07
SFQ	0.05	1.08
Lottery	0.85	1.40

Table 4. Effect of seek overhead on a sequential workload.

Approach	Latency (Isolation) [ms]	Latency (Interference) [ms]	Seek overhead
Quanta	15.00	24.60	**1.63**
Quanta-WC	9.41	25.40	2.70
EDF	16.50	34.10	2.06
EDF-WC	12.80	28.60	2.24
SFQ(D)	37.90	80.60	2.13
Lottery	8.74	24.30	2.78

5.4 Seek overhead

Table 4 shows the effect of seek overhead for all the studied scheduling approaches. The overhead is computed as the penalty introduced in the average latency of a sequential workload when a competing sequential workload is started, as specified in Section 4. All the studied schemes bear some degree of seek overhead. However, as shown in Figure 5 the lowest seek overhead belongs to the Quanta scheduler. This is achieved because within a quanta time, only requests from one flow are served; therefore, the disk head stays local within the regions belonging to that flow. We also observe that the worse relative seek overhead belongs to Lottery and Quanta-WC schedulers. This indicates that the I/O requests from the two workloads are highly interleaved, and hence it incurs high seek overhead.

6 Discussion

6.1 Work Conservation vs. Isolation

Isolation schemes can be categorized in work-conserving (WC) and non-work-conserving (NWC) approaches. WC is intended to avoid waste of the disk bandwidth capacity. Thus, it tries to keep the disk busy as long as there are outstanding I/Os. While this is a good approach for resources such as CPU, keeping the disk device busy all the time does not necessarily results in better performance. Disk mechanical settings, especially head seek times, play a major factor in performance and utilization of the disk device. In most cases, it is favorable to use NWC approaches that sacrifice disk utilization for better performance gains [4], and better isolation.

Moreover, NWC methods hide complexity and unpredictability of storage devices. For example, Quanta scheduler allocates bandwidth without needing to know about requests or I/O latency. In contrast, WC approaches, such as SFQ(D), are substantially dependent on accurate estimation of I/O latency or arrival rates for each workload. Moreover, with prevalence of disk arrays, heterogeneity of disk controller frameware and diversity of OS level I/O scheduling, it is improbable to derive an accurate model that can capture the complexity of storage devices. Therefore, we believe that deploying NWC schemes that hide complexity are an inevitable choice for isolation.

6.2 Hard vs. Soft Isolation

Isolation approaches can be categorized as *hard isolation*, which strictly isolates one workload from other competing workloads, and *soft isolation*, which provides isolation as long as disk is not saturated. For example, EDF-WC provides latency guarantees as long as disk service time is less than arrival rate. If one workload issues excessive requests, it can affect others. In this context, EDF, EDF-WC, Quanta-WC, Lottery, Façade and SFQ(D) can be categorized as soft deadline providers, while Quanta, and Leaky Bucket deliver hard isolation. In fact, strict isolation stems from the non work conservative nature of these schemes.

6.3 Detailed Analysis

In this section we analyze each studied approach in details:

- **Quanta-based.** Quanta-based scheduling guarantees bandwidth allocation by slicing quantum time of device service based on flow's share. However, selecting right quantum size is non-trivial. Allocating large quantum size reduces seek overhead because accesses during the quantum interval would be from one flow and most probably locally distributed on disk. Especially sequential streams would benefit more by larger quantum size. However, large quantum size incurs higher latency variation and the worst case latency will increase.

- **Earliest Deadline First.** An EDF based scheduler guarantees latency by issuing I/Os to the device according to the deadline of each request. Although it exhibits good latency predictability, it fails to provide guaranteed throughput, fairness, and hence strong isolation. It also incurs high disk head overhead. The other issue with EDF based scheduling is that deadlines should be determined by the application which issues I/Os. In fact, commercial applications usually do not provide such fine grain deadlines, mainly because it is non-trivial to map high-level objectives to low level I/O deadlines.

- **Start-time Fair Queuing.** SFQ achieves latency predictability as well as proportional bandwidth allocation. It also provides strong fairness. However, the main limitation of SFQ is its reliance on an accurate model for the I/O cost which was shown to be difficult to estimate. Moreover, SFQ overlooks the cost of head seek overhead.

- **Lottery.** Unlike aforementioned schemes that are deterministic, lottery scheduling is a probabilistic scheme for providing isolation. This scheme suffers from high latency variation, potentially unfairness, and high seek overhead.

- **Leaky Buckets.** Leaky buckets are in fact a mechanism to control throttling rate of I/O requests to the storage device. Leaky bucket approach fails to address latency predictability. The schemes is inherently non-work conserving which can result in waste of resource if throthling rate is not configured properly. For example, a sequntial I/O might benefit substantially if issuing rate is high to decrease head seek overhead. Our experience showed that finding a proper throthling rate is non-trivial.

