Storage Performance Virtualization via Throughput and Latency Control

Jianyong Zhang* Alma Riska† Anand Sivasubramaniam* Qian Wang* Erik Riedel†

*The Pennsylvania State University, {jzhang,anand}@cse.psu.edu, quw6@psu.edu.
†Seagate Research Center, Pittsburgh, {alma.riska,erik.riedel}@seagate.com.

Abstract

I/O consolidation is a growing trend in production environments due to the increasing complexity in tuning and managing storage systems. A consequence of this trend is the need to serve multiple users/workloads simultaneously. It is imperative to make sure that these users are insulated from each other by virtualization in order to meet any service level objective (SLO).

This paper presents a 2-level scheduling framework that can be built on top of an existing storage utility. This framework uses a low-level feedback-driven request scheduler, called AVATAR, that is intended to meet the latency bounds determined by the SLO. The load imposed on AVATAR is regulated by a high-level rate controller, called SARC, to insulate the users from each other. In addition, SARC is work-conserving and tries to fairly distribute any spare bandwidth in the storage system to the different users. This framework naturally decouples rate and latency allocation. Using extensive I/O traces and a detailed storage simulator, we demonstrate that this 2-level framework can simultaneously meet the latency and throughput requirements imposed by an SLO, without requiring extensive knowledge of the underlying storage system.

1 Introduction

On the hardware end, large disk arrays and networked storage are enabling immense storage capacities and high bandwidth access to this storage in order to facilitate consolidated storage systems. Simultaneously, there is a growing demand on the management and workload side to consolidate storage needs. As storage systems become more complex, they are becoming increasingly difficult to deploy, tune and manage. A substantial investment is required not just to procure such systems, but in the cost of personnel to manage them. It is, thus, more attractive economically to out-source the storage services for consolidation at a data center, whether it be within a single business enterprise, or across enterprises. Further, such consolidation can also facilitate data sharing across these services, which is necessary in some environments, e.g. different applications in the supply-chain of an enterprise may need access to the same data for different purposes. Consequently, we see a growing trend of consolidated data centers, where different workloads/applications/services share the storage infrastructure (sometimes referred to as a *storage utility*) for their end-goals.

While such consolidation is attractive economically, and naturally facilitates data sharing when needed, sharing of the underlying storage infrastructure can lead to interference between the users/workloads/services leading to possible violations in performance-based service level objectives (SLO) that may be binding towards ensuring revenue inflow to the data center. For the purposes of the discussions in this paper, without loss of generality, we assume each user belongs to a different class, with a performance-based SLO agreed upon a priori for each class based on a pricing structure. In order to ensure the revenue stream, the data center would need to insulate the users from each other. Such isolation is referred to as *performance virtualization*, since it gives the impression of the storage utility being fully devoted to each user.

In this paper, we present an interposed 2-level scheduling framework that can manage I/O request processing to achieve performance virtualization, that is, meet SLOs for different classes. The frameowrk achieves virtualization by introducing a layer on top of the storage utility which uses very little information about its underlying implementation, i.e., treats the storage utility as a black box. Such an approach is referred to as an "interposed " [7, 9, 4, 8] scheduler between the user requests and the underlying storage utility as depicted in Figure 1. The interposed scheduler can re-arrange and/or delay requests before dispatching them to the storage utility, but it cannot affect the storage utility subsequently. The interposed scheduler acts as a *QoS gateway* between the stream of client requests and the storage back-end. Real storage systems do sometimes have gateways that manage the traffic between the clients and the storage system, such as, the Logical Volume Manager, the SAN virtualization switch, etc [4]. Our interposed scheduler can reside in these gateways and extend the set of functionalities that these devices offer as depicted in Figure 1.

In our system, incoming I/O requests are classified into different classes, with an SLO pre-determined for each class. Note that latency guarantees and throughput guarantees are equally critical to many (if not all) workloads. Thus, it is important to support both SLO types in a unified framework.

The framework includes two levels. The higher level mechanism uses a credit-based rate controller (called SARC), to regulate the stream of requests so that they are insulated from each other. In addition, it gets an estimate of whether the utility is being under-utilized from the lower

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 IEEE

level mechanism (called AVATAR) and tries to distribute this spare bandwidth in a reasonably fair manner across the classes. The lower level AVATAR scheduler uses feedback from monitoring the underlying system to regulate requests, i.e., the amount and the order being dispatched to the storage utility.

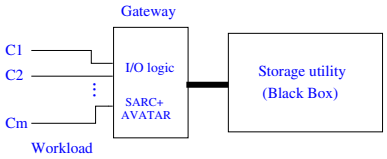


Figure 1. Overall System Model

The rest of this paper is organized as follows. The next section discusses related work. Section 3 gives overview of our framework, together with implementation details of its two components. Section 4 presents results from our evaluation. Finally, section 5 summarizes the contributions of this paper and outlines directions for future work.

2 Related Work

One way to achieve I/O performance virtualization, is to perform resource provisioning/partitioning across workloads in a static manner (e.g. [1, 2]), which is typically coarse-grained. Static (or coarse-grained) resource allocation/partitioning cannot alone handle short-term transient conditions, where much more fine-grained resource management decisions need to be made (usually we may need both kinds of mechanisms in place - resource partitioning at a coarse time granularity, and fine-grained resource control). This paper focuses on the online fine-grained resource control problem.

