

Towards Multi-objective Scheduling in Shared Storage Systems*

Ajay Gulati
Department of Computer Science
Rice University, Houston, TX
email: gulati@rice.edu

Arif Merchant
HP labs
Palo Alto, CA
email: arif@hpl.hp.com

Peter Varman
Department of ECE & CS
Rice University, Houston, TX
email: pjv@rice.edu

Abstract

Economies of scale and ease of centralized management has led to the rapid growth of shared data centers. Meeting response time deadlines and throughput requirements of various workloads contending for these shared resources is quite challenging. This paper presents a two-level scheduling framework to meet response time guarantees by dynamically reordering the requests while maintaining the throughput given to each of the clients. Preliminary results show that our approach provides performance isolation and is work conserving.

1 Introduction

Consolidated data centers providing terabytes of storage and hundreds of GB/s of bandwidth are increasing in popularity, driven by the economics of sharing and the advantages of centralized management. Server farms are often shared by multiple commercial organizations or different divisions or departments within an institution. Such sharing arrangements are often codified in the form of service level agreements which specify the desired amount of service in terms of attributes such as requests/sec, CPU time, maximum response time per job etc. Providing quality assurances to multiple job streams accessing shared resources is often very challenging. The problem becomes more challenging if the inputs streams are unpredictable and varying. Sometimes the workloads have very different service requirements in terms of throughput and response times. For example, large file transfers require high bandwidth but can tolerate high response times as well, whereas a user interface program needs

only a small bandwidth but needs a small response time.

In this extended abstract, we propose a two-level scheduling scheme that decouples the bandwidth and response time requirements, and meets response time deadlines without affecting the throughput requirements of the flows. This scheme arbitrates contention for the resources among competing flows while enforcing rate control, and dynamically reorders requests at the second level to control response times. The scheme also permits spare capacity to be dynamically allocated among the contending flows, and is work conserving.

Generalized processor sharing and many of its variants [1, 4, 5] have been well studied in the context of network scheduling, to allocate resources to provide fair or differentiated quality-of-service. The inverse relationship between the bandwidth provided to a flow and its response time [7, 11] has been used to provide deadline guarantees by controlling the throughput. A problem with the approach is that a flow with higher weight can misbehave and consume more than its designated share of resources by sending a burst of requests. Parekh and Gallager [11] have shown bounds on delays using a leaky bucket based mechanism to control flow rates, and a variant of GPS (PGPS) to do the weighted allocation.

The rest of the extended abstract is organized as follows. In Section 2 we describe the model and definitions. In Section 3 we present our components for rate allocation and request reordering to meet response time deadlines. Section 4 presents some preliminary evaluation results. We review related work in section 5 and end with a discussion of some important issues in section 6.

*Support by the National Science Foundation under Grant CCR-0105565 and the IRD program is gratefully acknowledged.

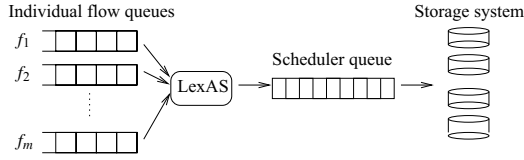


Figure 1: System model

2 System Model

The storage system consists of multiple disks that can concurrently serve m flows as shown in Figure 1. Each flow i , $1 \leq i \leq m$, consists of a sequence of *service requests*; its service requirements are represented by a tuple $\mathcal{R}_i = \langle T_i, D_i \rangle$, where T_i is the *throughput* in terms of *service units* per unit time (such as bytes/sec, I/Os/sec, jobs/sec, CPU-cycles/sec etc.) and D_i is a bound on the *average response time* (measured over a suitable window) for the requests. To allocate service units to a flow we assign each flow j a *weight* w_j that represents the fraction of total service that it should receive in some window of time. We choose $w_j = T_j / \sum_{i=1}^m T_i$, so that if the system has sufficient capacity every flow can be guaranteed its throughput requirement. These weights need to be re-computed if a new flow is added into the system. We want to achieve the following goals in the system: (1) The throughput assigned to each flow is in proportion to its weight (2) Every flow meets its latency requirements as long its arrival rate is less than its throughput requirement (3) Spare capacity should be allocated without negatively affecting the response times of flows (4) No flow should be penalized for using spare capacity.

