Reward Scheduling for QoS in Cloud Applications

Ahmed Elnably, Kai Du, Peter Varman
Dept. of Electrical and Computer Engineering
Rice University, USA
Houston, Texas
Email: ahmed.elnably, kai.du, pjv@rice.edu

***Abstract*—We present a novel QoS scheduling algorithm for multi-tiered storage servers made up of hard disks and SSDs. The work is motivated by the difference in access times for a workload as its SSD hit ratio changes. Our scheme is designed to reward clients according to their runtime behavior, while honoring their static QoS settings including shares (or weights), reservations and limits.**

A model based on entitlements is developed to describe the reward allocation policy (RAP). Simulation results show the advantages of RAP over conventional proportional share allocation in adapting to dynamically varying workloads. The proposed algorithm allows the client to directly reap the benefits of application performance tuning as if the client is on a dedicated system, addressing a key complaint of clients when moving to a shared infrastructure.

***Keywords*-QoS, multi-tier storage, cloud, performance isolation, scheduling.**

I. Introduction

The popularity of the cloud as a general-purpose computing platform is raising awareness of the need for versatile QoS policies and mechanisms that can support diverse applications and time-varying workloads [1]. Commercial virtualization technology [2]–[4] has facilitated aggressive resource consolidation, whereby multiple tenants or clients can share computing, storage, and networking infrastructure provided by the service provider [5]–[7]. Customers benefit from ease and speed of deployment, elastic resource availability, and lowered costs for short-term or unknown computing needs, while service providers can reap the benefits of consolidation in terms of reduced infrastructure and management costs.

Despite the benefits of shared infrastructure and pay-as-you-go pricing models over dedicated and owned resources, performance concerns persist, especially in the area of QoS for shared storage and IO. Resource controls for storage servers [8], [9] are still in an early stage of deployment [10], and provide relatively modest functionality. Unlike CPU and memory, storage resource management has to deal with stateful devices and variable service times, which make it much more difficult to provide effective QoS guarantees. The problem will increase as new multi-tiered storage organizations supplant traditional hard-disk SAN and NAS arrays. By employing aggressive SSD-based tiering or caching within a traditional SAN or distributed clustered storage [11], these solutions boost performance at reasonable cost. As a consequence, QoS management now needs to deal with multiple heterogeneous devices with very different access speeds, and applications whose performance changes drastically with changes in the access profiles. In Section II we discuss in more detail the limitations of conventional QoS approaches in such an environment, and how our reward-based model addresses the problem.

In this paper we propose a storage QoS performance model that freshly examines the issue of fair resource allocation in a heterogeneous environment in which server performance is highly dependent on workload characteristics (like SSD hit ratio). We believe that customers that move to a shared infrastructure will increasingly demand to see the benefits of better caching, smart data placement decisions, and good workload behavior, reflected in the performance of their applications in the shared environment, just as they would on a dedicated infrastructure. 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 less-well-performing applications. The subsidy model exemplified by *proportional sharing* is currently the most common solution for resource allocation in storage systems (see Section II). However such solutions implicitly assume homogeneous resources and are less acceptable when server performance can fluctuate with application behavior.

The rest of the paper is organized as follows. In Section II we motivate and define our problem more precisely, and compare it with existing approaches. In Section III we present a formal model for reward-based allocation and summarize its key properties. In Section IV we describe our reward scheduling algorithm. In Section V we extend the model to include reservations and limits in addition to weights (shares). Finally, we present simulation results in Section VI, and conclude in Section VII.

II. Overview

We motivate our reward QoS model using an example. Consider a storage system that is being shared by three clients A, B, and C. In a proportional share resource allocation scheme, each client has a weight (or shares) that reflects

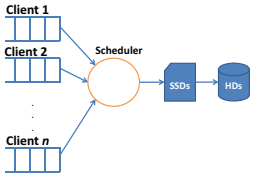


Figure 1. Reward Scheduler for Multi-Tiered Storage

its importance relative to the other clients. Suppose, for example, that the weights of A, B, and C are 1, 2 and 2. If the storage system has a capacity of 100 IOPS, then a traditional proportional share scheduler would allocate 20 IOPS to A and 40 IOPS to each of B and C. The implicit underlying assumption is that all IOs are equally expensive, so that the average service time of each client's requests are roughly the same (around 10 ms). As long as the workloads of the clients satisfy that assumption (say all are doing random 4KB IOs on the device) the situation will be acceptable to them.

Now suppose that the characteristics of A's workload changes and it is able to effectively exploit caching on front-end SSDs. For specificity, assume that its SSD hit rate is around 50% so that the average service time of its requests is 5ms. The proportional share scheduler would continue to allocate IOs to A, B, and C in the ratio $1:2:2$. Since A's IOs take 5ms and B and C's IOs take 10ms each, the system will complete IOs at the rate of 5 IOs every 45 ms or 111.1 IOPS. Of this capacity, A will get 22.2 IOPS while B and C will each get 44.4 IOPS each.