The main goal of performance virtualization is to meet the throughput and latency requirements (and perhaps fairness issues) of different users/classes, and earlier solutions can be placed in two categories:

1. Schemes in the first category use a proportional bandwidth sharing paradigm. YFQ [3], SFQ(D), FSFQ(D) [7], and CVC [6] use the Generalized Processor Sharing (*GPS*) principle [10]. These schemes are advantageous in that (i) they provide, at least in theory, a strong degree of fairness (as in GPS), and (ii) they are work conserving (i.e. they do not let the resource remain idle when there are requests waiting in some class). However, on the downside, the following problems make them less attractive in a practical setting: (i) They need a performance model to estimate the service time of an individual I/O request. A performance model for a storage system is very difficult to get. (ii) These schemes couple rate and latency allocation together, making them less flexible, potentially leading to resource overprovisioning [6].

2. Schemes in the second category use feedback-based control to avoid the need of an accurate performance model. Even in this category, schemes fall into two classes:

- Schemes that perform *only* rate control, such as Triage [8] (that does adaptive throttling), SLEDS [4] (that uses a leaky bucket) and RW(D) [7] (that implements a window based rate modulator). These schemes can provide performance isolation and have good scalability. However, they suffer from the following drawbacks: (i) Latency guarantees are not necessarily the intended goals, making it difficult to bound response times; (ii) They are not fully work-conserving. [1] (iii) When spare bandwidth in the underlying storage utility is observed, these schemes may not be very fair in distributing this spareness to the different classes.
- Schemes that directly deal with the latency guarantee problem, as in Facade [9] which uses combination of real-time scheduling and feedback-based control of the storage device queue. As we show later in the paper, even though this scheme is simple to implement, it cannot easily isolate performance, and is not fast enough in adapting to transient workload changes.

One can envision our framework as combining the benefits of these two solution categories while avoiding their drawbacks, in the following ways:

– Similar to the schemes in the second category, we use a feedback-based mechanism, without requiring an extensive performance model.

– A high level rate controller, which is lacking from schemes such as Facade, is used to regulate the requests from different classes in order to provide performance isolation.

– In addition, similar to the schemes in the first category, our solution allocates spare bandwidth in the underlying storage utility to the different classes, making it more work-conserving than schemes in the second category while still being relatively fair across the classes.

– Unlike the schemes in the first category, our solution decouples rate and latency allocation, making it more flexible and suitable for meeting multi-dimensional requirements.

3 The 2-Level Scheduling Framework

3.1 Service Level Objectives – SLOs

In our framework, the incoming workload consists of multiple classes, each with a prescribed performance-based Service Level Objective (SLO). The SLO specification for each class i, for $1 \leq i \leq m$, is the tuple $< R_i, D_i >$, where R_i is the maximum class i arrival rate with a latency guarantee requirement of at most D_i . If class i arrival rate is higher than R_i, then its throughput should be at least R_i, but there are no latency guarantee requirements. We call R_i as the rate requirement and D_i as the latency or response time requirement. This SLO specification is enforced in each time interval of predetermined length, i.e., 1 second in our evaluation.

Various schemes that provide latency guarantees [4, 9, 8] maintain the *average* latency within the required SLO bound. However, it is possible that a few fast requests, such as those that represent cache hits offset a large portion of requests that exceed the SLO bound. Consequently in our approach, we use a *statistical latency guarantee*, that is the SLO requires x% (95% in our evaluations) of all requests be bounded by a latency of D_i, for $1 \leq i \leq m$.

[1] Note that RW(D) cannot fully utilizes the degree of concurrency offered by the underlying storage utility.

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 IEEE

3.2 Architecture Principles

Aiming to provide both throughput and latency guarantees, we opt for a 2-level architecture separating rate and latency allocation rather than integrating everything together into a single entity. The higher level of our architecture does traffic regulation via a rate controller, called SARC. The lower level of our architecture provides performance guarantees via a feedback-based controller, called AVATAR. The architecture of our framework is depicted in Figure 2.

The high level rate controller, SARC, is mainly responsible for regulating the incoming traffic to meet the rate requirements and ensuring isolation between classes. SARC aims to fully utilize the underlying storage bandwidth by distributing fairly between all classes any spare bandwidth that is available. SARC manages the incoming traffic in class-based FIFO queues. A credit amount is assigned to each FIFO queue that indicates how many of the outstanding requests can be dispatched to the lower level.

The low level controller, AVATAR, is mainly responsible for satisfying performance guarantees and ensuring effective usage of the storage utility. To meet latency requirements, we use a real-time scheduler, i.e., an Earliest Deadline First(EDF) queue, which orders all incoming requests based on their deadlines. However, employing just an EDF scheduler may reduce overall throughput, because this queue does not optimize the operation of the storage utility. For example, if the utility is a single disk, one prefers a seek-based or position-based optimization scheme [13] rather than EDF. Nevertheless, it is important to strike a good balance between optimizing for latency and optimizing for throughput. AVATAR uses feedback control to dynamically regulate the number of requests dispatched from the EDF queue to the storage utility, where they may get re-ordered for better efficiency. Note that when dispatching a large number of requests to the storage utility the system is being optimized for efficiency and throughput, while restricting the number of requests at the storage utility gives more priority to deadlines.