Many previously proposed schemes for controlling response times are based on appropriately choosing the weights, w_i , of the flows. These schemes assign a larger weight to a flow in order to decrease its response time. For instance, consider two flows f_1 and f_2 with requirements, $\mathcal{R}_1 = \langle 500u/sec, 0.1sec \rangle$ and $\mathcal{R}_2 = \langle 500u/sec, 0.75sec \rangle$ respectively. Here u/sec denotes the service units per sec. Assume that the capacity of the system is $1200u/sec$ which is more than the cumulative service requirement of both flows. Since the throughput requirements of both flows are the same, we have $w_1 = w_2 = 0.5$. Thus the requests from 2 flows will be scheduled in a 1:1 interleaved fashion. It is possible that this ordering may cause f_1 to exceed its response time bound, but allow f_2 to meet it. A scheme based on single set of weights will increase the weight of f_1 in order to meet

its latency deadline. Suppose that the weights $w_1 = 2/3$ and $w_2 = 1/3$ satisfy the bounds on response times. This in turn reduces the throughput allocated to f_2 to 400 u/sec which is less than its requirement of 500u/sec. In this case f_2 will only get its desired throughput if f_1 is well-behaved and leaves sufficient spare capacity for f_2 . Should however f_1 make requests at 800u/sec, it will be granted that much at the expense of f_2 . To avoid degradation of f_2 's service, the throughput provided to f_1 needs to be controlled in some way, while allowing it higher priority to satisfy its response time requirement. The problem is aggravated in dynamic situations where requests have resource constraints, and service times and arrival rate are unpredictable. Thus a static assignment of weights and scheduling policy is unlikely to be totally satisfactory.

In this paper we propose a two-level scheduling scheme to meet the dual constraints on throughput and service time. The first level allocates service to flows in proportion to their throughput requirements, while the second level reorders requests to meet response-time deadlines. Thus the throughput requirement determines the weight of a flow, and timing deadlines decide the order in which requests are actually scheduled from the second level. Spare system capacity is shared by all the backlogged flows in proportion to their weights, to achieve work conservation. A related 2-level approach has been proposed recently by Zhang et al. [15], but their main focus is on controlling the length of the storage system queue. Also [15] considers the storage system as a black-box whereas we consider resource conflicts explicitly.

3 Scheduling Algorithms

There are two sets of queues associated with the system. First, there are a set of *flow queues*, one for each flow. The flow queues buffer incoming requests before they are promoted to the *scheduler queue*. Requests in the scheduler queue are ordered by the priority with which they will be dispatched to the server. Although the scheduler queue can be implemented by appropriate ordering and linking of requests in the flow queues, for conceptual clarity we will consider it to be a separate object as shown. Requests are finally dispatched to the storage system from the scheduler queue. Whenever a disk becomes free the next request for that disk from the scheduler queue is dispatched to the it.

The first-level scheduler uses weights w_i to gate requests to the scheduler queue. This is done by the **LexAS** [6] algorithm which selects requests in proportion to the weights assigned to each of the flows, while distributing them as evenly as possible among the disks. LexAS can be tuned to provide both short and long term weighted allocation. These are called FWS and DUaL policies in [6]. We exploit this flexibility to meet goals 3 and 4 (allocation of spare capacity and not penalizing a flow for using it). Once the requests are selected and put into the scheduler queue, our second algorithm **ARR** (*Adaptive Request Reordering*) reorders these requests to meet timing deadlines for the flows. ARR assigns a dynamically adjustable parameter δ_i to flow i ; the larger the value of δ_i the greater the share of server resources allocated to that flow, and the lower its response time. Now we will discuss rate control using the LexAS and ARR algorithms in detail.

