An Adaptive Mechanism for Fair Sharing of Storage Resources

Chao Jin and Rajkumar Buyya
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
{*chaojin, raj*}*@csse.unimelb.edu.au*

***Abstract*—To ensure Quality of Service (QoS) for data centers, it is critical to enforce a fair share of storage resources between competing users. Interposed schedulers are one of the most practical methods for performance isolation. Most fair queuing-based proportional sharing algorithms for existing interposed scheduler are variants of counterparts designed for network routers and may result in breaking the fairness of proportional sharing required by Service Level Agreements for storage systems. This paper presents a novel algorithm to address this problem. As an extension of the fair queuing-based algorithm, it can dynamically adapt to the performance variation of storage systems and guarantee a fair sharing of resources as well as satisfying the minimal performance requirements for different clients. The design and performance evaluation are presented.**

***Keywords*-Storage; Quality of Service; Fair Sharing;**

I. Introduction

Data centers can not only consolidate large scale of data, but also provide the benefits of statistical sharing and lower management costs with better resource efficiency [1]. With consolidation and resource sharing, it is a challenge to control the interference between competing users, i.e., isolate performance perceived by different clients or applications. Normally, resource reservation with Service Level Agreements (SLA) [2] may specify performance requirements, in terms of absolute values, for different classes of users. The storage utility within data centers should statistically ensure the required QoS defined in the SLA. The goal of this performance virtualization (PV) is to make each client experience as if a subset of physical resources were dedicated to it, as long as enough aggregated resources exist.

This PV can be supported at various levels within the storage system. One of the most practical methods is interposing a QoS scheduler between clients and storage utilities [3]. The interposed scheduler treats the storage utility as a "black box". This method possesses several advantages: i) it uses little information about the implementation of storage service; ii) it can be used for existing storage services which have no internal QoS supports; iii) it does not interfere with operations of existing storage services. These advantages make it applicable to a wide range of storage services.

Interposed schedulers aim to meet the minimal performance requirements for each client, and ensure the absolute fairness of spare resources sharing between competing clients. Many variants of fair queuing-based scheduling algorithms [4] [5] [6] have been proposed to satisfy both requirements.

Fair queuing-based proportional sharing algorithms, are originally designed for network routers to dispatch packets. They assume that the process time required for dispatching each packet to the outbound link is uniform and same [7]. The counterpart of dispatching packet in disk-based storage system corresponds to reading or writing unit data on disks. Unfortunately, modern disk systems cannot support uniform access time for each sector and the dynamic seeking latency of disk head makes the variation of performance more complicated [8]. This generates a natural unfair access time for competing clients, and it eventually makes existing fair queuing-based proportional sharing algorithms hard to meet the requirements of SLA (Section II-C).

To overcome this problem, we propose a novel algorithm as an extension of SFQ(D) [5]. Our new algorithm provides fair resources sharing defined by SLA for competing users with automatic adaptation to performance variation of disk-based storage systems. We evaluate the proposed algorithm both analytically and experimentally on a distributed storage system. The experimental results confirm that our algorithm can allocate resources with stringent performance and fairness requirements defined by SLA even under fluctuating disk performance.

The remainder of this paper is organized as follows. Section II reviews related work and identifies the problem of unfairness. Section III presents our new algorithm and describes its design and implementation in details. Section IV provides a performance evaluation in real systems. Section V concludes the paper.

II. Overview and Related Work

PV is critical for data centers that share resources to many users with various requirements. To achieve this target, an interposed scheduler (IS) is always used to isolate performance perceived by clients. In practice, the IS intercepts requests from clients and forwards them to storage services, while the PV is gained by throttling (delaying) requests before forwarding them.

Façade [3] implements an EDF (Early Deadline First) algorithm in an IS with proportional feedback to control the length of disk queue. It aims to meet the real time requirements of disk IO scheduling. However, it needs a

rudiment model of the storage system. Triage [9] adopts a control theory to predict the system performance and correspondingly adjust its system model for performance isolation and differentiation. Its system model is not sensitive to the performance dynamics perceived by concurrent clients due to different physical data position.

Different from the above methods, proportional sharing algorithms are proposed to enforce fairness between competing users. Proportionally sharing processors and network resources has been deeply investigated and many implementations [10] [7] [4] are based on the fair queuing algorithm. To address the similar problem in the domain of storage, various adaptations of these algorithms have been proposed. For example, YFQ [11], SFQ(D) [5] are based on Start-time Fair Queue (SFQ) [12]; SLEDS [13] and SARC [14] are adaptations of the leaky bucket algorithm; CVC [15] [4] adopts VirtualClock [7]. However, directly applying fair queuing-based algorithm to storage system can introduce unfairness within a short period of time (Section II-C).

In the remainder of this section, we present the basic model of IS, review the Start-time Fair Queue algorithm and analyze its unfairness in storage systems.