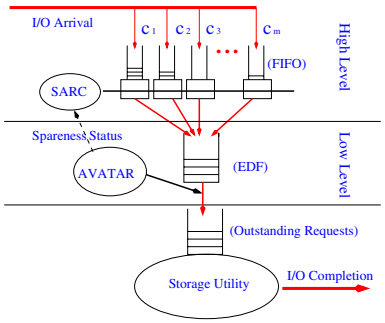


Figure 2. The architecture of our 2-level scheduling framework

We aim to have a work-conserving and fair framework, which requires knowledge of the storage utility utilization status, referred to as the *spareness status*, at the high level rate controller SARC. However, this information is available only at the lower level of our architecture. We use a feedback loop from AVATAR (low-level) to SARC (high-level) which provides to the latter the spareness status of the storage utility.

Upon arrival, an I/O request is tagged with its class id. If the corresponding FIFO queue has any available credit for the request class, then it is dispatched without delay to the low level EDF queue, otherwise it is queued in the FIFO queue waiting until the next replenishment time. In the lower level, AVATAR decides when to dispatch the outstanding requests from the EDF queue to the storage utility. Finally, the storage utility services the outstanding requests based on its own service discipline. Note that our scheme operates outside the storage utility and does not require any changes in it.

3.3 The High Level Rate Controller: SARC

SARC manages the FIFO queues, one for each workload class, and the available credits for each class. Each incoming request consumes only one credit from its class. If one is available, it is dispatched to the lower level; otherwise, it is queued in the corresponding FIFO queue and marked as a backlogged request. SARC has access to the spareness status maintained by AVATAR and described in more detail in Section 3.4.4.

SARC design is related to the amount of class credits and the replenishment policy. Each class i, for $1 \leq i \leq m$, has a maximum amount of credits, denoted by r_i which is directly related to the rate R_i from the class i SLO requirement. Initially, credits of class i are set to r_i. During each replenishment event credits of all classes are reset to full amounts regardless of whether or not any class is devoid of credits. A replenishment event happens

- upon a time period, T_{SARC}, having elapsed since the last replenishment event.
- upon a new arrival at a FIFO queue with no available credits, while the spareness status indicates that the storage utility has spare bandwidth available,
- upon AVATAR changing the spareness status and indicating that spare bandwidth has become available,

The rationale for the periodic replenishment event is similar to the goals of a leaky bucket controller [4], in order to ensure that the storage utility bandwidth is distributed across the classes according to their SLO requirements. The second and third replenishment events make sure that there is no backlogged request at the high level if the storage utility has spare bandwidth available. Note that as a consequence of the above replenishment rules, no two replenishment events are more than T_{SARC} apart.

Upon a replenishment event (whether it be periodic or due to spareness), each class i, for $1 \leq i \leq m$, is replenished to its full r_i credits, where $r_i = T_{SARC} \cdot R_i$. As can be seen, the credit allocation is determined by the rate agreed upon earlier (SLO) to ensure that not more than r_i requests are allowed for any class during T_{SARC} when the storage system is busy.

SARC uses synchronous replenishment, where the credits for all classes are reset at the same time. Note that whenever there are backlogged requests in any class during a replenishment event, they (at most r_i of them) can be dispatched to the low level EDF queue. Intuitively, the synchronous replenishment provides *fairness* in the allocation

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 **IEEE**

(determined by the SLO) of any spare bandwidth across the classes.

If the credit replenishments are done only periodically (i.e. remaining oblivious of the storage utility spareness), the credit-based rate controller would not be *work-conserving*, becoming similar to the rate controllers used in SLEDS [4] and Triage [8]. By tracking underlying spareness and distributing that in a relatively fair manner across the classes, SARC becomes more work conserving, i.e., fully utilizes the storage utility in the presence of backlogged requests.

3.4 The Low Level Controller: AVATAR

In the lower level of our architecture, depicted schematically in Figure 3, there are two queueing centers, i.e., the priority EDF queue and the storage utility queue. The flow of requests between these two queues is controlled by AVATAR. While requests are ordered in the EDF queue according to their deadlines (which are set to the sum of the request arrival time to the EDF queue and its class latency requirement), their dispatching to the storage utility is controlled by the request class latency requirements and the utilization of the storage utility.

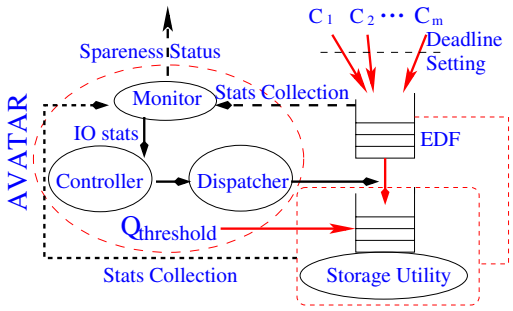


Figure 3. The architecture of the low level scheme

AVATAR is responsible for ensuring that latency requirements of each class are met, making its design and accuracy critically important. Similar to [9], AVATAR uses feedback-based control. However, unlike [9], which is based on various heuristics, AVATAR is based on queueing theory principles and careful approximations. AVATAR combines feedback-based control and Little's law-based bound analysis to periodically set system parameters. We refer to each period, during which AVATAR collects system statistics and adapts system parameters, a *time window*. The most critical parameter set by AVATAR is the threshold of the storage utility queue length. This threshold is used to control request flow within the low level as well as act like an indicator of the spareness status at the storage utility. As we mentioned in Section 3.2, the spareness status is used by SARC, the high level rate controller of our framework.