3.1 LexAS : Rate Control

LexAS does scheduling in rounds. In each round it considers the set of requests pending in the flow queues and finds a one to many mapping between flows to disks such that the number of IOs allocated to a flow is proportional to its weight. LexAS maintains a **normalized cumulative allocation vector** $\mathcal{N}(t)$, consisting of weight-scaled values of the number of requests serviced for a flow. That is, $\mathcal{N}(t) = [\eta_1, \eta_2, \dots, \eta_m]$, where $\eta_i = B_i/w_i, i = 1, \dots, m$ and B_i is the number of requests allocated to i^{th} flow. At every scheduling step k , it considers the current vector $\mathcal{N}(k)$ and finds an allocation that is work conserving and results in the lexicographically smallest vector $\mathcal{N}(k+1)$ possible, subject to disk contention. This in turn leads to a weighted allocation of number of IOs to flows.

In a dynamic situation flows exhibit bursty behavior where the arrival rate of a flow exceeds the designated throughput in some intervals and idles during others. There are two main issues that arise during the first-level allocation.

(1) Spare Capacity Allocation: If a flow sends more requests than its designated allocation and the system has spare capacity, then those requests should not necessarily be stopped by the rate control mechanism. However, releasing those requests blindly might cause other flows to miss their deadline as these extra requests will occupy spaces in the scheduler queue. To avoid this, LexAS caps the amount

by which it allows any flow to exceed its designated share. In a round, only those flows whose allocation η_i is below a threshold are candidates for promotion to the scheduler queue, unless the scheduler queue length falls below a certain length designated as ARR_{min} . The threshold, T_{max} , is the largest possible allocation so that the cumulative deficit (the sum over all flows j of the positive difference between the threshold and $\eta_j, \sum_{\forall j} T_{max} - \eta_j$, if $T_{max} \geq \eta_j$) is below a designated parameter $LexAS_{max}$. This mechanism allows a flow to use spare capacity but it must wait in the flow queue until most of the legitimate requests are first serviced. This prevents the excess requests from delaying other flows in the scheduler queue, while still permitting them to utilize the spare capacity.

(2) Not Penalizing Flows for using spare capacity: A flow's allocation can get ahead of other flows due to two reasons; either because it sends extra requests and utilizes the spare capacity, or other requests send fewer than the designated number of requests. The rate control mechanism should ensure that a flow is not penalized in the future in either case. To avoid this, we artificially increase the allocation of all lagging active flows to a minimum threshold at the start of a round. The threshold, T_{min} , is set to the lowest value so that the cumulative deficit ($\sum_{\forall j} \eta_{max} - \max(T_{min}, \eta_j)$, where η_{max} is the maximum allocation) is no more than a maximum amount designated by $LexAS_{min}$.

Both the schemes compute the threshold values (T_{max}, T_{min}) based on the average allocation values for all the flows. We assume a scheduler queue of finite length L . Thus LexAS will schedule requests from active flows as long as the number of pending requests is less than L . Thus at any time the number of requests observed by a newly activated flow is bounded.

3.2 Adaptive Request Reordering

In this section we will present our reordering algorithm ARR that actually re-orders the requests in the scheduling queue so as to meet timing deadlines. The algorithm maintains an allocation vector $\mathcal{A} = [a_1, a_2, \dots, a_n]$, where a_i corresponds to the number of requests serviced for i^{th} flow. Each flow is assigned a parameter δ_i to control the response time experienced by the flow; a higher δ_i signifies a smaller response time. The response times of flows are monitored by the system, and the δ_i 's are adaptively changed to meet the deadlines. Whenever a disk

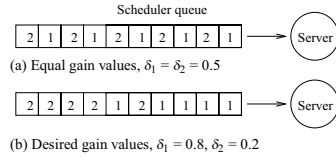


Figure 2: Schedules with different gain values

d_i is idle, a new request is chosen from the flow j having at least one request for d_i , and with minimum value of $(a_j + 1)/\delta_j$. Thus the lower the value of $(a_j + 1)/\delta_j$, the higher the priority of the flow. Components of \mathcal{A} are reset to zero periodically or whenever δ values are changed.