There are two drawbacks to using the proportional share scheduler in this situation. From *client A's perspective* the allocation is quite unfair. The additional system capacity has arisen entirely due to its more efficient IOs (which presumably required careful application structuring, and may also incur costs for use of the SSD). Yet its throughput increases by only 11%. In contrast, in a dedicated environment the increase in A's hit ratio could have doubled its throughput. From the *service provider's perspective*, better resource utilization is possible if more of the faster IOs are done at the expense of the slower ones. It is reasonable to continue to give B and C the 40 IOPS they were receiving, and divert all the additional capacity to the client generating it (in this case A). Hence, the allocation of A will be doubled, so that the clients receive service in the ratio $2:2:2$. With this change, the server capacity is now 120 IOPS (6 IOs every 50 ms); and all three clients receive 40 IOPS. Client A has doubled its throughput as it would have in a dedicated environment, and B and C are not affected by the increase in A's allocation.

A. Related Work

There has been substantial work dealing with proportional share schedulers for networks and cpu [12], [13], [14]. These schemes have since been extended to handle the constraints and requirements of storage and IO scheduling [8], [9], [15]–[22]. Reservation and limit controls for storage servers were introduced in [9], [23], [24]. These models provide strict proportional allocation based on static shares (possibly subject to reservation and limit constraints). In contrast, our work suggests dynamically changing shares to adapt to the characteristics 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.

A number of papers deal with time-quanta based IO allocation [15], [25]–[30]. The motivation in these schemes is to isolate fast sequential IOs from slower random IOs [15], [26], [27] or segregate slow SSD writes from faster reads [25]; 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. 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.

Methods for accurate accounting of VM IO resource usage were presented in [31], [32].

III. Scheduling Model

A model of the tiered storage system consisting of a front-end SSD cache and back-end disks is shown in Figure 1. If a request to the server is found in the SSD cache, it is termed a *hit* and served from the SSD; else it is a *miss* and is served from the hard disk subsystem. The access times for the SSD and disk are denoted by τ and Υ respectively, where $\tau << \Upsilon$. Typical values of these parameters for read accesses are 50μs and 10ms respectively.

There are n clients that share the server. Each client j has a *weight* ω_j. The weights represent the relative static priorities of the clients. The system maintains n queues to hold pending requests, one for each client, as shown in Figure 1. Requests within a queue are served in FCFS order. When the server is free, the scheduler chooses the first request from one of the non-empty queues and dispatches it to the server. The server checks if the request is a SSD hit or a miss and serves the request from the appropriate device.

Policy	Total	Client$_i$
RAP	$\sum_{i\in\mathcal{A}}(f_i \times E_i)$	$E_i \times f_i$
PS	$1/\sum_{i=1}^{n}(f_i/E_i)$	$f_i/\sum_{i\in\mathcal{A}}(f_i/E_i)$

Table I
THROUGHPUT ALLOCATIONS UNDER RAP AND PS.

We now consider the performance of a single continuously-backlogged client running in isolation. Define the *hit ratio* h_i of client i to be the fraction of its requests that are served from the SSD. The average service time for client i is given by $\Phi_i = \tau \times h_i + \Upsilon \times (1 - h_i)$. When run in isolation on the server, client i will receive a throughput equal to $1/\Phi_i$. We call this throughput the *entitlement* of client i, and denote it by E_i. Note that E_i depends only on the client's hit ratio h_i and speed of the devices.

When several clients are sharing the server, the aim of the scheduling policy is to try and provide clients with the same behavior that they would see in a dedicated system. If the system has sufficient capacity then each client should receive its entitlement. However, if the system has less than this capacity, our policy is to allocate the capacity to clients in *proportion to their entitlements*. The policy is formally specified below.

Reward Allocation Policy: Given a set of clients with static weights ω_i and entitlements E_i, allocate capacity to the clients in the ratio $w_i \times E_i = w_i/\Phi_i$, $i = 1, \cdots, n$.

We illustrate the policy with the three clients in the example of Section II. With an initial hit ratio of zero for all clients we have $\Phi_A = \Phi_B = \Phi_C = \Upsilon = 10ms$, corresponding to an entitlement of 100 IOPs for each. Since the static weights are $\omega_A = 1, \omega_B = 2$, and $\omega_C = 2$, the initial allocations are in the ratio of $\omega_i \times E_i$, which is $100 : 200 : 200$ as expected. When the hit ratio of A changes to roughly half, $\Phi_A = 5ms$ and $E_A = 200$ IOPS, while E_B and E_C are unchanged at 100 IOPS. Hence, the allocations are made in the ratio $200 : 200 : 200$, and the absolute allocations are 40 IOPS each (since the system capacity has increased to 120 IOPS because of the efficient IOs by A).