AVATAR consists of three main components, namely the *Monitor* which continuously collects IO statistics from the underlying system, the *Controller* which determines the queue threshold at the storage utility for the current time window, and the *Dispatcher* which dispatches the requests from the EDF queue to the storage utility queue guided by the queue threshold set by the Controller. In the following subsections we describe the main components of AVATAR in more detail and explain how AVATAR maintains the spareness status.

3.4.1 AVATAR: the Monitor

The monitor collects various statistics from the underlying storage utility and the EDF queue for any time window k ($k > 0$).

- the number of request arrivals, $L^E_{New}(k)$,
- the number of request arrivals whose deadlines lie in the same time window as well, $L^E_{New-Deadline}(k)$,
- the number of completed requests, $X(k)$;
- average waiting time for requests of class i ($1 \leq i \leq m$) at the EDF queue, $MT^E_i(k)$,
- the 95th percentile of response time of class i ($1 \leq i \leq m$) at the storage utility, $T^O_i(k)$,
- maximum number of outstanding requests during the current time window at the storage utility, $L^O_{max}(k)$.

While many of the above statistics are used by the controller as indicators of the system status during the time window k, there are cases when the controller needs to predict the corresponding values for the next time window $(k+1)$. In such cases, we use Last Value Prediction, i.e. estimate the next observation (for time window $k+1$) to be the same as the last measured observation (during the time window k).

3.4.2 AVATAR: the Controller

The critical decision that the controller makes, is the periodical update of the queue threshold at the storage utility, denoted by L^O. The threshold is set based on the measured system performance statistics and workload parameters collected by the monitor. The controller makes decisions at the end of each time window k. In the following, we initially give an intuitive explanation of how the queue threshold L^O is controlled by AVATAR and then continue with its algorithmic details.

If there is abundant available bandwidth in the storage utility to accommodate the deadlines of all outstanding requests, we optimize the system for maximal throughput and increase L^O, because the deadlines will be met regardless of the service order at the storage utility queue. Recall that any scheduling optimizations at the storage utility reorders requests for better throughput (e.g., based on their position on the disk). On the other hand, when the system starts missing deadlines, the emphasis is on meeting the deadline and the latency requirements by shortening the storage utility queue, i.e., decreasing L^O , so that requests with incoming deadlines in the EDF queue are given higher priority and dispatched to the storage utility for service. However, if the system gets overloaded[2], a short queue at the storage utility impacts the storage utility throughput, which causes further cascading of deadline misses. In such extreme cases,

[2]During an overloaded period, the storage utility cannot serve all requests whose deadlines lie in that period under SLO demands.

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 **IEEE**

the preference is to considerably increase the queue threshold. Actually in overload we choose to set the queue threshold L^O equal to infinity so that the system is optimized for throughput and is able to quickly transition to the underloaded state.

We assume that the number of requests arriving during any time window at the storage utility approximates the number of requests that leave it. Based on this assumption, we apply Little's law:

$$ML^O = X \cdot MT^O,$$

where ML^O is the mean queue length, X is the throughput, and MT^O is mean response time at the storage utility. Now let's consider Little's law across two successive time windows. We get

$$\frac{ML^O(k+1)}{ML^O(k)} = \frac{X(k+1) \cdot MT^O(k+1)}{X(k) \cdot MT^O(k)} \quad (1)$$

If the system is under steady load, we relate the average queue length and mean response time with the queue threshold and the 95$^{\text{th}}$ percentile of response time via the following approximations:

$$\frac{L^O(k+1)}{L^O(k)} \approx \frac{ML^O(k+1)}{ML^O(k)}, \quad \frac{T^O(k+1)}{T^O(k)} \approx \frac{MT^O(k+1)}{MT^O(k)},$$

where $T^O(k)$ denotes the 95$^{\text{th}}$ percentile of response time at the storage utility during time window k. By combining the above two relations and the Little's law in Eq.(1), we get

$$\frac{L^O(k+1)}{L^O(k)} \approx \frac{X(k+1) \cdot T^O(k+1)}{X(k) \cdot T^O(k)} \quad (2)$$

In a system that operates in steady load, we determine values/ranges for the queue threshold L^O, based on various guidelines related to workload demands and queueing considerations. Below, we explain these guidelines in more details.

Queue Threshold Requirements Based on Response Time Deadlines, L^O_{RT}:

In Eq.(2), we assume that $X(k+1) \approx X(k)$, for $k > 0$, and get $\frac{L^O(k+1)}{L^O(k)} \approx \frac{T^O(k+1)}{T^O(k)}$. With this approximation, we we calculate the queue threshold at the storage utility as

$$E_i = \frac{D_i^O(k)}{T_i^O(k)} = \frac{D_i - MT_i^E(k+1)}{T_i^O(k)} \quad (3)$$

$${}_iL^O_{RT}(k+1) = E_i \cdot L^O(k) \quad (4)$$

In Equation 3, the class i request deadlines $D_i^O(k)$ at the storage utility are approximated by the difference between their class deadlines at the system that includes the EDF queue and the storage utility and the mean waiting time (MT_i^E) in the EDF queue.