A. Interposed Scheduler

Given a shared storage service, requests from clients are grouped into different classes, called *flows*, and a *service level objective* (SLO) is defined for each flow. According to the required minimal resource, each flow is assigned with a *weight* that represents the proportion of reserved resources. Without loss of generality, each flow may consist of requests from multiple clients, and requests from one client/process may be grouped into multiple flows.

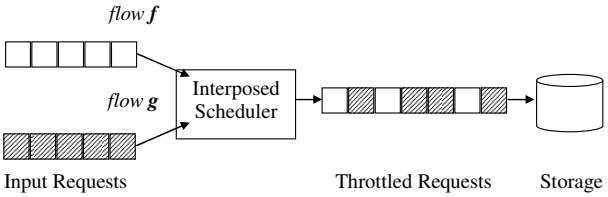

Figure 1: **Architecture of Interposed Scheduler.**

The architecture of IS is illustrated in Figure 1. All requests of every flow are intercepted by the scheduler before forwarding them to the storage service. Currently, many large scale storage services are achieved by a cluster of disk bricks [16]. The scheduler shares the storage resources proportionally according to the weight of each flow. This is achieved by reordering requests of all flows to meet the minimal required quality of storage service.

Without loss of generality, we use the principle of fair queue to analyze the model of fair queuing-based IS. Each flow consists of a sequence of requests: $r_f^0 \ldots r_f^i \ldots$ Each request is associated with a cost, $Cost(r_f^i)$; in particular, the cost may be the size of required data or the number of I/O requests per second. The weight of each flow f is denoted ϕ_f. $W_f(t_1, t_2)$ represents the aggregated cost of requests for flow f within the time period between t_1 and t_2. Therefore, unfairness $U(t_1, t_2)$ is defined as follows:

$$U(t_1, t_2) = \max_{f,g} \left| \frac{W_f(t_1, t_2)}{\phi_f} - \frac{W_g(t_1, t_2)}{\phi_g} \right| \quad (1)$$

DEFINITION 1. A service disciple is fair, if U is a small constant independent of the length of the time interval [17].

B. Start-time Fair Queuing Algorithm

Start-time Fair Queuing (SFQ) is a representative example of fair queuing-based scheduling algorithm. Compared with other fair queuing-base algorithms, SFQ enforces a more stringent fairness between competing flows [12]. Like most fair queuing algorithms, SFQ assigns tags to each request when it arrives, and dispatches requests in a non-descending order of tags while ties are broken arbitrarily.

Given a request r_f^i of flow f, SFQ assigns it with two tags: a start tag, $S(r_f^i)$ and a finish tag $F(r_f^i)$. These tags correspond to the time at which the request should start and finish according to a system maintained virtual time, $v(t)$.

The tags assigned by SFQ are defined as follows:

$$S(r_f^i) = \max\{v(A(r_f^i)), F(r_f^{i-1})\}, (i \geq 1). \quad (2)$$

$$F(r_f^i) = S(r_f^i) + Cost(r_f^i)/\phi_f, (i \geq 1). \quad (3)$$

$A(r_f^i)$ is the arrival time of request r_f^i. Initially, we have $F(0) = 0$, $v(0) = 0$. Within a busy period, $v(t)$ is defined to be equal to the start tag of the packet in service; at the end of busy period, $v(t)$ is set to be the maximum of finish tags assigned to any packets that have been serviced by time t. SFQ dispatches requests in an increasing order of start tags.

The fairness property of SFQ is presented by Equation (4). For any time interval $[t1, t2]$ in which flow f and g are backlogged, the unfairness is up to a predefined limitation.

$$\left| \frac{W_f(t_1, t_2)}{\phi_f} - \frac{W_g(t_1, t_2)}{\phi_g} \right| \leq \left(\frac{Cost_f^{max}}{\phi_f} + \frac{Cost_g^{max}}{\phi_g} \right) \quad (4)$$

where $Cost_f^max$ and $Cost_g^max$ are the maximum request cost for flow f and g respectively.

SFQ have been proved to provide a statistical fair service even when service rate fluctuates frequently due to random effects, such as variability in capacity of links and CPU process rate.

However, in storage systems, the fluctuation of the service rate is not restricted to random causes. More importantly, locations of data assign natural unfairness to each flow. This type of non-random fluctuation of service rate is not covered by SFQ.

C. Unfairness of SFQ for Storage System

Although SFQ ensures a fair sharing of storage in a long term, it may cause unfairness in a short term. This is not accepted by storage-bound applications [4]. The latency of many such applications is around several hundred milliseconds [2]. This section presents the reason of short term unfairness of SFQ by using examples with dynamic bandwidth.