The allocations made by the Reward Allocation Policy (RAP) and Proportional Scheduling (PS) are summarized in Table I. We denote by $\mathcal{A}$ the set of active clients, and consider an interval in which all active clients are continuously backlogged. We let $f_i = w_i/\sum_{j\in\mathcal{A}} w_j$ denote the normalized weight of active client i. Note that $\sum_{i\in\mathcal{A}} f_i = 1$. It is easy to see why the throughputs stated in Table I hold. For RAP, clients receive allocations in the ratio $\omega_i \times E_i$, and IOs of client i require average time Φ_i. Hence, the system performs $\sum_{i\in\mathcal{A}}(\omega_i \times E_i)$ IOs in time $\sum_{i\in\mathcal{A}}(\omega_i \times E_i \times \Phi_i) = \sum_{i\in\mathcal{A}}\omega_i$. Now, $\sum_{i\in\mathcal{A}}(\omega_i \times E_i) = \sum_{i\in\mathcal{A}}(f_i \times E_i) \times \sum_{j\in\mathcal{A}}\omega_j$. Hence, the system throughput is $\sum_{i\in\mathcal{A}}(f_i \times E_i)$. For PS, the system performs $\sum_{i\in\mathcal{A}}\omega_i$ IOs in time $\sum_{i\in\mathcal{A}}(\omega_i \times \Phi_i)$. By dividing and simplifying, the system throughput is $1/\sum_{i=1}^{n}(f_i/E_i)$.

Comparing PS and RAP experimentally: Consider two clients A and B of equal weight that are run on a server with characteristics $\Upsilon = 5$ms and $\tau = 0.1$ms. Each client is kept continuously backlogged.

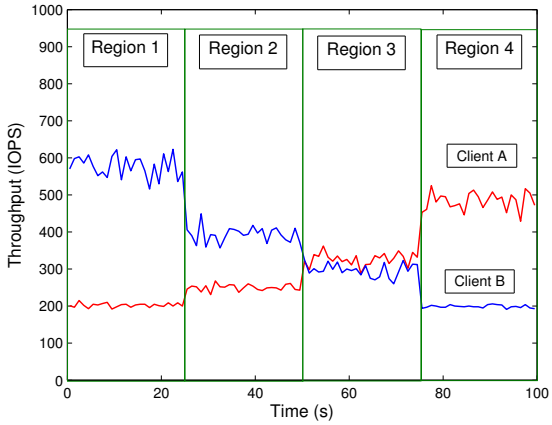
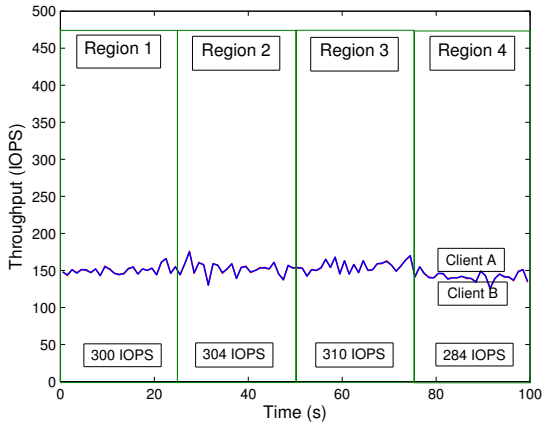


Figure 2. A and B run separately. Throughput is the clients entitlement.

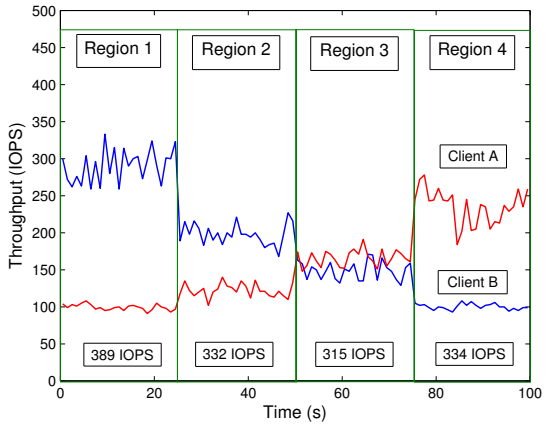
The time interval is divided into four regions, and a client's hit ratio is different in each region. Client A has a hit ratio of 0, $1/5$, $2/5$ and $3/5$ in regions 1 through 4 respectively, while client B has hit ratios of $2/3$, $1/2$, $1/3$ and 0. Figure 2 shows the throughput achieved by A and B in each region when run *separately by itself* on the system. By definition this is the *entitlement* of the client, and it depends only on its hit ratio and the device speeds. Figures 3(a) and 3(b) show the throughputs of A and B when both clients are *run together* on the same server, using PS and RAP respectively. Under PS the clients receives equal bandwidth because their weights are the same and the relative allocations made by PS are insensitive to hit ratio. The differing hit ratios change the overall system capacity a little as shown in the legend in each region. In contrast, RAP behaves very differently, and in each region gives A and B service in proportion to their entitlements in that region. Hence, the form of Figure 2 and Figure 3(b) are identical except that each throughput is divided by 2 since two equal-weight clients are sharing the server. The total system throughput in each region also exceeds the system throughput under PS as expected. One can verify that the results are consistent with the entries in Table I.

IV. REWARD SCHEDULING ALGORITHM

The reward scheduling algorithm to implement RAP is shown in Algorithm 1. Each client has a private queue to



(a) Clients A and B run together using PS.



(b) Clients A and B run together using RAP.