Lower Bound on Queue Threshold Based on the Throughput Required to Meet Response Time Demands, $L^O_{\underline{X}}$:

```
AVATAR(k+1)
 1  /* Called at the end of time window k */
 2  /* Set threshold L^O(k+1) for time window k+1 */
 3  X(k+1) ← L^O_Exist(k) + L^E_Exist−Deadline(k) + L^E_New−Deadline(k+1)
 4  X̄(k+1) ← L^O_Exist(k) + L^E_Exist(k) + L^E_New(k+1)
 5  L^O_X(k+1) ← L^O(k) · X(k+1)/X(k)
 6  L^O_X̄(k+1) ← L^O(k) · X̄(k+1)/X(k)
 7  for class i ← 1 to m
 8  do E_i ← (D_i − MT^E_i(k+1))/T^O_i(k)
 9     if L^O_i(k) < ∞
10        then /* underload case */
11             iL^O_RT(k+1) ← E_i · L^O(k)
12             switch
13                case iL^O_RT(k+1) < L^O_X(k+1) : L^O_i(k+1) ← ∞
14  /* Latency and throughput demand can not be satisfied simultaneously */
15
16                case iL^O_RT(k+1) > L^O_X̄(k+1) : L^O_i(k+1) ← L^O_X̄(k+1)
17  /* Queue threshold does not need to be above the upper bound */
18
19                case iL^O_RT(k+1) < L^O(k) or L^O_max(k) ≥ L^O(k) : L^O_i(k+1) ←i L^O_RT(k+1)
20  /* L^O(k) is decreased to satisfy lat. or increased to improve Tput */
21
22                case iL^O_RT(k+1) ≥ L^O(k) and L^O_max(k) < L^O(k) : L^O_i(k+1) ← L^O(k)
23  /* Balanced state: both latency and throughput demands can be satisfied */
24
25        else  /* overload case */
26             iL^O_RT(k+1) ← E_i · L^O_max(k)
27             switch
28                case X(k+1) ≤ X(k) · 0.9 : L^O_i(k+1) ← max(iL^O_RT(k+1), L^O_X(k+1))
29             /* It transitions to underload state */
30
31                case X̄(k+1) > X(k) · 0.9 : L^O_i(k+1) ← ∞
32             /* It is still in overload state */
33
34  L^O(k+1) ← min_i L^O_i(k+1)
35  return L^O(k+1)
```

Figure 4. Pseudo Code of the Controller

During a time window, at least all requests whose deadlines fall in this time window should be dispatched to the storage utility. This requirement puts a lower bound for the throughput demand that have to be met.

Assuming that $MT^O(k+1) \approx MT^O(k)$, we calculate the queue threshold at the storage utility from Eq.(2) for time window $(k+1)$ as: [3]

$$\underline{X}(k+1) = L^O_{Exist}(k) + L^E_{Exist-Deadline}(k) + L^E_{New-Deadline}(k+1) \quad (5)$$

$$L^O_{\underline{X}}(k+1) = \frac{\underline{X}(k+1)}{X(k)} \cdot L^O(k) \quad (6)$$

Upper Bound on Queue Threshold Based on Adequate Throughput, $L^O_{\overline{X}}$:

It is not desirable to increase the queue threshold with no boundary. Otherwise, it may incur oscillations when deadlines are violated and the queue threshold needs to be decreased. Thus we determine a maximum queue threshold $L^O_{\overline{X}}$ when the system tries to optimize only for throughput. Since it is suffice to serve all outstanding requests and the newly arrived requests in the EDF queue by the end of a time window k, similar to the lower throughput demand, we

[3] $L^O_{Exist}(k)$ denotes requests that are in the storage utility at the end of time window k. $L^E_{Exist-Deadline}(k)$ denotes requests in the EDF queue at the end of time window k, whose deadlines fall in the next time window.

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 IEEE

calculate the upper bound demand as: [4]

$$\overline{X}(k+1) = L^O_{Exist}(k) + L^E_{Exist}(k) + L^E_{New}(k+1) \quad (7)$$

$$L^O_{\overline{X}}(k+1) = \frac{\overline{X}(k+1)}{X(k)} \cdot L^O(k) \quad (8)$$

Pseudo Code of the Controller Algorithm

The three statistics that we introduced previously, i.e., L^O_{RT}, $L^O_{\underline{X}}$, and $L^O_{\overline{X}}$, provide the basis for the algorithm of the AVATAR controller. Once these statistics are computed, we evaluate the queue threshold, $L^O_i(k+1)$, for each class i, for $1 \leq i \leq m$. Note that the class specific queue threshold serves as a guideline only for the requests of that specific class. Actually, we use the minimal computed queue threshold across all classes of requests, as to put the most stringent requirements and meet the response time demands.

Due to space limit, we only show the pseudo code of our algorithm in Figure 4. Figure 4 includes some comments that give explanation about our algorithm. The reader is referred to [14] for more details.