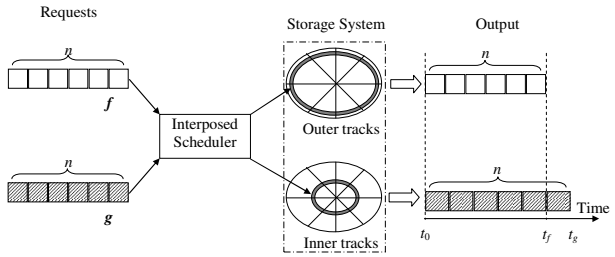

Figure 2: **Unfairness of SFQ(D) for Storage Systems.**

During the life cycle of a data center, it incrementally equipped with different types of disks, which provide heterogeneous performance. Even for the same type of disks, outer tracks can provide a much higher access rate than inner tracks. The performance variability in disk-based storage systems is mainly caused by the bandwidth difference between inner and outer tracks of disks. For modern disk devices, the increases in areal densities of magnetic disks have led to a bandwidth difference of 60% or more between inner and outer tracks [18]. This gap is expected to grow continuously in the future [19]. Moreover, caches at various levels of the file systems and disks cannot totally mask this difference. According to an IO report of Microsoft [20] about the performance perceived at the *application level* in Windows system, the outer sectors are more than 30% faster than inner sectors even with disk cache enabled. As a consequence, given a flow, the track position of its data has an overwhelming impact on its performance perceived by clients. This type of performance variance bound with flows is not fully covered by SFQ.

SFQ cannot be applied directly to the storage system. An extension of SFQ, called SFQ(D) [5], maintains a queue for servicing up to D concurrent requests. With this extension, during a busy period, the virtual time is set to be the maximum start tag of requests in service. The unfairness of SFQ(D) algorithm for disk systems is illustrated in Figure 2 with a simple example.

Assume that two continuously backlogged flow f and g have same weights and the depth of schedule queue is long enough. Both f and g send n requests with same cost to the IS nearly at the same time t_0, and the cost of each request is L. The data requested by f and g are located on different disks. Furthermore, the data requested by f are on located in the outer track and the data requested by g are on the inner track. (Without loss of generality, both disks can be attached to the same machine or different machines connected by high speed networks.) Without comprising the correctness of analysis, we assume that the overhead of SFQ(D) algorithm can be ignored with comparison to the time required by disk I/O access. Therefore, it is reasonable to expect that the requests of f can be finished before g, because access rate on outer tracks is faster than that on inner tracks. Assume f is finished at t_f and g is finished at t_g, $t_f < t_g$, while the service rates of inner and outer tracks are B_i and B_o respectively. The unfairness between f and g is as follows:

$$|\frac{W_f(t_1,t_2)}{\phi_f} - \frac{W_g(t_1,t_2)}{\phi_g}| = C * (t_f - t_0) \qquad (5)$$

where $C = (B_o - B_i)/L$.

From equation (5), the unfairness between f and g definitely depends on the time interval. Therefore, it conflicts with the fairness requirement of resource sharing according to Definition 1. Although the above analysis is based on the unfairness caused by dynamic disk bandwidth, it is also applicable to the case of unfairness that is caused by the dynamic seeking latency of disk heads.

III. Design of Adaptive SFQ

This section describes an adaptive method to address the unfairness problem of applying SFQ algorithm to proportionally share the resources of disk-based storage system. For a data center that is built from a number of small storage servers/bricks, a SLA planner [1] is assumed to take the responsibility to specify SLO contracts with different users and then allocate storage resources (including storage space and bandwidth) across distributed machines/disks according to the requirements. Our algorithm focuses on enforcing fair sharing of bandwidth resources on each machine or disk. To enforce fair sharing of resources across multiple distributed resources is out of the scope of this paper.

The unfairness caused by SFQ in disk-based storage systems is due to the fixed forwarding speed of requests for each flow, which is determined only by their weights. We propose an adaptive method to address the unfairness problem of applying SFQ algorithm to proportionally share the resources of disk-based storage system. Different from SFQ, ASFQ (Adaptive SFQ) takes into account both the weights and the real time performance perceived by each flow to decide the forwarding speed for their requests. In particular, a performance monitor is used to observe the real time performance perceived by each flow and a feedback mechanism is deployed to dynamically adjust the forwarding speed of requests according to the status of each flow observed by the monitor.

Our algorithm aims to meet the QoS requirements defined in SLA. AFSQ reserves a minimal capacity share for each flow proportional to its weight; surplus resources are shared between flows with outstanding requests also in proportion to their weights.