- **Façade.** Façade combines EDF and EDF-WC to provide guaranteed latency, and isolation. However, approaches based on earliest dead line first offer weak isolation when one workload misbehaves, issuing many requests. The only mechanism of Façade to control disk saturation is to determine maximum device queue length, which turns out to be insufficient to isolate well-behaved workloads from a misbehaved workloads. Façade also incurs high seek overhead and offers low latency predictability.

7 Dead-end Trials

7.1 Modeling latency

We have tried to model disk latency by M/M/m queuing model. M/M/m model assumes that arrival rate is Poisson, service time is exponential and we have m threads that service I/O requests. Initially, we were

suspicious about these assumptions, especially about the Poisson arrival rate. However, we were assuming that in a real system, because of multiple levels of cache (either user or storage), arrival pattern to the disk deviates significantly from an interactive system (close loop queue model).

However, the benchmarks and tools that are available simulate a close loop request pattern. Besides that, modeling m threads that serve I/Os, as m independent servers is not a correct assumption. In fact, service time per thread is totally dependent on other threads. If number of requests in the queue is less than m, service time per thread would be different from the case that queue size is larger than the number of threads. As a result, we could not find a stable pattern to relate request arrival rate to service time, which is an essential piece of information for any queuing model. We gave up this idea.

7.2 Leaky Bucket Approach

As already stated in Section 2 we also implemented a leaky bucket scheduling approach. However, due to its poor performance and its uniform behavior determined by the trottling mechanism we did not include the results in our evaluation. Not only that it generates a poor utilization of the disk resource but also it suffers from high latencies and low throughput. In conclusion, we consider a leaky bucket approach inappropriate for disk scheduling.

8 Related Work

As we have already seen there are two flavors for the consolidated storage problem: interposed solutions, which interpose a scheduler between the application and the underlying storage server, and integrated solutions, which integrate the QoS and isolation requirements of applications into the storage scheduler.

Interposed solutions The Service Level Enforcement Discipline for Storage (SLEDS) [5] is a network adapter which controls I/O throttling through a leaky bucket filter. SLEDS ensures that all service level objectives are met, as long as enough aggregate resources exist. It accomplishes that by taking periodic performance samples, and by throttling I/Os from overly-demanding clients whenever other clients experience inadequate performance. The main drawback of the SLEDS is that it is not work conserving. Therefore, in the case of surplus resource, leaky bucket approach enforces pre-configured throttling. Moreover, the SLEDS system does not consider admission control.

Façade [3] provides predictable response-time through a combination of real-time scheduling and feedback-based control of the storage device queue. Façade scheduling is based on Earliest Deadline First (EDF) IO scheduler and it combines a priority control scheduling with an admission control scheme to dispatch requests of each workload class in such a way that they are serviced according to predefined latency. Façade provides QoS guarantees, but it does not isolate request flows from unexpected demand surges by competing flows. It also fails to guarantee fair bandwidth proportion allocation.

Jin et al. propose in [2] three algorithms based on Start-time Fair Queuing (SFQ) [1]: min-SFQ(D), SFQ(D), and FSFQ(D). SFQ is shown to be fair for fluctuating service capacity. SFQ assigns two tags to each request from a flow: a start-tag and a finish-tag. The tag values represent the the notion of virtual time which in the general case is computed as the minimum start time of the latest dispatched request. Requests are dispatched in increasing order of the start tag. The way the three algorithms presented differ relates to the way the virtual time is computed and to the level of fairness achieved for different flows. For instance, min-SFQ(D) sets the virtual time to the minimum start-tag of the recently dispatched requests. This solution may result in unfairness toward more aggressive flows for the benefit of less aggressive flows. On the other hand, SFQ(D) computes the virtual clock from the start-tag of the out-standing requests, those requests arrived but not yet dispatched. This achieves better fairness than min-SFQ and don't penalize applications which hold more than their share on the disk concurrency window. Finally, Four-tag Start-time Fair Queuing (FSFQ(D)) is an improved version of SFQ(D) which gives extra credit to the late arrival requests to make up for their

lateness. The main drawback of these three algorithms is that they assume the cost of I/O known, about which we know that is difficult to predict and is still an open issue.

Zhang et al. [6] propose a two-level scheduling based on decoupling response time and throughput. SARC, the scheduler on the top level, is a credit-based rate controller which controls throughput. AVATAR, the scheduler at the lower level is an Earliest Deadline First (EDF) scheme, which is mainly responsible for satisfying latency objectives. The promising aspect of this scheme is that it targets both latency and throughput. The main drawback of this work is that it is evaluated only through simulations.