3.4.3 AVATAR: the Dispatcher

The role of the AVATAR dispatcher is to dispatch requests from the EDF queue to the underlying storage utility. By default any request in the EDF queue that missed its deadline is dispatched to the storage utility queue regardless of the queue threshold of the latter. Further, the dispatcher selects as many requests from the EDF queue as necessary to bring up the number of the outstanding requests at the storage utility to the queue threshold at any of the following times: (i) when new requests arrive at the EDF queue; (ii) when requests depart from the storage utility upon completion of service; (iii) at the beginning of each time window.

3.4.4 Spareness detection

As mentioned in Section 3, it is of critical importance for AVATAR to maintain the spareness status, so that our approach remains work-conserving. AVATAR sets the queue threshold so that the storage utility is fully utilized and the SLO requirements are not violated. We consider the queue threshold L^O as the degree of concurrency at the storage utility. Thus if the number of actual outstanding requests is less than the degree of concurrency, we consider the storage utility to have spare bandwidth. Because AVATAR characterizes the storage utility state as either underloaded or overloaded, we consider spareness detection in both of these two system states.

If the system is in the underloaded state, we compare the current number of outstanding requests (L_{curr}) and the queue threshold (L^O). If $L_{curr} < L^O \cdot 0.9$, we consider the storage utility to have spare bandwidth. We use 90% of the queue threshold for stability reasons and avoid prediction errors that would lead to assuming there is spare bandwidth when there is none available. If the system is in the overloaded state, then it is clear that the storage utility has no spare bandwidth. Since the spareness status can change only (i) when requests are dispatched to the lower level; (ii) when requests depart from the storage utility; (iii) at the begining of an overloaded time window, we update the spareness status using the above guidelines only at these points in time.

[4] $L^E_{Exist}(k)$ denotes requests that are in the EDF queue at the end of time window k.

4 Experimental Evaluation

We evaluate our 2-level framework via simulation-based analysis driven by real workload traces. We use Disksim 2.0 [5] as the detailed storage system simulator. The simulated underlying storage utility is a RAID 5 system with 8 Seagate Cheetah9LP 10K RPM disks. The RAID system uses a SCSI interconnect with 80MB/s transfer rate. The array controller has 128 MB cache and each individual disk has 1 MB of cache. The write policy is "write-through combining with immediate report" (i.e., each write request is reported to be complete as soon as it is cached and the data is sent immediately to the disk).

We designed and conducted various experiments that illustrate the performance guarantees of our framework. In all our experiments, we assume that the system operates under a two-class workload.

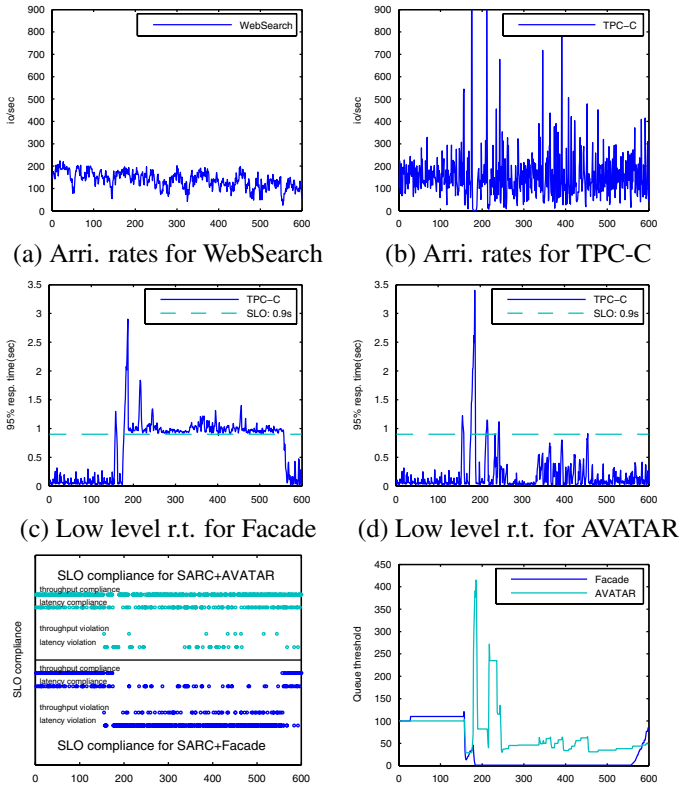
4.1 E1: Adaptability of AVATAR

Our first experiment (E1) focuses on the adaptability of AVATAR in meeting the throughput and latency requirements. We compare AVATAR with the only other scheme that uses feedback-based control and similar SLO representation, Facade [9]. We provide the high level rate controller SARC to both schemes, i.e., AVATAR and Facade, as to evaluate a system with the same level of performance isolation.

In this experiment, the simulation is driven by workload which is the mix of two I/O traces measured in real systems and run for 600 seconds. The first trace [12] comes from a system that deploys a Web Search Engine. The second trace [15] comes from an OLTP-type of system set up according to the TPC-C specifications, where the entire database size is approximately 25 GB. The arrival process characteristics of these two traces are shown in Figure 5(a) and (b), respectively. We set the SLOs to be $(160io/s, 400ms)$ for the WebSearch class (class one) and $(190io/s, 900ms)$ for the TPC-C class (class two). Figure 5(b) shows that the TPC-C class is bursty. There are times when its arrival rates are above $1000io/s$ (truncated in Figure 5(b)) resulting in a transient overload at the storage utility.