We use the following strategy to set δ_i values. Let the *gain vector* $G = [g_1, g_2, \dots, g_m]$, where each g_i is a parameter associated with flow i and is initialized to some value κ . The scheduling parameter for flow i , $\delta_i = g_i / \sum_j g_j$. ARR treats δ_i as the internal weight of flow i . After every few rounds, the system checks the average response time experienced by a flow to see if it is greater or smaller than its bound. For any flow f whose deadline is missed, g_f is incremented to $g_f + \gamma$, where γ is a tunable parameter. This will give preference to this flow in the scheduling queue without affecting the throughput of the flows. Changing δ_i changes the interleaving of the requests in the scheduler queue, with flows with higher δ having a larger fraction of slots at the front of the queue. For two flows it is easy to see that with this choice of adaptive function, the values of gains will stabilize, if the system can actually support timing deadlines for both flows. Let d_1 and d_2 denote the minimum values of δ_1 and δ_2 necessary for flows 1 and 2 respectively to meet their deadlines. The triangular region of the $\delta_1 - \delta_2$ plane bounded by $\delta_1 \geq d_1$, $\delta_2 \geq d_2$, and $\delta_1 + \delta_2 = 1$ defines the feasible range of δ values necessary to simultaneously meet both deadlines. The system state will move along the line $\delta_1 + \delta_2 = 1$ in steps of monotonically decreasing size till it lies within the feasible area. The general case will be considered in the full paper. If it is not possible to simultaneously meet all response time deadlines at the allocated throughput, then one or more flows will continuously miss their deadline. This condition should be monitored and reported to admission control.

Consider an example with two flows f_1 and f_2 , where each one has equal throughput requirements, but f_1 has a much more stringent timing deadline. Initially the gain values will

- (1) **ARR algorithm:**
- (2) Let Gain vector $G = [g_1, \dots, g_m]$, such that all components are equal
- (3) **Periodically** for each fbw k
- (4) **if**(deadline not met)
- (5) $g_k = g_k + \gamma$
- (6) *Allocation vector*, $A = [a_1, \dots, a_m]$, such that all components are zero
- (7) **for** all fbws, $\delta_j = g_j / \sum_i g_i$
- (8) **Issue requests as follows:**
- (9) **for** any idle disk d_k
- (10) Issue next request from fbw i having minimum value for $(a_i + 1)/\delta_i$
- (11) $a_i = a_i + 1$;

Figure 3: ARR algorithm

be same for each of them and the scheduler will order the requests in alternate order as shown in figure 2(a). But as f_1 misses its deadline, its gain value will be increased and that might cause $\delta_1 = 0.8$ and $\delta_2 = 0.2$. In that case the scheduling order would be as shown in figure 2(b) and this particular order might satisfy timing deadlines for both the flows. The details of the algorithm ARR (adaptive request reordering) are presented in Figure 3.

4 EVALUATION

We evaluated our algorithms using a simulation framework. We simulate N storage servers, where each server can service requests at a maximum rate R . This rate R is inversely proportional to the mean service times assigned to requests. The service times are assigned using a uniform distribution between t_{min} and t_{max} , where t_{min} is the time taken to fetch the next block on the same track and t_{max} is the sum of maximum seek and rotational delay. Thus the overall system throughput can vary between R to NR depending on how balanced the distribution of workload is across the disks. We ignore low level disk scheduling for this simulation. We experimented with two different input arrival patterns: Poisson and bursty. To simulate bursty workloads, a time instant in every interval of 1 sec was randomly chosen. Requests times are chosen from a low-variance Gaussian distribution centered at this reference instant.

We experimented with 3 flows with requirements $\langle 200 \text{ reqs/sec}, 0.03 \text{ sec} \rangle$, $\langle 300 \text{ reqs/sec}, 0.05 \text{ sec} \rangle$, $\langle 400 \text{ reqs/sec}, 0.1 \text{ sec} \rangle$ respectively. The system capacity is around 1300 reqs/sec ($N=8$, $t_{min} = 0.013 \text{ msec}$, $t_{max} = 12.1 \text{ msec}$) if all disks are accessed with equal probability. The other parameters are

as follows: $LexAS_{min} = 100, LexAS_{max} = 50, ARR_{min} = 20$, and L is set to half of cumulative arrival rate. These parameters are selected such that small variation in arrival rate due to Poisson arrivals is absorbed, but the effect of the large variation due to an ill behaved flow is apparent.