Argon [7] is a shared storage server which limits cache and disk interference between workloads encountered in traditional systems. The main goal of Argon is to provide each service with at least a specified fraction (e.g. R-value=0.9) of the throughput it obtains when it has the storage sever only to itself, within its share of the server. Moreover, when resources become scarce and not suffice for a new workload, the system is not scheduling new workloads in order to ensure that the rest of the workloads have enough resources to achieve their assigned efficiency. Argon tries to insulate each workload by combining three known techniques in a novel way: aggressive amortization, cache partitioning and quanta-based scheduling. One of the drawback of Argon is that is using throttling in order to reduce the level of concurrency at the disk level. Moreover, in some situations it has to drain out requests in order to ensure quantum limits.

Integrated solutions Cello [8] is a disk scheduling framework which support applications with diverse performance requirements. It has a two-level disk scheduling architecture: a class-independent scheduler and several class-specific schedulers. Moreover, Cello allocates disk bandwidth at two time scales: the class-independent scheduler controls the coarse-grained bandwidth allocation to each application class and the class-specific schedulers controls the fine-grained interleaving of the requests from application classes in order to ensure the required performance guarantees. Cello ensures fairness across multiple-class applications and is work conserving. Moreover, it avoids performance interference of application classes. However, one of the drawbacks is that Cello requires performance models in order to estimate the service time for I/O requests. There may be the case that we don't have all the underlying system details in order to get a reliable performance model.

Stonehenge [9] is a multi-dimensional storage virtualization capable to virtualize a cluster-based storage system along multiple dimensions such as: bandwidth, capacity and latency. In other words, Stonehenge can multiplex multiple virtual disks, each with different attributes on a single physical storage system as if they are on separate physical disks. Stonehenge propose a new disk scheduler (CVC) and an admission control algorithm (MBAC) able to exploit the multiplexing of the input loads in order to improve the number of virtual disks that can be admitted while still supporting guarantees on bandwidth and delay bound. CVC is real-time disk scheduler capable to maximize disk utilization and at the same time is able to satisfy the performance guarantees for the virtual disks. The achieved efficiency is up to 80% better than generic real time schedulers. On the downside it uses a centralized storage manager which can become a performance bottleneck in large scale systems.

YFQ [10] is a disk scheduling algorithm implemented as part of the Eclipse/BSD operating system. Experiments show that YFQ can guarantee file accesses with a high throughput, low delays and fairness between different workloads. Because the QoS guarantees to individual application can hinder disk scheduling optimization, the authors evaluate several disk enhancements: batching, overlapping, tie-breaking and fragmenting, in order to give YFQ an aggregate disk throughput, approaching that of FreeBSD conventional scheduler. Experiments show that these enhancements are useful for YFQ and may also be useful for other disk scheduler algorithms.

9 Conclusions

In this paper, we implemented and evaluated several isolation approaches for consolidated storage servers. We analyzed isolation offered by each approach based

on QoS guarantees, latency predictability, and relative seek overhead. We showed that there is a trade-off between work-conservativeness and isolation. We believe that non work conserving approaches are better choices for isolation of workloads sharing same physical storage devices, as such approaches offer better QoS guarantees and hide complexity.

References

[1] P. Goyal, H. M. Vin, and H. Cheng, "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks," *IEEE/ACM Trans. Netw.*, vol. 5, no. 5, pp. 690–704, 1997.

[2] W. Jin, J. S. Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," in *SIGMETRICS*, E. G. C. Jr., Z. Liu, and A. Merchant, Eds. ACM, 2004, pp. 37–48.

[3] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Façade: Virtual storage devices with performance guarantees," in *FAST*. USENIX, 2003.

[4] S. Iyer and P. Druschel, "Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 117–130, 2001.

[5] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee, "Performance virtualization for large-scale storage systems," in *SRDS*. IEEE Computer Society, 2003, pp. 109–118.

[6] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel, "Storage performance virtualization via throughput and latency control," *Trans. Storage*, vol. 2, no. 3, pp. 283–308, 2006.

[7] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: performance insulation for shared storage servers," in *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2007, pp. 5–5.

[8] P. J. Shenoy and H. M. Vin, "Cello: a disk scheduling framework for next generation operating systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 44–55, 1998.

[9] L. Huang, G. Peng, and T. cker Chiueh, "Multi-dimensional storage virtualization," in *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2004, pp. 14–24.

[10] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz, "Disk scheduling with quality of service guarantees," in *ICMCS, Vol. 2*, 1999, pp. 400–405.