Efficient QoS for Multi-Tiered Storage Systems

Ahmed Elnably *Rice University*
Hui Wang *Rice University*
Ajay Gulati *VMware Inc.*
Peter Varman *Rice University*

Abstract

Multi-tiered storage systems using tiers of SSD and traditional hard disk is one of the fastest growing trends in the storage industry. Although using multiple tiers provides a flexible trade-off in terms of IOPS performance and storage capacity, we believe that providing performance isolation and QoS guarantees among various clients, gets significantly more challenging in such environments. Existing solutions focus mainly on either disk-based or SSD-based storage backends. In particular, the notion of *IO cost* that is used by existing solutions gets very hard to estimate or use.

In this paper, we first argue that providing QoS in multi-tiered systems is quite challenging and existing solutions aren't good enough for such cases. To handle their drawbacks, we use a model of storage QoS called as *reward scheduling* and a corresponding algorithm, which favors the clients whose IOs are less costly on the backend storage array for reasons such as better locality, read-mostly sequentiality, smaller working set as compared to SSD allocation etc. This allows for higher efficiency of the underlying system while providing desirable performance isolation. These results are validated using a simulation-based modeling of a multi-tiered storage system. We make a case that QoS in multi-tiered storage is an open problem and hope to encourage future research in this area.

1 Introduction

This paper raises issues that arise in providing QoS in multi-tiered storage architectures, and presents a resource allocation model and scheduling framework suited for this situation. Two major trends motivate this work. The first is the growth in virtualized data centers and public clouds hosted on shared physical resources. In these cases, paying customers increasingly insist on receiving certain performance SLAs in terms of CPU, memory and IO resource allocation similar to what they would experience in a dedicated infrastructure. However, as we argue below, existing QoS models for storage do not address this need adequately.

The second driver is the rapid spread of multi-tiered storage systems that employ aggressive SSD-based tiering or caching within storage devices that are centralized [2–5], or created using local storage from virtualized hosts [1]. These solutions boost performance while lowering cost, but complicate the (already complex) storage resource allocation problem significantly by requiring the management of heterogeneous devices. Storage QoS needs to be aware of the massive performance differences across tiers in order to provide performance isolation while maintaining high efficiency of the underlying devices.

In this paper we discuss how the significant differences in speed between SSD and disk accesses makes traditional proportional-share resource allocation models unsuitable in meeting client's expectations. The key challenge is that existing IO schedulers rely on some notion of **IO cost** to do accounting for IOs for future scheduling. Estimating this cost is critical and also hard in such multi-tiered systems. To some extent, this problem is present even in traditional storage systems where hard disks have different service times for random vs. sequential IOs and SSDs have the same issue for read vs. write IOs. In case of multi-tiered systems this problem becomes quite acute and needs further attention.

We adapt a QoS performance model called *reward allocation* that was proposed in [8, 9] as an approach to handle this situation. The idea behind our proposed reward-based QoS model is to explicitly favor applications that make more efficient use of the resources, rather than use the gains from one application to subsidize the performance of under-performing applications. The subsidy model exemplified by *proportional sharing* is currently the most common solution for resource allocation in storage systems (see Section 2).

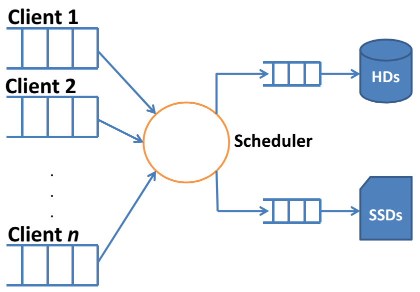


Figure 1: Reward Scheduler for Multi-Tiered Storage

In this paper, we examine how well a black-box storage model using only the measured response times at the host is able to provide reward discrimination. Our aim is not to present a final solution that works under all circumstances, but rather to create a discussion on the effect of tiered storage and application expectations on the required QoS directions for future storage systems.

The rest of the paper is organized as follows. In Section 2 we motivate and define our problem, and compare it with existing approaches. In Section 3 we describe our reward scheduling algorithm. We present simulation results in Section 4, and conclude in Section 5.

2 Overview

Figure 1 shows a schematic model of our tiered-storage architecture. Client requests are directed to the storage array. Within the array, the requests are sent to the queue of either the SSD or hard disk (HD) tier, from where they are served based on the array's scheduling priorities. The scheduler uses proportional share scheduling to choose which request to dispatch to the array next.