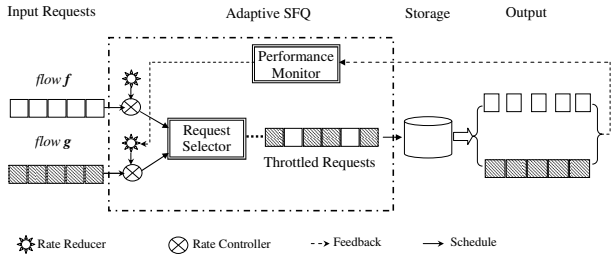

Figure 3: **Principle of ASFQ.**

A. Principle Architecture of ASFQ

Our adaptive method is achieved by the cooperation between a *performance monitor* (PM) and a *request selector* (RS), as illustrated in Figure 3. Specially, the PM collects the most recent statistics of resources consumed by each flow. RS is responsible to choose the next proper request from all the flows with pending requests. The request selected to be the next forwarding request depends on the *rate controller* of each flow. For each flow, its rate controller is defined to be the combination of the start tag of the head request and the negative compensation of its *rate reducer*. Normally the rate reducer is inactive. In case any unfairness is found by the PM, the rate reducer is triggered to slow the forwarding rate of flows that consume more resources than what they deserve. It is designed to make the forwarding speed slow down to a level which does not conflict with the fairness while still meeting the requirements of SLO. Essentially, the PM and the rate reducer forms a feedback mechanism to adjust the forwarding rate with adaptation to the variance of performance. If no unfairness is found, the rate reducer remains inactive and therefore the ASFQ behaves as a normal SFQ(D) algorithm.

B. Feedback Mechanism of ASFQ

Performing an efficient and effective feedback is critical for ASFQ. The feedback is determined by two issues: the accuracy of monitored performance and a negative compensation used in the rate reducer.

The PM collects consumed resources for each flow periodically. Assume that the collection period of PM is T_m and T_m^p represents p-th period (p=0, 1, 2, ...). $W_f(T_m^p)$ stands for the aggregated cost for flow f within T_m^p, i.e., the aggregated resources consumed by flow f. The maximal unfairness permitted in SLA is ε. That means if $U(T_m^p)$ is larger than ε, the feedback mechanism should start to adjust the forwarding rate for the flows which receive more resources than what they deserve. We call this type of flows CMTD (Consume More Than Deserve). On the contrary, those flows that consume less than what they deserve are called CLTD (Consume Less Than Deserve).

To adjust the forwarding rate for CMTD flows, we calculate a negative compensation factor, Δ. It is used to decrease the priority of head request of CMTD flows. Given T_m^p and two flows, f and g, if $U_{f,g}(T_m^p)$ is larger than ε, ASFQ aims to ensure in the next period, a compensation should be paid to CLTD flows by delaying requests of CMTD flows. This means that we have to enforce $U_{f,g}(T_m^{p+1})$ to be less than ε. Assume that f is a CMTD flow and g is a CLTD flow. Therefore, to limit the unfairness in T_m^{p+1} in an ideal case, we have the following objective:

$$|\frac{W_f(T_m^{p+1})}{\phi_f} - \frac{W_g(T_m^{p+1})}{\phi_g}| = 0 \tag{6}$$

We already know that, in the last collection period:

$$|\frac{W_f(T_m^{p+1})}{\phi_f} - \frac{W_g(T_m^{p+1})}{\phi_g}| = U(T_m^p) \tag{7}$$

Since we aim to limit the short term unfairness, it is reasonable to expect that the pending costs of g in T_m^{p+1} are nearly the same as in T_m^p. Therefore, as a combination of equation (6) and (7), we have the following:

$$\Delta^{p+1} = W_f(T_m^{p+1}) - W_f(T_m^p) = U(T_m^p) * \phi_f \tag{8}$$

The result of equation (8), Δ, is the negative compensation for flow f. It is used to decrease the priority of head request in flow f for the next collection period. If no unfairness is found to exceed the predefined limitation ε, Δ is set to be equal to 0. Thus we have the following:

$$\Delta^{p+1} = \begin{cases} U(T_m^p) * \phi_f & \text{if } U(T_m^p) > \varepsilon \\ 0 & \text{if } (U(T_m^p) \leq \varepsilon) \wedge (\Delta^p = 0) \\ \Delta^p & \text{if } (U(T_m^p) \leq \varepsilon) \wedge (\Delta^p > 0) \end{cases}$$

Finally, the priority used in the rate controller of flow f is defined as follows:

$$P(f) = S(r_f^i) + \Delta^{p+1} \tag{9}$$

The above analysis is based on two flows. In case there are more than two flows, we first need to find out the flow with the most unfairness, and then apply relative unfairness analysis for each of the rest flows. In addition, the negative compensation can be calculated for each flow.

C. ASFQ Algorithm

Based on the above principle of ASFQ, this section describes the ASFQ algorithm. Each flow f has a pending queue, Q_f, used to buffer arriving requests for f. Moreover, a variable, Δ_f, stands for the negative compensation for f at the current time. To maximize the system utilization, ASFQ sends up to D concurrent requests to the storage utility. These requests are maintained by a throttling queue, *TQ*. Whenever a request is selected for forwarding, it is added to the tail of *TQ*; when any request is finished by the storage service, it is removed from *TQ*. *Count* is the number of requests under service in the throttling queue; D stands for the maximal depth of *TQ*.