Figure 3. Comparison of allocation when clients are run separately using PS and RAP. PS does not distinguish the behavior of the clients within a region, while the allocation by RAS is proportional to the clients' entitlements. Overall system throughput under RAP exceeds that under PS.

buffer its requests until they are dispatched to the storage system. Each request is stamped with a *tag* which is used by the scheduler to arbitrate between the requests in different queues. Within a client's queue the requests are served in FCFS order (arrival order). Hence, we only require to explicitly tag the request at the head of each client queue. We denote the tag of the first request of client j as $\mathbf{s}\text{Tag}_j$. When invoked, the scheduler chooses the request with the minimum tag value to dispatch to the server.

The scheduler maintains statistics of the *average service times* of each client. In this design we track the average service time of the last N (a configurable parameter) requests of each client. We denote the measured average service time of client j by $\bar{\phi}_j$. Clients with a low value of $\bar{\phi}_j$ get tags that are spaced closer together and are hence served more frequently than requests with larger service times.

Procedure `RequestArrival` shows what happens

Symbol	Meaning
$\mathcal{A}$	Set of active tasks
Φ_j	Average service time of task j
ω_j	Static weight of Task j
$\mathbf{s}\text{Tag}_j$	Scheduling tag of task j

Table II
SYMBOLS USED BY THE SCHEDULER

when a client j sends a request r at time t. If j already has pending requests then r is simply appended to the end of the queue, and its arrival time is noted. It will be assigned an **s**Tag when the request reaches the head of the queue. On the other hand, if the request arrives to an empty queue, then it is assigned a tag equal to the larger of its arrival time and the **s**Tag computed when the last request from that queue completed.

When a request (say from client j) completes service, the procedure `RequestCompletion` is invoked. The average service time of client j, $\bar{\Phi}_j$ is updated by incorporating the service time of the newly completed request. This is used to compute the next value of $\mathbf{s}\text{Tag}_j$ by incrementing its current tag value by $\bar{\Phi}_j/\omega_j$. Thus the tags of successive requests of the client are spaced by an amount that is proportional to the average service time over the last N requests, and inversely proportional to the static weight of the client. Hence clients who are completing their requests faster are given priority over those with slower requests, as are clients with higher static weights.

The `AdjustTags` procedure is needed to handle the dynamic arrival and departure of clients. When a client joins the system by sending a request (either for the first time or after a period of inactivity), the tag assigned to this request needs to be synchronized with the tags of requests already in the system. Before a request is scheduled the tags are adjusted so that the smallest tag is equal to the current real time, but the relative spacing of the tags is unchanged. That is, the tags are moved as a block to synchronize with the current time. This is the same synchronization mechanism used in [9], [16], and is used to prevent starvation of either the newly arriving request or the existing requests.

A. Properties of the Basic Reward Algorithm

The reward scheduling algorithm is designed to reward efficient applications which use less service time at the server by scheduling them more frequently. Consequently, a client whose average service time is smaller will be given preference over a client with a longer service time. In contrast, conventional storage fair schedulers are agnostic to the speed of service, and allocate IOPS to each client based solely on their static weight.

Over the long term, the Basic Reward Scheduler will allocate clients *total service time* in *proportion to their*

Algorithm 1: Basic Reward Scheduling Algorithm

```
RequestArrival (request r, client j, time t)
begin
    if Task j Queue empty then
        Add j to set of active clients A;
        sTag_j = max(sTag_j, t);
        Add r to queue of task j with tag sTag_j;
    else
        Add r to queue of task j with timestamp t;

ScheduleRequest ( )
begin
    Dispatch request with min_j{sTag_j: j ∈ A};

AdjustTags (time t)
begin
    minTag = min_j{sTag_j: j ∈ A};
    Δ = minTag - t;
    ∀ j ∈ A: sTag^j = sTag^j − Δ;

RequestCompletion (task j, time t)
begin
    Φ̄_j = UpdateServiceTime(j, Φ_j);
    Remove completed request from queue;
    sTag_j = sTag_j + Φ̄_j/ω_j;
    AdjustTags(t);
    if Task j Queue empty then
        Remove j from set of active clients A;
    ScheduleRequest();
```

weights, as shown in Table I. Client i receives a throughput of $E_i \times f_i$; since its IOs take an average of $\bar{\Phi}_i$ each, the total service time it receives in T time units is $f_i \times T$. Hence, if two applications have the same weight but their average service times are 5ms and 20ms respectively, then both will spend 500 ms of every second using the device. However, the first application will complete 100 IOs per second, while the second will only get 25 IOPS. As discussed earlier, an IOPS based proportional scheduler would give each application 40 IOPS, penalizing the efficient application over what it would expect to see if it was deployed on dedicated resources.

V. Reservations and Limits Model

In this section we consider an important practical extension to the basic reward model presented in Section II, specifically the incorporation of *reservations* and *limits*. A reservation for client i, denoted by R_i, is a lower bound on IOPS guaranteed to it, while a limit, denoted by L_i is an upper bound on the IOPS provided to it. Clients often like to place an upper-bound in a pay-for-service model to limit costs for low priority jobs or to prevent unexpected charges due to runaway applications. Reservations allow the client to anticipate a certain minimum performance under congested conditions; without reservations the client may receive very little allocation due to its low weight or because of a drop in available system capacity.