We motivate our reward QoS model using an example. Consider two clients A and B with equal weights. For the base case we consider the situation where all accesses are from HD tier, with an average service time of 10ms. A single hard disk system in this case provides an aggregate throughput of 100 IOs/sec (IOPS), equally shared between the two clients. Now consider what happens when the hit ratio of A in to SSD tier increases from 0 to $2/3$. Assume that an SSD access takes 50μs. The system completes IOs at the rate of 6 IOs every 40ms or 150 IOPS. Each client receives 75 IOPS.

We argue that proportional sharing as described above is not ideal for either the client or the system. With proportional sharing the throughputs of both clients increases when the hit ratio of client A improves. However, A does not experience the same increase in IOPS that it would have on a dedicated infrastructure. In a dedicated system (with sole access to the HD and SSD), its IOPS performance will increase 300% when its hit ratio increases to $2/3$. In the shared infrastructure its IOPS increases by a paltry 50%. With additional clients sharing the system the performance improvement will be even smaller. In a pay-for-services situation, the fact that A is actually subsidizing the improved performance of B may be considered unfair (A may be paying for SSD use, or have development costs for cache friendliness). In some cases, B may also object to receiving additional IOPS if they cost extra.

The *reward allocation model* [8, 9] emulates a client's performance profile when running on a dedicated infrastructure. In this example, the increased IOPS resulting from A's better hit ratio will all be directed to A, while B's allocation will remain unchanged. Furthermore, since the resources are directed towards the more efficient client, the system throughput will be higher. The reward allocation model will modify the ratio in which A and B are served, from $1:1$ to $3:1$, increasing the overall system throughput to 200 IOPS. Furthermore, B continues to receive 50 IOPS, while A triples its throughput to 150 IOPS (as it would expect when its SSD hit ratio increases to $2/3$).

2.1 Related Work

There has been substantial work towards proportional share scheduling for network IO and CPU allocation [10], [24], [7]. These schemes have since been extended to handle the constraints and requirements of storage and IO scheduling [11–14, 17, 21, 23, 25, 27]. Reservation and limit controls for storage servers were introduced in [13, 16, 26]. These models provide strict proportional allocation based on static shares (possibly subject to reservation and limit constraints). In contrast, our work suggests changing shares to adapt to the behavior of the workload, rewarding well-behaved clients by targeted allocation rather than simply distributing the gains over all workloads. This characteristic is a desirable property of multi-tiered storage systems, where changes in access locality (and hit ratio) can drastically alter an application's profile in different execution phases.

The reward allocation model in [8] dealt with sequential IOs in a multi-tiered system. The model in [9] allows parallel operations and uses the measured hit ratio of the SSD to do reward scheduling. However, such hit ratio information is generally not available at the host level.

A number of papers have proposed time-quanta based IO allocation [6, 18–20, 22, 23].The motivation in these schemes is to isolate fast sequential IOs from slower random IOs [19, 22, 23] or segregate slow SSD writes from faster reads [18]; however, we target multi-tiered storage in this paper. Time-quantum based approaches can be seen as a complementary method to our tag-based scheduling approach. The major issue with time quanta based allocation is the latency jitter caused by waiting for all remaining clients to finish their allocated quantum before scheduling pending requests. This also reduces the overall concurrency at the storage device causing lower throughput. In contrast, the method in this paper is a fine-grained allocation where client requests are interleaved at the level of individual requests, preventing the latency jitter.

Algorithm 1: Basic Reward Scheduling Algorithm

```
RequestArrival (request r, client j, time t)
  if Task j Queue empty then
      Add $j$ to set of active clients $\mathscr{A}$;
      $\mathbf{s}\mathrm{Tag}_j = \max(\mathbf{s}\mathrm{Tag}_j, t)$;
      Add $r$ to queue of task $j$ with tag $\mathbf{s}\mathrm{Tag}_j$;
  else
      Add $r$ to queue of task $j$ with timestamp $t$;
ScheduleRequest ( )
  Dispatch request with $\min_j\{\mathbf{s}\mathrm{Tag}_j\colon j \in \mathscr{A}\}$;
AdjustTags (time t)
  $\mathrm{minTag} = \min_j\{\mathbf{s}\mathrm{Tag}_j\colon j \in \mathscr{A}\}$;
  $\Delta = \mathrm{minTag} - t$;
  $\forall\, j \in \mathscr{A}\colon \mathbf{s}\mathrm{Tag}^j = \mathbf{s}\mathrm{Tag}^j - \Delta$;
RequestCompletion (task j, time t)
  $\bar{\Phi}_j$ = UpdateResponseTime($j$, $\Phi_j$);
  Remove completed request from queue;
  $\mathbf{s}\mathrm{Tag}_j = \mathbf{s}\mathrm{Tag}_j + \bar{\Phi}_j/\omega_j$;
  AdjustTags($t$);
  if Task j Queue empty then
      Remove $j$ from set of active clients $\mathscr{A}$;
  ScheduleRequest();
```

3 Reward Scheduling Algorithm

Each client i is assigned a weight ω_i to represent its relative priority. The reward scheduling algorithm maintains a queue per client and a *tag* for each client queue. When invoked, the scheduler dispatches the request with the smallest tag to the storage array. A high level description of the algorithm is shown in Algorithm 1.