4.1 Underloaded system

First we show that we can meet both throughput and latency requirements for an underloaded system. Figure 4 and 5 show the throughput and response times obtained by the three flows when the system has sufficient excess capacity. All flows receive their required throughput and meet or better their response time requirements.

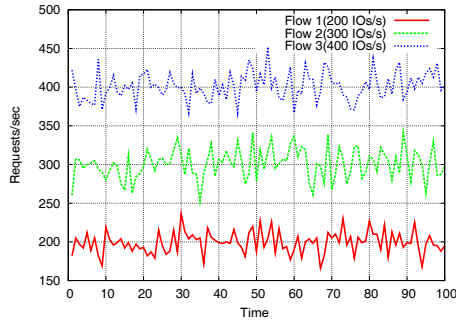


Figure 4: Throughput - Underloaded system

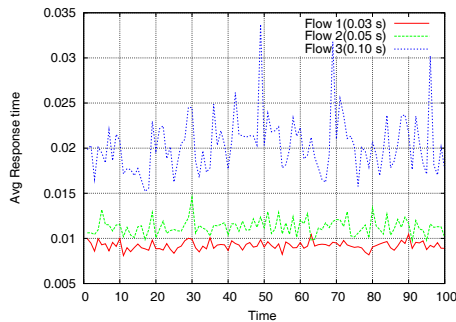


Figure 5: Response time - Underloaded system

4.2 Overloaded system

Next we experimented with a system that is overloaded to different extents. We experimented with two situations in the case. One is when the system can meet the throughput requirements but not the response time deadlines. In this case each flow would simply observe a higher but bounded response time. In the second scenario, the total arrival rate is higher than the

system capacity and the latency will keep on increasing. Figures 6 and 7 show the throughput and response times obtained by various flows when they all send more than the expected number of requests. The actual rate of sending requests is 250, 400 and 600 for flows 1, 2 and 3 respectively. Please note that the throughput attained by each of the flows closely matches their arrival rate due to the work conserving property of our algorithms, but the latency deadlines are missed (though still bounded) by all of them.

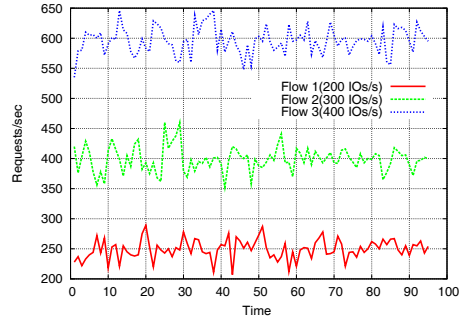


Figure 6: Throughput - Overloaded system

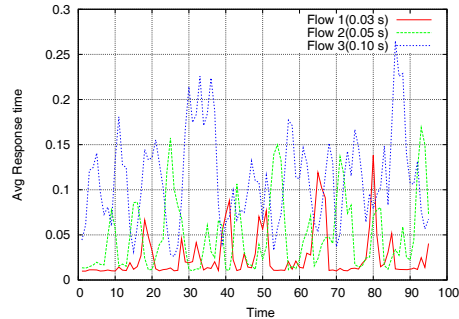


Figure 7: Response time - Overloaded system

Next we experimented with a scenario where the total arrival rate exceeds the system capacity. In this case, throughput requirement is set to 300 req/sec for all flows but they send 450 each instead. Figures 8 and 9 show the throughput and response times obtained by various flows when they all send more than the expected number of requests. Please note that system is fully utilized but the response times keep on increasing due to queue build up at the input.