In Figures 5 (c), (d), and (e), we present the experimental results for the TPC-C class only. The results for the WebSearch class are similar and we omit them here for lack of space. Figures 5(c) and (d) show the 95$^{\text{th}}$ percentile of request response times at the low level. Note that in our framework, if the request deadlines of a class are satisfied at the low level, then both throughput and latency requirements for that class are satisfied overall. Thus we use the response times at the low level as an indicator of performance guarantees. In order to illustrate the bigger picture of performance guarantees, Figure 5 (e) shows the SLO compliance of the two schemes in terms of whether the scheme meets the throughput requirements when the arrival rate is

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 IEEE



(a) Arri. rates for WebSearch (b) Arri. rates for TPC-C

(c) Low level r.t. for Facade (d) Low level r.t. for AVATAR

(e) SLO compliance for TPC-C (f) Queue threshold per period

Figure 5. Illustrating the adaptability of AVATAR (E2). The x-axis is time (sec).

higher than that specified in the SLO, whether it meets the latency bound when the arrival rate is lower, or it does not meet the appropriate SLO.

Figure 5 (e) shows that SARC+AVATAR has a much better SLO compliance than SARC+Facade. For instance, there are many more periods of latency violation (and also throughput violations) for SARC+Facade than SARC+AVATAR. Figures 5 (c) and (d) give insights on this behavior. SARC+AVATAR satisfies all request deadlines except for a few overloaded intervals (Overall the miss ratio for SARC+AVATAR is less than 5%). Observe that during the time interval of $(180, 580)$ seconds, the response time under (SARC+AVATAR) is less than the SLO latency requirement most of the time, when under SARC+Facade the request deadlines are continuously missed (Overall the miss ratio for SARC+Facade is above 50%). The reasons behind this difference in performance lies in the different ways that Facade and AVATAR handle overload.

According to the storage utility queue lengths, presented in Figure 5(c), the overload starts approximately at the 176^{th} second. Facade detects overload by monitoring *only* the arrival rate and comparing it with the SLO rate requirement. Because SARC regulates the rate of requests dispatched to the low level, Facade does not consider the system to be overloaded at this time. However, since the system starts missing deadlines, Facade aggressively reduces the queue threshold, L^O, at the storage utility, which reaches as low as 1 (see Figure 5(f)) at approximately the 180^{th} second. This low value remains until approximately the 580^{th} second, even though the load is not always high during the entire period. The correct action during the transient overload is to increase the threshold considerably so that the storage utility is optimized for throughput rather than latency. AVATAR uses current system conditions rather than arrival rate to immediately detect, handle, and recover from the transient overload. Observe in Figure 5(f) that AVATAR is able to set the queue threshold high and adapt to transient overload.

Experiment 1 emphasizes the importance of detecting overload in a timely manner, optimizing the system performance for throughput, and recovering from the transient overload condition. The adaptability in AVATAR is related to the wide range of system statistics and metrics that it uses to make decisions, as explained in the previous section, which is missing in Facade.

4.2 E2: Spare Bandwidth Utilization

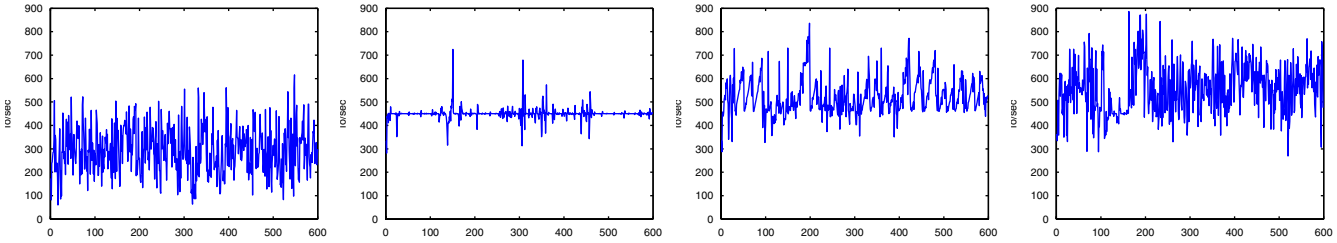
Experiment E2 is designed to evaluate the effectiveness of our high level rate controller SARC in utilizing spare bandwidth. We compare its performance with two other rate controllers that do not use spareness detection. The first approach, called FIXED, periodically replenishes full credits for each class, with no other replenishments otherwise. FIXED is in the same spirit as the leaky bucket mechanism with fixed parameters. The second scheme, called ADAPTIVE, sets the full credit amounts for each class at 1 second intervals. ADAPTIVE is similar to FIXED except that it changes the full credit amounts for classes every second adaptively based on the SLO requirements and load conditions. ADAPTIVE is in the same spirit as SLEDS [4]. See [14] for more details on the ADAPTIVE approach. Note that for the ADAPTIVE scheme, we tried various values for its critical parameters, i.e., P_{rt}, P_{inc}, and P_{dec} (see [14]), and select the set that generates the best results in each experiment. All three rate controllers use AVATAR as the low level scheduler.