Algorithm 1 and 2 illustrate the ASFQ algorithm, including the process to add a new request into the pending

queues and the main loop for dispatching pending requests to the storage utility whenever the service is available.

```
/* When a new quest of flow f, r_f^i, arrives, append it
to the pending queue of f. */
Enqueue(r_f^i)
begin
    if TQ.Count > 0 then
        vct = v.S;
    end
    if TQ.Count = 0 then
        vct = v.F;
    end
    Q_f.Enqueue(r_f^i,vct);
end
```

Algorithm 1: Algorithm of Adaptive SFQ — Enqueue.

```
/* When the throttling queue is not full and there are
pending requests in the system, select proper requests
and send them to storage service.*/
Dequeue()
begin
    while TQ.Count < D do
        /*select the proper request from pending
        queues*/
        r = SelectNextRequest();
        /*send the selected request to the throttling
        queue for accessing storage service*/
        TQ.Add(r);
        /*Update virtual clock*/
        if v.S < r.S then
            v.S = r.S;
        end
        if v.F < r.F then
            v.F = r.F;
        end
    end
end
```

Algorithm 2: Algorithm of Adaptive SFQ — Dequeue.

The virtual clock, v, is maintained as a global variable. It consists of two properties: $v.S$ and $v.F$, which represent the maximum start tag and finish tag of all requests sent to the throttling queue. Whenever a new request, r_f^i, is arriving, we use Equation (2) and (3) to calculate its start and finish tag. This is accomplished by Q_f.Enqueue(r_f^i, cvt), where cvt is the current virtual time. The current virtual time depends on the status of throttling queue. When *TQ* is not empty, that means the storage service is busy and the virtual clock is set to be equal to the maximum start tag of all requests that have sent to *TQ*. On the contrary, if *TQ* is empty, the storage utility is idle and the virtual clock should be equal to the maximum finish tag of all request sent to *TQ*.

To select proper requests for service, we use Equation (9) to calculate the priority for head request of each of the flows. This is accomplished by SelectNextRequest() in the algorithm.

IV. Performance Evaluation

The ASFQ algorithm aims to ensure a fair share of the server capacity for competing flows to meet their minimal performance required even in case that the performance variance is bound to a subset of flows. In this section, we evaluate ASFQ in an object-based distributed storage system, called WinDFS, which is designed to support a .NET-based MapReduce programming model [21]. The evaluation system consists of one index server with one object server and several clients. Server machines manage objects (files) on disks. Each machine has a single Pentium 4 processor, 1GB of memory, 160GB IDE disk (10GB is dedicated to WinDFS), 1 Gbps Ethernet network and runs Windows XP. To simplify the performance comparison, the file buffer of Windows was turned off during reading or writing files. The ASFQ algorithm was implemented on the object servers.

During experiments, several workload generators on client machines kept sending I/O requests to server machines. Each workload generator was assigned with a workload object, the size of which is 1GB. In addition, each workload consisted of a number of threads and each thread sent independent request to access its workload object. To evaluate the performance, we collected the aggregated throughput (MB/sec) perceived by all threads of each workload generators. In the remainder of this section, each generator is also called a client. The cost for each request is defined to be its size.

A. SLA Compliance

The key capability of ASFQ is to meet the minimal performance defined by SLA for each flow with a fair manner. With sequential disk IO experiments, the maximal throughput of each server is around 12∼14MB/s. Assume two clients share this resource and with a reasonable resource reservation, the minimal throughput required by each client was set to be 5MB/s. The maximal unfairness was set to be 1MB/s. The size of each request was 512KB. Correspondingly, the weight ratio of the clients was 1:1. Each client consisted of 10 threads. All threads of the two clients started to work at nearly the same time. Experiments run with ASFQ enabled and disabled respectively.

Both clients finished within about 180 seconds and the throughput perceived by each client was sampled per second. Figure 4 illustrates the number of SLA violation (i.e. the perceived performance is less than 5MB/s.) In particular, each cross or circle in the figure represents the performance perceived by clients which is less than 5MB/s. With ASFQ