4.3 Badly-Behaved Flows

In this section we will discuss some experiments consisting of a mixture of well and ill behaved flows. We first consider the situation where

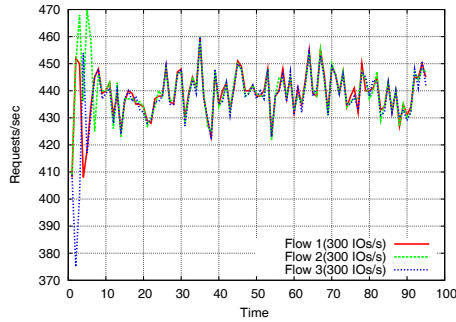


Figure 8: Throughput - Capacity violation

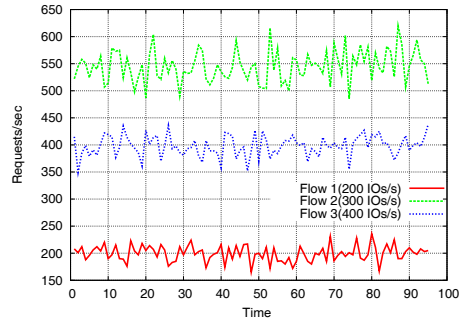


Figure 10: Throughput - Flow 2 misbehaves

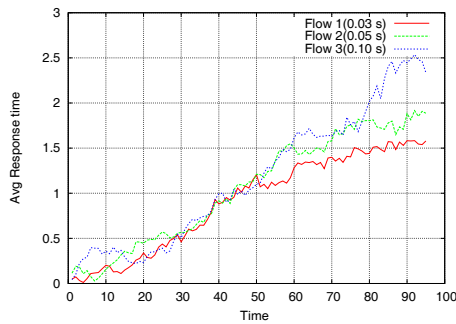


Figure 9: Response time - Capacity violation

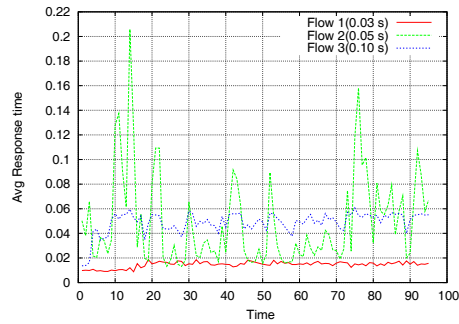


Figure 11: Response time - Flow 2 misbehaves

one flow sends more requests than its designated throughput requirement. Figure 10 shows the throughput achieved by our scheme, when flow 2 misbehaves and has an arrival rate of 550 reqs/sec instead of 300 reqs/sec. Please note that other flows still get their desired throughput and flow 2 utilizes the spare capacity in the system. Figure 11 shows the response time achieved by the 3 flows. Well-behaved flows 1 and 3 stay within their specifications, while flow 2 misses its deadline. This is because LexAS holds the excess requests for flow 2 in the flow queues till the requests from other flows have been serviced. Flow 2 increases its value of δ_2 in an attempt to decrease its response time. This causes the response time of flows 1 and 3 to increase. However, as soon as they miss their deadline, they prevent flow 2 from increasing δ_2 further, and the system stabilizes.

We then experimented with a bursty arrival pattern. In this case flow 1 has a bursty arrival pattern starting from 25 seconds onwards. Figure 12 shows the average response time observed by them. We observed that the throughput is unaffected by the behavior and the response time of flow 1 increase because of burstiness. Similar to the previous case, the response time of other flows also gets a little higher

but remain within their specifications. We also experimented with an ON-OFF arrival pattern, where one flow is periodically ON and OFF and it sends requests at a higher rate during the interval it is ON. Figure 13 and 14 show the throughput and response times observed when flow 2 is periodically ON and OFF for 5 sec intervals. The deadlines are set to 0.03, 0.05 and 0.05 sec for flows 1, 2 and 3 respectively. The arrival rate is double (600 req/sec) during the ON intervals. Please note that the response times of flows 1 and 3 increase and oscillate around their deadlines whereas flow 2 misses its deadline due to the ON-OFF behavior.