In the reward model considered here, reservations are specially important for a client to protect itself from fluctuations in its own workload characteristics. A client may wish to reserve a minimum number of IOPS irrespective of its hit ratio. In this case, even if its entitlement falls due to a poor hit ratio, the reservation will ensure a floor. As an example, consider client B of Figure 3(b). In Region 4, when its hit ratio falls to $1/3$ its throughput falls to around 100 IOPS. The client may wish instead to place a reservation of 150 IOPS so that even in Region 4 it would be guaranteed at least 150 IOPS. In Regions 1 to 3, it would receive more than its reservation (based on its entitlement), but in Region 4 would receive its reservation (more than its entitlement). This situation is unique to the reward model and requires to be carefully handled.

Handling Reservations: Guaranteeing reservations is difficult. The reward scheduler should ensure that each task gets at least its reservation while rewarding the tasks with lower service times. This requires that the system capacity must be sufficient to satisfy the reservations of all admitted clients, even under their worst-case access profiles (for instance, zero hit ratio). That is, admission control must ensure that: $C_{min} \geq \sum_{j \in \mathcal{A}} R_j$. Other, more permissive admission control criteria can be adopted, by suitable SLA restrictions on the client workloads. A detailed discussion of these approaches is beyond the scope of this paper.

We handle reservations and limits using the elegant technique of using multiple tags proposed in [9]. Requests at the head of each queue are assigned a reservation tag (**r**Tag) and a limit tag (**l**Tag) in addition to the **s**Tag described earlier. Successive **r**Tags of client i are equally spaced apart by $1/R_i$. If a client's **r**Tag lags the current time it means that it requires service to meet its reservation. If the **r**Tag is ahead of the current time it means that all its reservations have been currently satisfied and it should be served according to its **s**Tag. The **s**Tags are updated as in Algorithm 1.

A client may dynamically change its access characteristics and move back and forth between being reservation bound or entitlement bound. In this case the scheduler needs to track the change, and synchronize between the **r**Tags and **s**Tags of the client. In [9] the two sets of tags were relatively independent and did not require any synchronization between them. Specifically, when a client switches from being reservation bound to entitlement bound (its hit ratio improves so that its weighted entitlement exceeds its reservation) its **s**Tag needs to be resynchronized with the current **r**Tag value. Otherwise it would face a prolonged delay before receiving its entitlement due to its **s**Tag having run arbitrarily ahead of its **r**Tag. We show this behavior experimentally in Section VI-C. The switch from being reservation bound to being entitlement bound is detected by checking if $\frac{\Phi_i}{\omega_i} < \frac{1}{R_i}$. To prevent instability, we require this condition to hold over a certain minimum number of consecutive requests, to conclude that a genuine change of phase has occurred.

Symbol	Meaning
L_j	**Limit** capacity of task j
R_j	**Reserved** capacity of task j
$\mathbf{r}Tag_j$	Reservation tag of task j
$\mathbf{l}Tag_j$	Limit tag of task j

Table III
ADDITIONAL SYMBOLS USED BY THE REWARD SCHEDULER

Handling Limits: The request at the head of the queue of client i is assigned a limit tag **l**Tag$_i$ that are spaced $1/L_i$ apart. If the current time is less than **l**Tag$_i$, it means that client i has received more allocation than specified by its limit, and should not be serviced till **l**Tag$_i$ falls behind the current time.

Reward Scheduler with Reservations and Limits: The details of the algorithm are given in Algorithm 2. The scheduler searches the active client queues for requests whose **r**Tag are no more than the current time. The request with the smallest such reservation tag is selected for servicing and dispatched. If there are no eligible **r**Tags, the scheduler selects the request with the lowest (**s**Tag), provided its **l**Tag is less than the current time. When a request completes, the service time Φ is recorded to be used in updating the average service time of the client, and for checking if there is a phase transition.

The Reward Scheduling algorithm allocates the system's capacity in the following order: (1) Each client will get at least its reservation capacity provided the system capacity constraint is satisfied. This happens because the reservation tags have the highest priority in the system and requests required to satisfy a reservation are are always served in preference to other requests. (2) If there is additional capacity beyond that needed for reservations, it is allocated in accordance with the dynamic weights ($\bar{\Phi}_i/\omega_i$) of the clients.