In order to do IO cost accounting, the scheduler maintains a *running average* of the response times of the last N (a configurable parameter) requests of each client. Since the storage system is treated as a black box, the response times is the elapsed time between the dispatched time and the time the request completes.

When a request from client j completes service, procedure `RequestCompletion` is invoked. The running average response time of client j, $\bar{\Phi}_j$ is updated to include the completed request using function *UpdateResponseTime*( ). This average is used to compute the next value of $\mathbf{s}\mathrm{Tag}_j$ by incrementing its current value by $\bar{\Phi}_j/\omega_j$. Thus tags of successive requests of the client are spaced by an amount that is proportional to the average response time over the last N requests, and inversely proportional to the static weight of the client. In this way, clients who complete requests faster are given priority over those with slower requests, as are clients with higher static weights.

Procedure `RequestArrival` adds the request to the queue; if it is the only request it assigns it a starting tag equal to the larger of the current time or the last tag value for this client. The `AdjustTags` procedure is needed to synchronize the tags of requests already in the system with newly arriving requests. This is the same synchronization mechanism used in [12, 13].

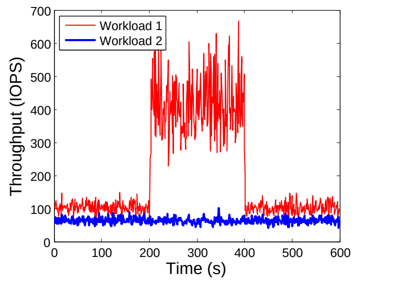
4 Preliminary Evaluation

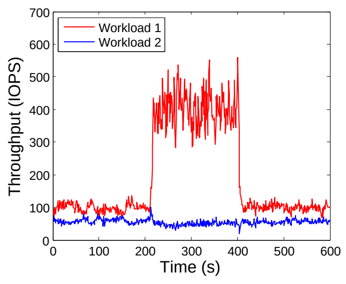
We experimented with two different setups: (1) Simulation-based and (2) Linux-based real system. In first, we created a simulation model of a multi-tiered storage system using Yacsim [15] simulation environment. We created a system with 1 HD and 1 SSD. We used a simple SSD model and compared the qualitative behavior to a storage system consisting of a single HD and a single SSD device. We used SSD read time to be exponentially distributed with a mean of 200μsec. For the HD the service time is exponentially distributed with a mean of 10 ms.

We also implemented a prototype on Linux by interposing a reward scheduler in the IO path in user space. The micro-benchmarks are made to access storage devices through this user-space process. Requests are dispatched to the underlying storage devices using our scheduler. Raw IO is performed to eliminate the influence of buffer caching. We present two experiments showing different aspects of reward scheduling.

Hit Ratio Variation: In this experiment, we used two workloads and varied the hit-ratio in SSD tier of one of the workloads while keeping the other workload's hit ratio at 100%. We also varied the weights of the workloads to three different values. Each workload is issuing 100% random read request with a backlog of 100 requests at all times. Table 1 and 2 show the results using our Linux-based testbed and simulation-based storage system.

Note that the actual achieved ratio in terms of IOPS is different than the weights for different hit-ratio values. This is due to the reward scheduling nature of our algorithm. One can also verify that the IOPS-ratios obtained in various cases are similar to the latency-ratios of different workloads. For example, when hit-ratio = 0.2, the latency of the workload $B = 0.8 \times 10 + 0.2 \times 0.2 = 8.04$ ms, and the latency of workload A is 0.2 ms. In this case the actual ratio of IOPS is close to 40.2. Similar computation can be performed for other cases as well.


(a) Simulation for reward scheduling


(b) Reward scheduling

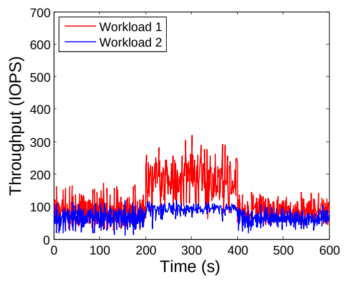

(c) CFQ scheduling

Figure 2: Workload 1 changes hit ratio from 0.6 to 0.9 from time 200s to 400s.