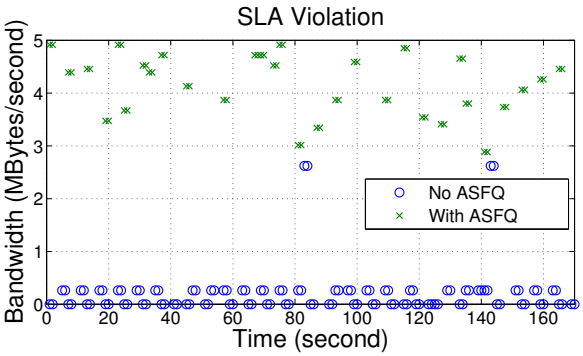

Figure 4: **SLA Violation.**

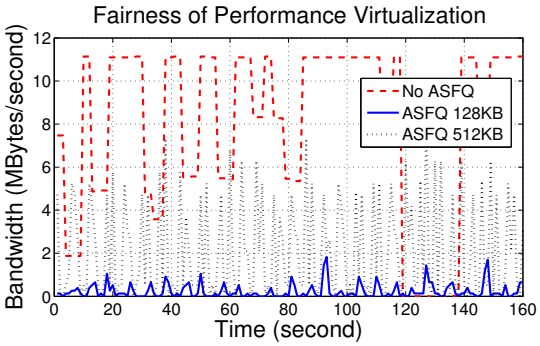

Figure 6: **Performance difference between clients.**

enabled, the number of violations is decreased up to 50% in comparison with the case without ASFQ. Moreover, the deviation of violated performance from SLA in the case with ASFQ is almost within 3MB/s and 5MB/s, which is not as worse as the case without ASFQ.

B. Performance Isolation vs. Efficiency

ASFQ trades system efficiency for fairness. However, the performance should not be degraded to an unacceptable level. The configuration of clients in this section is the same as Section IV-A. Figure 5 illustrates the performance collected at server side with ASFQ disabled and enabled. With ASFQ enabled, the size of requests was set to be 128KB and 512KB respectively. With ASFQ disabled, the size of each request was set to be 512K. The difference in efficiency at the server side is not significant. Specifically, with each configuration, each client finished its job in about 180 seconds. The reason is the maximum block size used by NTFS (the file system of Windows XP) to access disk is 64KB. It is reasonable to expect that any block size larger than 64K can efficiently utilize the system bandwidth.

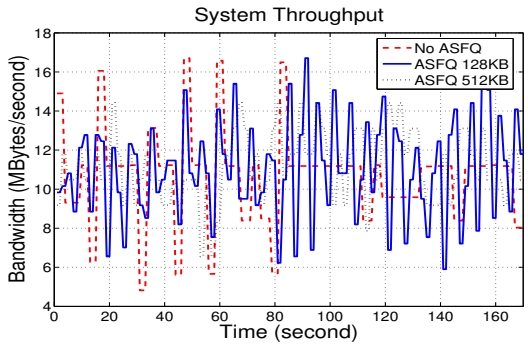

Figure 5: **Throughput of server.**

However, the difference in fairness perceived by clients is quit significant. Figure 6 shows the performance difference perceived by both clients with each configuration. Without ASFQ, the performance difference perceived by both clients can be as large as nearly 11MB/s. With ASFQ enabled, it is definitely under control and 128KB configuration allows a finer difference (about 1MB/s) than 512KB.

C. Fairness of ASFQ

In this section, we evaluate the fairness property of ASFQ by comparing it with SFQ(D) in case service rates perceived by clients are different. The experiment was conducted two times: respectively for ASFQ and SFQ(D). The depth of the throttling queue for both ASFQ and SFQ(D) was set to be 10. Two clients kept writing their objects to server. The configuration of clients is the same as Section IV-A.

Currently, the Windows operating systems do not expose methods to control the position of files on disks. To simulate different access rates, a delay was added to each of the requests of client 1. As a consequence, after each of the requests of client 1 is finished by the storage system, it does not return back immediately. On the contrary, it is sent back to the client after $d$ms of delay. To determine a proper value of d, we conducted an experiment to measure the average cost for accessing 512KB data on disk. It shows the average cost is about 150 ms. To simulate 30% performance difference at application level between outer and inner tracks, the value of delay is up to 45 ms. In experiments, a random value between 0 and 45 ms was selected as the delay for each of the requests of client 1.

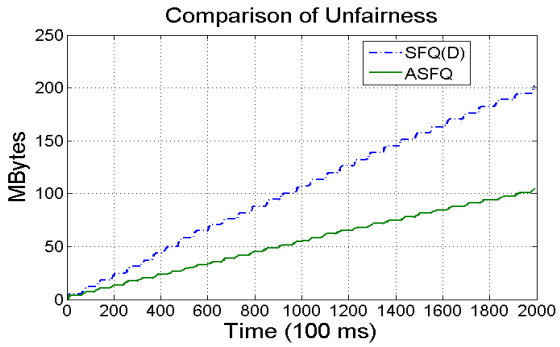

Figure 7: **Unfairness Comparison.**

To measure the fairness, *accumulated unfairness* (AU) was collected for both clients. AU stands for the sum

of unfairness within the short term. In particular, each of the clients collected its consumed resources every 100 ms; the difference between the consumed resources by the two clients is the unfairness within short term because weights for both clients are same. The sum of these values collected during the time of conducting experiment is AU, since all the threads started to work nearly at the same time.