Algorithm 2: Reward Scheduling Algorithm

```
RequestArrival (request r, time t, task j)
begin
    if Task j Queue empty then
        sTag_j = max(sTag_j, t);
        rTag_j = max(rTag_j, t);
        lTag_j = max(lTag_j, t);
        Add r to j's queue with sTag_j, rTag_j & lTag_j;
        Add j to set of active clients A;
    else
        Add r to task j's queue with timestamp t;

ScheduleRequest (time t)
begin
    Let E be the set of requests with rTag ≤ t;
    if E not empty then
        Dispatch request with min rTag from E;
    else
        Let E' be the set of requests with lTag ≤ t;
        if E' not empty then
            Dispatch request with min sTag from E';
            Let it belong to task j;
            rTag_j = rTag_j - 1/R_j;

AdjustTag (time t)
begin
    minTag = min_j{sTag_j : non-empty queues j};
    Δ = minTag - t;
    ∀ non-empty queues j: sTag^j = sTag^j − Δ ;

RequestCompletion (task j, time t)
begin
    Φ_j = service time;
    Remove completed request from queue;
    rTag_j = rTag_j + 1/R_j;
    lTag_j = lTag_j + 1/L_j;
    sTag_j = sTag_j + Φ_j/ω_j;
    if Φ_j/ω_j < 1/R_j then
        /* Synchronize sTag if phase change */;
        sTag_j = rTag_j;
    AdjustTag(time t);
    if Task j Queue empty then
        Remove j from set of active clients A;
    ScheduleRequest(time t);
```

VI. EVALUATION

In this section, we describe the empirical evaluation of our scheduling algorithm, *Reward Scheduling*, using a process-driven system simulator Yacsim [33]. Each client workload consists of a sequence of fixed-size block requests. The request service times are assumed to be uniformly distributed with a mean of 5ms for a disk device, and a mean of 0.1ms for the SSD. The workloads are all kept continuously backlogged with at least 1 outstanding request at any time. The experiments are designed to validate three aspects of the Reward Scheduling algorithm: (*i*) Guaranteeing weight-based allocation; (*ii*) Rewarding well behaved tasks; and (*iii*) Guaranteeing Reservations.

A. Guaranteeing Weight Proportional Allocation

This experiment demonstrates that Reward Scheduler reverts to a simple proportional scheduler when the clients have the same workload characteristics. Each client has a fixed static weight based on its SLA. At run time the Reward Scheduler generates the dynamic weight of the client by dividing it by the average service time of its requests. When the clients have the same access profiles, the ratios of their dynamic weights and static weights are the same. We used three clients A, B and C with static weights 1, 2 and 5 respectively. All the clients have a hit ratio of 0 (*i.e.* all disk requests) and reservations of 0. As shown in Figure 4, the Reward Scheduling algorithm successfully divides the

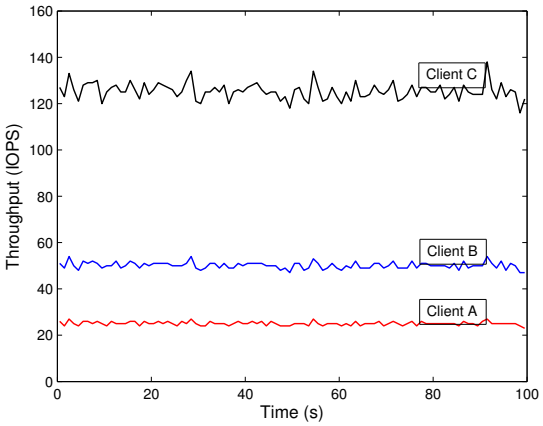


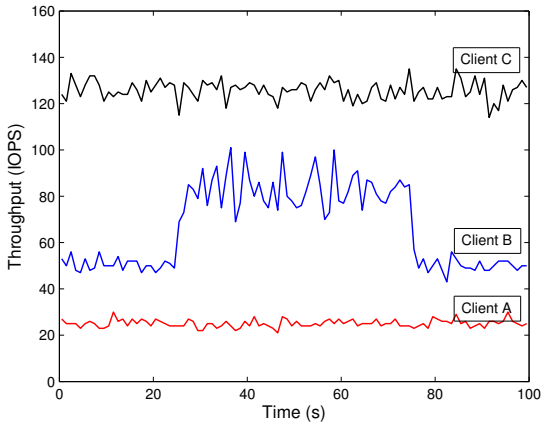
Figure 4. Guaranteeing Weight Proportional Allocation

capacity of the system for all clients in proportion to the weights. The capacity of the system is roughly 200 IOPS when the mean service time equals 5ms. The allocated bandwidths for the three clients are centered around 25, 50 and 125 IOPS respectively, in the ratio of their weights.

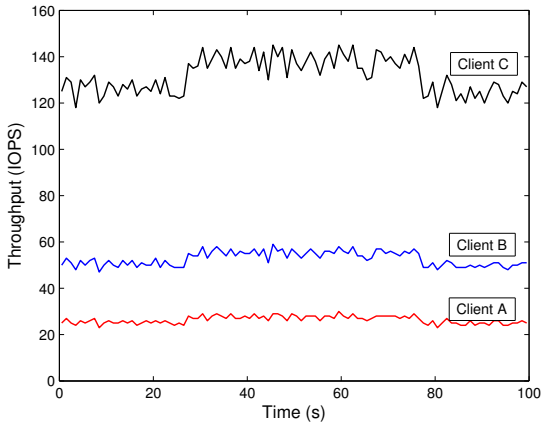
B. Reward Scheduling

Rewarding workloads which have better runtime characteristics is a critical feature of the proposed algorithm. By reward we mean that the task with a shorter average service time is able to obtain a higher share of the IOPS of the server. As in the previous setup, the three clients A, B, and C have weights in the ratio $1 : 2 : 5$, and are kept continuously backlogged. The hit ratios of A and C are kept constant at zero, as is the hit ratio of B before time $t = 25$ and after time $t = 75$. Between times $t = 25$ and $t = 75$, client B's hit ratio is at the higher value of 0.4. During this interval its average service time is equal to 3.04ms, compared to the average service time of 5.0ms when all its requests are to the disk.