	FIXED	ADAPTIVE	SARC
Openmail	249.3	288.8	295.4
TPC-C	199.9	234.0	261.2
Total	449.2	522.8	556.6

Table 1. Avg. Tput. (io/sec) for E2.

We use two real traces in this experiment. The first trace is obtained measuring Openmail [11] in a production environment with thousands of users. We use the TPC-C trace, explained in the previous experiment, as the second workload class, but scale down its inter-arrival times to increase its burstiness. The arrival process for the Openmail trace is shown in Figures 6(a) (The arrival process of the TPC-C trace has shown in Figure 5(b)). We use loose SLOs which are $(200io/s, 600ms)$ for TPC-C and $(250io/s, 900ms)$ for Openmail. With these SLOs, spare bandwidth is available and can be utilized to improve throughput. The entire

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 IEEE



(a) Arri. rates for Openmail (b) FIXED+AVATAR Aggr. Tput. (c) ADAPTIVE+AVATAR Aggr. Tput. (d) SARC+AVATAR Aggr. Tput.

Figure 6. Utilizing the spare bandwidth (E3). The x-axis is time (sec).

trace runs for 600 seconds. In this experiment, the ADAPTIVE's parameters are set to: $P_{rt} = 20\%$, $P_{inc} = 2.5\%$, and $P_{dec} = 5\%$.

Both classes are bursty and the average aggregate rate is more than the rate the system can handle causing the system to operate close to its capacity most of the time. The stipulated SLOs are satisfied by all three schemes, i.e., the miss ratios are less than 5%. Figures 6(b), (c) and (d) show the aggregate throughput for both classes for the three schemes. One can visually see evidence of higher aggregate throughput delivered by SARC, compared to the other two rate controllers. In general, there are more spikes leading to higher throughput over the duration of the experiment, showing that there is better adaptability in utilizing spare bandwidth. Even the average throughput for the entire experiment, presented in Table 1, indicates that SARC's throughput is 10% and 5% higher than FIXED's and ADAPTIVE's throughputs, respectively. Note that the higher the throughput for each class (and not just the aggregate throughput), the better the fairness on distributing spare bandwidth.

5 Concluding Remarks

This paper has presented a new 2-level framework for meeting multi-dimensional performance virtualization goals in deploying shared storage systems. It can accommodate several workloads accessing the underlying storage utility, while meeting their individual Service Level Objectives (SLOs). Our proposed framework can successfully guarantee critical properties such as work-conservation, fairness, performance guarantee, and isolation. We have conducted a thorough analysis of these issues and formally proved the fairness property. The reader is referred to [14] for more details.

Instead of requiring a detailed performance model, the low level scheduler of our framework - AVATAR - uses feedback to control the relative importance of deadline vs. throughput based scheduling of requests for fast adaptation to transient conditions. In addition, the high level rate controller, SARC, is not only able to isolate the classes from each other, but can also fairly distribute any spare bandwidth to these classes towards providing better aggregate throughput.

Both SARC and AVATAR algorithms are not computationally expensive, and in fact, most of the code is not in the critical path of requests. Only request dispatching from FIFO queues to underlying storage that follow the EDF order is in the critical path with a computation cost of $O(\lg m)$ where m is the number of classes. Although the decision-making in the proposed framework is somewhat centralized, we believe that it is scalable for small and mid-size systems. In very large storage systems, the performance virtualization needs to be hierarchical anyway and we are currently evaluating how the proposed 2-level framework can be used as a basic building block for a more decentralized hierarchical solution.

Acknowledgements: This research has been supported in part by NSF grants 0325056, 0130143, 0429500, 0097998, and through grants from IBM and the Pittsburgh Digital Greenhouse.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of FAST*, pages 175–188, January 2002.

[3] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *ICMCS, Vol. 2*, pages 400–405, 1999.

[4] D. Chambliss, G. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. Lee. Performance virtulization for large-scale storage systems. In *Proceedings of SRDS*, October 2003.

[5] G. Ganger, B. Worthington, and Y. Patt. *The DiskSim Simulation Environment Version 2.0 Reference Manual*. http://www.pdl.cmu.edu/DiskSim/.

[6] L. Huang, G. Peng, and T.-C. Chiueh. Multi-dimensional storage virtualization. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[7] W. Jin, J. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of SIGMETRICS*, June 2004.

[8] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proceedings of the 9th International Workshop on Quality of Service (IWQoS 04)*, 2004.

[9] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Proceedings of the Conference on File and Storage Technology (FAST'03)*, pages 89–102, April 2003.

[10] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single Node Case. *IEEE/ACM Transactions on Networking*, June 1993.

[11] The Openmail trace. http://tesla.hpl.hp.com/private_software/.

[12] WebSearch trace. http://traces.cs.umass.edu/storage/.

[13] B. Worthington, G. Ganger, and Y. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1994.

[14] J. Zhang, A. Riska, A. Sivasubramaniam, Q. Wang, and E. Riedel. An interposed 2-level i/o scheduling framework for performance virtualization. Technical Report CSE-05-003, CSE, PSU. http://www.cse.psu.edu/~jzhang/tr-2level.pdf.

[15] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the Usenix Technical Conference*, June 2001.

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 **IEEE**