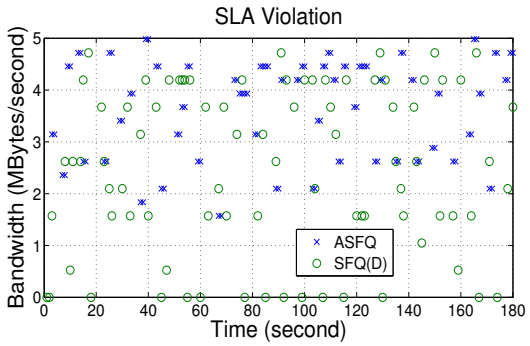

Figure 8: **SLA Violations with ASFQ and SFQ(D).**

The sampling period of the PM in ASFQ is selected to meet the latency requirements of practical storage-bound applications [2]. In our experiment, it was set to100ms. The condition to start rate controllers is:

$$|W_f(T_m^p) - W_g(T_m^p)| > \varepsilon \tag{10}$$

The results are illustrated in Figure 7 Under the same configuration, unfairness limited by ASFQ is more stringent than SFQ(D). Through the whole experiment, unfairness accumulated by ASFQ is nearly half of SFQ(D). This means ASFQ can provide more stringent fairness than SFQ(D) in case that performance variance by each client is different.

Next, we compare ASFQ with SFQ(D) on satisfying the minimal performance required by SLA. The delay configuration is the same as above. The number of SLA violations is collected for cases respectively with ASFQ and SFQ(D), as illustrate in Figure 8. The number of SLA violations with ASFQ is 30% less than the case with SFQ(D). Furthermore, the deviation of performance violation with ASFQ is within better control than SFQ(D). The reason is the feedback mechanism works effectively to balance the performance differences caused by physical positions for both clients.

With experiments, ASFQ is shown to be more effective on satisfying the minimal performance requirements than SFQ(D).

D. Impacts of Depth of Throttling Queue

In this section, we evaluate ASFQ under different depths of *TQ*. Two clients reserved bandwidth of 3MB/s and 6MB/s respectively. The maximal unfairness was set to be 1MB/s. The size of request was set to be 128KB. The depth of *TQ* was set to be 5, 10 and 15. The performance collected at server and clients are illustrated in Figure 9 and Figure 10 respectively.

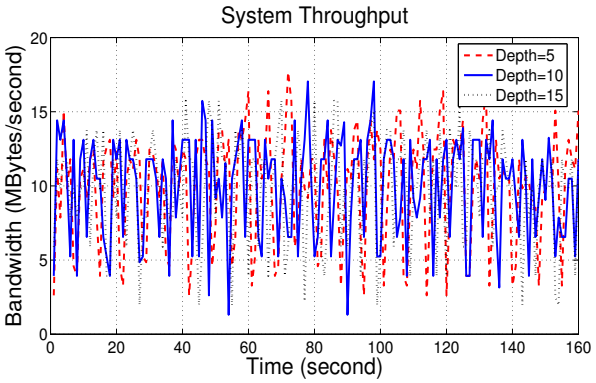

Figure 9: **System Efficiency.**

A larger depth of throttling queue results in a weakened fairness between competing clients and introduces more SLA violations. The reason is that the disk driver of Windows XP is more likely to reorder the requests when it receives a larger number of requests at the same time. As a consequence, the throttling control of ASFQ is weakened. However, the system utilization is not increased significantly. This is because the implementation of the throttling queue is achieved by an asynchronous model. Specifically, whenever there are pending requests in the *TQ*, they can be forwarded to the storage service almost immediately. This keeps the storage system always busy when there are waiting requests.

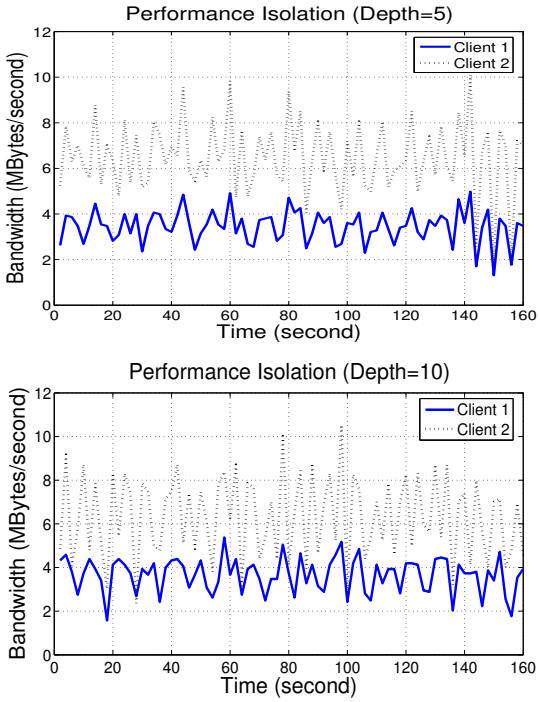

Figure 10: **Performance Isolation**

V. Conclusion

In this paper, we presented an Adaptive Start-time Fair Queue algorithm to meet the requirements of performance virtualization for shared storage systems. It can accommodate several workloads accessing the storage utility while meeting their individual Service Level Objectives. At the same time, it ensures the short term fairness enforcement between competing clients, even in case when the variance performance of disks is bound to a subset of clients.