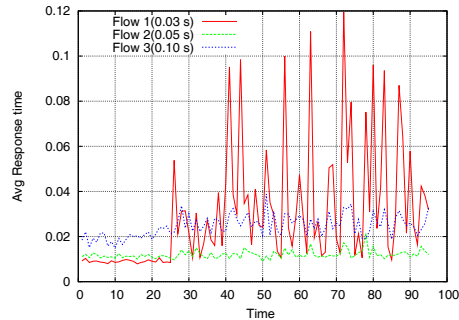


Figure 12: Response time - Flow 1 is bursty

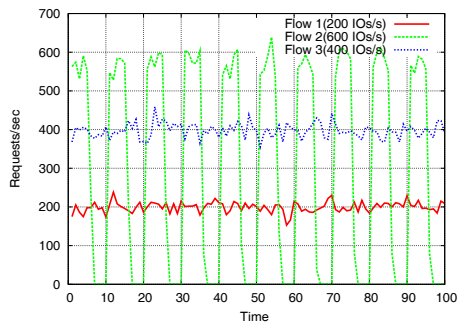


Figure 13: Throughput - Flow 2 is ON-OFF

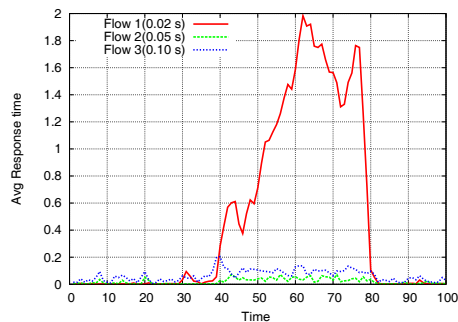


Figure 16: Response time obtained by ARR

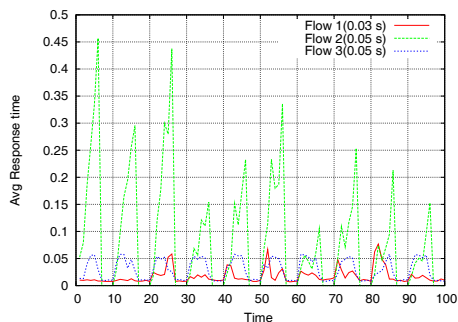


Figure 14: Response time - Flow 2 is ON-OFF

Finally we consider the effect of using a non-adaptive scheduling strategy like Earliest Deadline First in place of ARR. For simplicity we consider a single server for this case. Fig 15 shows the response times obtained using EDF when one of the flows misbehaves and sends in more than its designated share of the requests. The extra requests will get higher priority in the EDF queue and delay requests from the well-behaved flows. In contrast, ARR insulates the other flows from the ill behaved flow as shown in Figure 16. Dynamic adaptation allows the excess capacity to be utilized without degrading the response time of well-behaved clients.

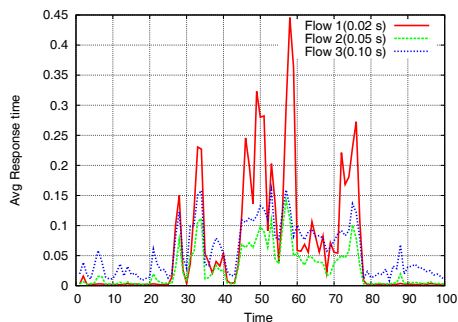


Figure 15: Response times obtained using EDF

5 Related work

A lot of research has been done in networking for weighted assignment of bandwidth [1, 4, 5]. Most of these schemes try to approximate the idealized Generalized Processor Sharing (GPS) strategy [5] to trade off computational speed and fairness in the context of high-speed routers. These schemes do not address response time requirements explicitly, but instead guarantee a certain bandwidth within any time window. Parekh and Gallager [11, 12] proved the bounds on response time observed by a flow using a leaky bucket input model and PGPS, which is another approximation to the processor-sharing algorithm. Their schemes can be used in static scenarios where the parameters of the leaky bucket and weight assignment are known a priori. In contrast our scheme tries to dynamically adapt to the variation in the input flows, and exploit any unused capacity at a fine time granularity. For storage systems, Facade [10], SLEDS [3], Stonehenge [7], Interposed Proportional Scheduling [8], Triage [9] and Lexicographic Scheduling [6] provide schemes for bandwidth allocation. Response time management when addressed, is done by monitoring queue lengths or adjusting the weight. As noted earlier, having only a single control point for both bandwidth and response time is insufficient to deal with ill-behaved flows.