Weights	hit-ratio(A:B)	IOPS (A)	IOPS (B)	IOPS-ratio (A/B)
1:1	1 : 0.2	3106	75	41.4
	1 : 0.5	3055	129	23.7
	1 : 0.75	2816	295	9.6
	1 : 1	2000	2033	0.98
1:2	1 : 0.2	2501	80	31.2
	1 : 0.5	2459	139	17.7
	1 : 0.75	2257	311	7.3
	1 : 1	1358	2682	0.51
2:1	1 : 0.2	3359	72	46.6
	1 : 0.5	3251	125	26
	1 : 0.75	3047	277	11
	1 : 1	2689	1347	2.0

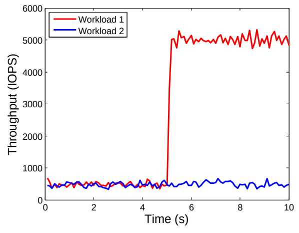
Table 1: Linux-based test: IOPS observed by two workloads when hit-ratio of workload B is varied from 0.2 to 1 and it is kept fixed at 1 for workload A.

Weights	hit-ratio(A:B)	IOPS (A)	IOPS (B)	IOPS-ratio (A/B)
1:1	1 : 0.2	4923	122	40
	1 : 0.5	4841	193	25
	1 : 0.75	4612	373	12
	1 : 1	2474	2474	1.0
1:2	1 : 0.2	4917	124	40
	1 : 0.5	4836	198	24
	1 : 0.75	4538	400	11
	1 : 1	1599	3350	0.48
2:1	1 : 0.2	4931	116	42
	1 : 0.5	4862	173	28
	1 : 0.75	4704	326	14
	1 : 1	3350	1599	2.1

Table 2: Simulation-based test: IOPS observed by two workloads when hit-ratio of workload B is varied from 0.2 to 1 and it is kept fixed at 1 for workload A.

Next we compared the behavior of Reward scheduling and Linux CFQ scheduling. The storage server includes a 1TB SCSI hard disk and 80GB SSD. Two continuously backlogged workloads issued random read requests to the hard disk or the SSD, based on the hit ratio. Each workload had a weight of 0.5. Workload 2 had a fixed hit ratio of 0.4 throughout the experiment. The hit ratio of workload 1 is 0.6 from time 0 to time 200s, which then increases to 0.9 from time 200s to 400s, and then comes back to 0.6 after that.

Figures 2 (a) and (b) show the performance of reward scheduling using simulation and Linux-based setups respectively. Here, workload 1 gets much higher throughput during 200 to 400 second interval and workload 2 sees no appreciable change in performance. Figure 2 (c) shows the performance of Linux CFQ scheduling. In time interval (200-400 seconds), both workloads get more throughput as the system capacity increases due to the higher hit ratio of workload 1. The additional capacity is shared by both workloads. As can be seen, reward scheduling allocates all excess capacity to workload 1, while a fair scheduler tries to equalize the IOs subject to hit ratio constraints. The overall throughput of CFQ scheduler is lower, as is the amount obtained by workload 1.


(a) Queue Length = 1

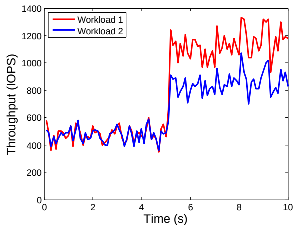

(b) Queue Length = 8

Figure 3: Workload 1 changes read ratio from 0.5 to 1 after 5 seconds.

Read/Write Ratio Variation: In this experiment we used only the SSD as the storage device (*i.e.* hit ratio = 100%) and varied the read write (RW) ratio. Both workloads have equal weights and RW ratio equal to 0.5. At time $t = 5$ seconds, workload 1's RW ratio changes to 1.0. Figure 3 (a) shows the behavior of the two workloads when the disk queue was set to 1.

We then increased the disk queue length and Figure 3 (b) shows the allocation when the disk queue was set to 8. We see that workload 2 got some of the gains and the gain of workload 1 reduced compared to (a). The reason is because the estimator gets diluted as the Q length gets larger, since all of them include the fixed portion of the delay introduced by queuing the requests in the disk queue. This shows the caveats in using average latency as the estimator. We are looking at better estimators that would work more robustly in different situations.

5 Discussion and Open Issues

In this paper we discuss why traditional proportional-share scheduling may be inadequate in shared multi-tiered storage systems, where the access times can vary drastically due to different tiers and workloads phases. The reward scheduling model allocates resources to clients based on their dynamic behavior and emulates their performance as if they were run in isolation. For each client, we allocate IOPS based on its average response time and overall weight.