ASFQ achieves its objectives by using a feedback mechanism. A monitor is used to observe the performance difference in a real time manner and to collect information that is used by ASFQ for balancing performance perceived by different flows according to the defined SLA. We implemented this algorithm and evaluated it on a real distributed storage system. The fairness of algorithm is presented with experimental results. The result of experiments shows the effectiveness of ASFQ to meet SLA requirements. In addition, various impacts from different request sizes and the depth of throttling queue were evaluated. These results demonstrate that our algorithm can isolate performance for competing users with stringent performance and fairness requirements.

Acknowledgment

This work is partially supported by research grants from the Australian Research Council (ARC) and Australian Department of Industry, Innovation, Science and Research (DIISR). We would like to thank Christian Vecchiola and Saurabh Garg for their comments on improving the quality of the paper.

References

[1] J. Wilkes, "Traveling to Rome: QoS specifications for automated storage system management," *Lecture Notes in Computer Science*, vol. 2092, pp. 75–92, 2001.

[2] G. DeCandia, D. Hastorun, M. Jampani, et al, "Dynamo: Amazon's Highly Available Key-Value Store," in *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[3] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Façade: Virtual Storage Devices with Performance Guarantees," in *Proc. of the 5th USENIX conference on File and Storage Technologies*, 2002.

[4] P. Gang and T. Chiueh, "Availability and Fairness Support for Storage QoS Guarantee," *Proc. of the 28th International Conference on Distributed Computing Systems*, 2008.

[5] W. Jin, J. S Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," *Proc. of the 2004 ACM Sigmetrics Conference on Measurement and Modeling of Computer System*.

[6] Y. Wang, and A. Merchant, "Proportional-share scheduling for distributed storage systems," *Proc. of the 5th USENIX conference on File and Storage Technologies*, 2007.

[7] L. Zhang, "Virtual clock: a new traffic control algorithm for packet switching networks," *ACM SIGCOMM Computer Communication Review*, vol. 20, no. 4, pp. 19–29, 1990.

[8] G. R. Ganger, "Blurring the Line Between OSes and Storage Devices," in *CMU-CS-01-166, White Paper of Parallel Data Lab, Carnegie Mellon University*, 2001.

[9] M. Karlsson, C. Karamanolis and X. Zhu, "Triage: performance differentiation for storage systems using adaptive control," *ACM Transactions on Storage,*, vol. 1, no. 4, pp. 458–480, 2005.

[10] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors," in *Proc. of the 4th Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[11] J. Bruno, J. Brustoloni, E. Gabber, et al, "Disk scheduling with quality of service guarantees," *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, 1999.

[12] P. Goyal, H. M. Vin, H. Cheng, "Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," *Proc. of ACM SIGCOMM*, 1996.

[13] 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 *Proc. of the 22nd International Symposium on Reliable Distributed Systems (SRDS)*, 2003.

[14] J. Zhang, A. Sivasubramaniam, Q. Wang, E. Riedel, and A. Riska, "An interposed 2-Level I/O scheduling framework for performance virtualization," *Proc. of the 2005 ACM Sigmetrics Conference on Measurement and Modeling of Computer System*.

[15] L. Huang, G. Peng, and T. Chiueh. , "Multidimensional storage virtualization." *Proc. of the International Conference on Measurement and Modeling of Computer Systems*, 2004.

[16] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence, "Fab: Building distributed enterprise disk arrays from commodity components," *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[17] S. J. Golestani, "A Self-Clocked Fair Queuing Scheme for High Speed Applications," *Proc. of IEEE INFOCOM'94*, 1994.

[18] Hitachi Global Storage Technologies, "Hard Disk Drive Specification: Hitachi Travelstar 5K80 2.5 inch ATA/IDE hard disk drive," 2003.

[19] E. Papathanasiou and M. L. Scott, "Aggressive Prefetching: An Idea Whose Time Has Come," in *Proc. of the 10th HotOS*, 2005.

[20] E. Riedel, C. Ingen, J. Gray, "Sequential I/O on Windows NT^{TM} 4.0 - Achieving Top Performance," in *Technical Report of Microsoft Research*, 2000.

[21] C. Jin, R. Buyya, "MapReduce Programming Model for .NET-based Cloud Computing," in *Proc. of Euro-Par 2009*.