The throughput achieved by the tasks is shown in Figure 5(a). During the interval $[25, 75]$ the bandwidth allocated to B increases from 50 IOPS to roughly 83 IOPS, while the throughputs of clients A and C remain constant at 25 and 125 IOPS respectively throughout the experiment. The increase in B's allocation is directly related to the decrease in its service time by a factor of $3/5$ that increases its throughput by a factor $5/3$, while leaving the other allocations unchanged. This is the main feature in our algorithm. In Figure 5(b) we show the same scenario but using a Proportional Scheduler. During the interval $[25, 75]$, the system capacity increases a little from 150 to 222 IOPS. This increase is proportionally distributed among the three clients, which all show a small increase in their throughput. Meanwhile client B which was the main cause for this increase in the total bandwidth, was barely rewarded, as shown by comparing its allocation



(a) Allocation by Reward Scheduler



(b) Allocation by Fair Scheduler

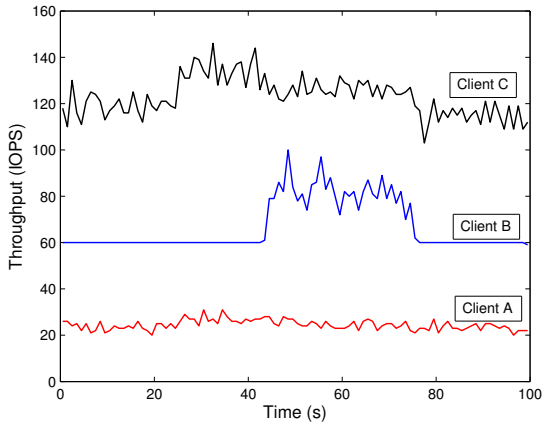
Figure 5. Comparison of Capacity Allocation in PS and Reward Schedulers

during this interval in Figures 5(a) and 5(b).

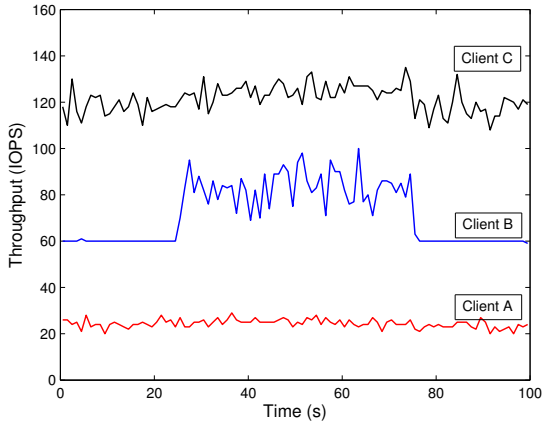
C. Reward Scheduling with Reservations

Finally we put all the pieces together and consider reward scheduling and reservations together. The only change from the previous example is that client B has a reservation of 60 IOPS. Hence during the initial interval $[0, 25]$, client B must be allocated more than its entitlement of 50 IOPS; consequently the bandwidth allocated to clients A and C are reduced proportionately (see Figures 6(a) and 6(b)). At $t = 25$ the hit ratio of client B changed to 0.4.

We ran this experiment for two cases, The first is to show the starvation problem that arises if the **s**Tags and **r**Tags are not synchronized properly when the phase changes. As shown in Figure 6(a), the throughput allocated to B did not increase till roughly time 40 and it received additional bandwidth only for the interval $[40, 75]$. In contrast, in the experiment shown in Figure 6(b) where the tags are synchronized (Algorithm 2), client B receives additional bandwidth



(a) Starvation in Reservation



(b) Reward Scheduling with Reservations

Figure 6. Comparison Between Capacity Allocation With and Without rTag/wTag Synchronization

based on its entitlement almost immediately after its hit ratio changes, as desired. In Figure 6(b), the allocations of A and C are unaffected when the hit ratio changes. In Figure 6(a) the allocations to A and C increase during the interval $[25, 40]$; during this time the system is generating additional IOs but B is not receiving them due to the drift between its tags; hence the capacity is shared between A and C which see an increase in allocation in this interval.

VII. Conclusion

In this paper, we presented a novel QoS scheduling algorithm for multi-tiered storage servers made up of hard disks and SSDs. Our scheme is designed to rewards clients according to their runtime behavior, while honoring their static QoS settings including shares (or weights), reservations and limits. The work is motivated by the difference in access times for a workload in different phases of execution as its hit ratio to the SSD device changes. A model based on entitlements is developed to describe the reward allocation policy (RAP). Simulation results show the advantages of this policy over conventional proportional share allocation, and its ability to adapt to dynamically varying workloads.