Recently another 2-level scheduling framework similar to ours has been proposed by Zhang et al. [15], which does rate control based on a credit based scheme at first level and uses EDF scheduling at the second level. Our scheme differs in two aspects: first Zhang et al. [15] consider the storage system as a single shared resource, whereas we consider multiple resources and conflicts among the requests during rate control. Secondly, they [15] use EDF

and storage system queue manipulation to balance throughput and latency bounds, whereas we use ARR to reorder requests before sending them to the storage system. As noted earlier, a static scheme such as EDF at the second level may allow a flow with large number of pending requests to occupy the front of the EDF queue, causing other flows to miss their deadlines as well. In our scheme one badly behaving flow cannot cause the response time of other flows to go beyond their deadline if the system is capable of supporting them.

Several schemes for real-time scheduling of disks for multimedia streaming have been proposed (e.g. [2, 13, 14]), in the context of more predictable workloads than the situation considered in this paper.

6 Conclusions and Discussion

We presented a two-level scheduling framework to obtain both proportional throughput and latency deadlines for well behaved flows in a storage system. Preliminary evaluation shows that our approach performs well in terms of isolating flows and utilizing the system. We believe that our approach raises some interesting research challenges on achieving robust rate control and dynamically adapting to meet response time deadlines. Some of the parameters that affect our scheduling framework are as follows:

- (1) *Adaptive weight assignment function*: On missing a deadline we change the δ_i by incrementing the value of g_i . It might be interesting to see what other functions can be used. Can we multiply g_i by a constant (> 1)? Which function will provide least oscillations? Can it be shown which functions always converge if there is a possible assignment, and which provide the fastest convergence rate?
- (2) *Rate control using LexAS*: How do we choose $LexAS_{min}$ and $LexAS_{max}$? Can they be made adaptive based on the latency deadlines.
- (3) *Low level disk scheduling*: What effect would lower level disk scheduling have on response time guarantees?

References

- [1] J. C. R. Bennett and H. Zhang. WF^2Q : Worst-case fair weighted fair queueing. In *INFOCOM (1)*, pages 120–128, 1996.
- [2] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Volume 2*. IEEE Computer Society, 1999.
- [3] 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*, pages 109–118, 2003.
- [4] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–26, September 1990.
- [5] A. G. Greenberg and N. Madras. How fair is fair queuing. *J. ACM*, 39(3):568–598, 1992.
- [6] A. Gulati and P. Varman. Lexicographic QoS scheduling for parallel I/O. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 29–38, Las Vegas, Nevada, United States, 2005. ACM Press.
- [7] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Perform. Eval. Rev.*, 32(1):14–24, 2004.
- [8] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. *SIGMETRICS Perform. Eval. Rev.*, 32(1):37–48, 2004.
- [9] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *IWQoS*, June 2004.
- [10] C. Lumb, A. Merchant, and G. Alvarez. Façade: Virtual storage devices with performance guarantees. *File and Storage technologies (FAST'03)*, pages 131–144, March 2003.
- [11] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to fbw control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, 1993.
- [12] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to fbw control in integrated services networks: the multiple node case. *IEEE/ACM Trans. Netw.*, 2(2):137–150, 1994.
- [13] A. L. N. Reddy and J. Wyllie. IO issues in a multimedia system. *IEEE Computer*, 27(3):69–74, 1994.
- [14] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS*, pages 44–55. ACM Press, 1998.
- [15] J. Zhang, A. Sivasubramaniam, A. Riska, Q. Wang, and E. Riedel. An interposed 2-level I/O scheduling framework for performance virtualization. Technical Report CSE-05-003, The Pennsylvania State University, University Park, PA, February 2005.