One key open issue is teasing apart queuing delay or a component of it from the response time. Array vendors can get service times with much better accuracy, since they control the underlying tiers and have relatively smaller queue depth per device. Doing it from outside the array will show some differentiation but not enough in cases of large queue depths.

We are also investigating the relationship between reward scheduling and keeping a queue size per client. Thus for each finished request, only the corresponding queue can issue an extra request. Having static queue sizes won't work, but one can try to make them dynamic by using a control equation like PARDA [11]. Also the behavior of a workload is dependent on the hit-ratio of other workloads. We are looking to reduce this interference as part of future work. We are also investigating if reward scheduling can introduce any feedback loop where a workload can keep on getting higher hit rate due to the preference given by the scheduler.

We hope that the research community will pay more attention to the QoS problem in multi-tiered systems, which are rapidly becoming ubiquitous. Another very relevant variant of this problem is to provide strict latency and IOPS guarantees in such systems by combining techniques for block placement across tiers and scheduling per tier. Doing admission control of workloads in such systems also remains an open and challenging problem that will require better modeling techniques.

References

[1] Nutanix complete cluster: The new virtualized datacenter building block. http://www.nutanix.com/resources.html, 2011.

[2] EMC: Fully automate storage tiering. http://www.emc.com/about/glossary/fast.htm, 2012.

[3] NetApp: Flash cache. http://www.netapp.com/us/products/storage-systems/flash-cache/, 2012.

[4] Nimble storage. http://www.nimblestorage.com, 2012.

[5] Tintri: Vm aware storage. http://www.tintri.com, 2012.

[6] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (1999), pp. 400–405.

[7] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair-queuing algorithm. In *ACM SIGCOMM* (1989), pp. 1–12.

[8] ELNABLY, A., DU, K., AND VARMAN, P. Reward scheduling for qos in cloud applications. In *12th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (2012).

[9] ELNABLY, A., AND VARMAN, P. Application-sensitive qos scheduling in storage servers. In *ACM Symposium on Parallelism in Algorithms and Architecture* (2012).

[10] GOYAL, P., VIN, H. M., AND CHEN, H. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *IEEE/ACM Transactions on Networking* (1997).

[11] GULATI, A., AHMAD, I., AND WALDSPURGER, C. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *In Usenix FAST '09* (2009).

[12] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pclock: An arrival curve based approach for qos in shared storage systems. In *ACM SIGMETRICS* (2007).

[13] GULATI, A., MERCHANT, A., AND VARMAN, P. J. mClock: Handling throughput variability for hypervisor IO scheduling. In *USENIX OSDI* (2010).

[14] JIN, W., CHASE, J. S., AND KAUR, J. Interposed proportional sharing for a storage service utility. In *ACM Sigmetrics* (2004).

[15] JUMP, J. R. Yacsim reference manual. http://oucsace.cs.ohiou.edu/~avinashk/classes/ee690/yac.ps.

[16] KARLSSON, M., KARAMANOLIS, C., AND ZHU, X. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage* (2005), 457–480.

[17] LUMB, C. R., SCHINDLER, J., GANGER, G. R., NAGLE, D. F., AND RIEDEL, E. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. In *Usenix OSDI* (2000).

[18] PARK, S., AND SHEN, K. Fios: A fair, efficient flash i/o scheduler. In *FAST* (2012).

[19] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T. M., AND MALTZAHN, C. Efficient guaranteed disk request scheduling with fahrrad. In *ACM EuroSys* (2008).

[20] SHAKSHOBER, D. J. Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 Kernel. In *In Red Hat magazine* (June 2005).

[21] SHENOY, P. J., AND VIN, H. M. Cello: a disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS* (1998).

[22] VALENTE, P., AND CHECCONI, F. High Throughput Disk Scheduling with Fair Bandwidth Distribution. In *IEEE Transactions on Computers* (2010), no. 9, pp. 1172–1186.

[23] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. Argon:Performance Insulation for Shared Storage Servers. In *In Usenix FAST '07* (2007).

[24] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery scheduling: flexible proportional-share resource management. In *Usenix OSDI* (1994).

[25] WIJAYARATNE, R., AND REDDY, A. L. N. Integrated QoS management for disk I/O. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (1999).

[26] WONG, T., GOLDERING, R., LIN, C., AND BECKER-SZENDY, R. Zygaria: Storage performance as managed resource. In *Proc. of RTAS* (April 2006).

[27] ZHANG, J., SUBRAMANIAM, A., WANG, Q., RISKA, A., AND RIEDEL, E. Storage performance virtualization via throughput and latency control. *Trans. Storage* (August 2006), 283–308.