We believe that the proposed reward scheduling algorithm leads to QoS guarantees, which are more useful to clients than existing proportional share schemes in multi-tenant shared environments. The proposed algorithm allows the client to directly reap the benefits of application performance tuning as if the client is on a dedicated system. This addresses a key complaint of clients when moving to a shared infrastructure. From the infrastructure's viewpoint, reward scheduling also results in higher throughput. Currently we are in the process of extending the Reward Scheduling algorithm to handle increased device concurrency, and experimenting with actual implementation in a Linux environment with multiple types of storage.

Acknowledgements: The support of the National Science Foundation under NSF Grants CNS 0917157 and CCF 0541369 is gratefully acknowledged.

References

[1] R. N. Calheiros, C. Vecchiola, D. Karunamoorthya, and R. Buyyaa, "The aneka platform and qos-driven resource provisioning for elastic applications on hybrid clouds," *Future Generation Computer Systems*, 2011.

[2] "Vmware vsphere 5: Private cloud computing, server and data center virtualization," http://www.vmware.com/products/vsphere/overview.html, 2011.

[3] "Microsoft server and cloud platform," http://www.microsoft.com/en-us/server-cloud/windows-server/hyper-v.aspx, 2011.

[4] "Xenserver6," http://www.citrix.com/English/ps2/products/product.asp?contentID=683148, 2011.

[5] "Amazon web services," http://aws.amazon.com, 2011.

[6] "Bluelock," http://www.bluelock.com/, 2011.

[7] "Verizon cloud," http://www.verizonbusiness.com/Medium/products/itinfrastructure/computing/, 2011.

[8] A. Gulati, I. Ahmad, and C. Waldspurger, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in *In FAST '09: Proceedings of the 7th Usenix Conference on File and Storage Technologies*, 2009, pp. 85–98.

[9] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling throughput variability for hypervisor IO scheduling," in *USENIX OSDI*, 2010, pp. 1–7.

[10] P. Manning, "Storage i/o control technical overview and considerations for deployment," *VMWare White Paper*, 2010.

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

[12] P. Goyal, H. M. Vin, and H. Chen, "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks," in *Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 1996, pp. 157–168.

[13] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: flexible proportional-share resource management," in *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, 1994.

[14] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair-queuing algorithm," in *ACM SIGCOMM*, 1989, pp. 1–12.

[15] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. Ganger, "Argon:Performance Insulation for Shared Storage Servers," in *In FAST '07: Proceedings of the 5th Usenix Conference on File and Storage Technologies*, 2007, pp. 61–76.

[16] A. Gulati, A. Merchant, and P. J. Varman, "pclock: An arrival curve based approach for qos in shared storage systems," in *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2007, pp. 13–24.

[17] H. Chang, R. Chang, W. Shih, and R. Chang, "Gsr: A global seek-optimizing real-time disk-scheduling algorithm," *Journal of Systems and Software*, vol. 80, no. 2, pp. 198–215, 2007.

[18] J. Zhang, A. Subramaniam, Q. Wang, A. Riska, and E. Riedel, "Storage performance virtualization via throughput and latency control," *Trans. Storage*, vol. 2, pp. 283–308, August 2006. [Online]. Available: http://doi.acm.org/10.1145/1168910.1168913

[19] W. Jin, J. S. Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2004, pp. 37–48.

[20] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Riedel, "Towards higher disk head utilization: extracting free bandwidth from busy disk drives," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation*, 2000, p. 7.

[21] R. Wijayaratne and A. L. N. Reddy, "Integrated qos management for disk i/o," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, 1999, pp. 487–492.

[22] P. J. Shenoy and H. M. Vin, "Cello: a disk scheduling framework for next generation operating systems," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 1998, pp. 44–55.

[23] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance differentiation for storage systems using adaptive control," *Trans. Storage*, vol. 1, pp. 457–480, 2005.

[24] T. Wong, R. Goldering, C. Lin, and R. Becker-Szendy, "Zygaria: Storage performance as managed resource," in *Proc. of RTAS*, April 2006, pp. 125–34.

[25] S. Park and K. Shen, "Fios: A fair, efficient flash i/o scheduler," in *FAST*, 2012.

[26] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn, "Efficient guaranteed disk request scheduling with fahrrad," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008, pp. 13–25.

[27] P. Valente and F. Checconi, "High Throughput Disk Scheduling with Fair Bandwidth Distribution," in *IEEE Transactions on Computers*, no. 9, 2010, pp. 1172–1186.

[28] L. L. Abeni, G., and G. Buttazzo, "Constant bandwidth vs. proportional share resource allocation," in *Multimedia Computing and Systems, 1999. IEEE International Conference on*, vol. 2, 1999, pp. 107–111.

[29] D. J. Shakshober, "Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 Kernel," in *In Red Hat magazine*, June 2005.

[30] 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*, 1999, pp. 400–405.

[31] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *USENIX OSDI*, 1999, pp. 45–58.

[32] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in xen," in *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, 2006, pp. 342–362.

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