Lexicographic QoS Scheduling for Parallel I/O*

Ajay Gulati
Department of Computer Science
Rice University
Houston, TX - 77005
gulati@rice.edu

Peter Varman
Department of Electrical and Computer Engineering
Rice University
Houston, TX - 77005
pjv@rice.edu

ABSTRACT

High-end shared storage systems serving multiple independent workloads must assure that concurrently executing clients will receive a fair or agreed-upon share of system I/O resources. In a parallel I/O system an application makes requests for specific disks at different steps of its computation depending on the data layout and its computational state. Different applications contend for disk access making the problem of maintaining fair allocation challenging.

We propose a model for differentiated disk bandwidth allocation based on lexicographic minimization, and provide new efficient scheduling algorithms to allocate the I/O bandwidth fairly among contending applications. A major contribution of our model is its ability to handle multiple parallel disks and contention for disks among the concurrent applications. Analysis and simulation-based evaluation shows that our algorithms provide performance isolation, weighted allocation of resources, and are work conserving. The solutions are also applicable to other shared resource environments dealing with non-uniform heterogeneous servers.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.4.2 [**Operating Systems**]: Storage Management—*allocation/deallocation strategies, secondary storage*

General Terms

Algorithms, Management, Performance

Keywords

Fair Scheduling, Lexicographic minimization, Parallel I/O, QoS, Resource allocation, Storage virtualization

*This work was partially supported by the National Science Foundation under Grant CCR-0105565.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA'05, July 18–20, 2005, Las Vegas, Nevada, USA.
Copyright 2005 ACM 1-58113-986-1/05/0007 ...$5.00.

1. INTRODUCTION

Storage requirements of institutional and commercial organizations are growing at an ever increasing rate. Commercial enterprises increasingly depend on the continuous availability of large-capacity storage systems, which can provide high, predictable performance on a number of transactional and business applications. In addition, many modern scientific and engineering workloads require access to large data sets that reside on shared storage facilities. These trends suggest that storage systems will increasingly be responsible for serving varied workloads from different organizational units. Advancements in storage and networking technologies together with economic advantages of consolidated management, are driving a growth in the deployment of large data centers that serve the storage needs of several departments or separate institutions.

Managing a shared storage system is a complex operation. In addition to clients' concerns regarding data security and privacy, sharing requires consideration of issues of fairness in allocating resources among concurrent applications and efficient utilization of shared resources. Insulating concurrent users from each other (a process referred to as virtualization) so that each client appears to be running in isolation is a major goal in these systems. This requirement is often formalized in the form of a service-level agreement between the storage service provider and the client, guaranteeing a certain minimum quality-of-service (QoS). While some of the problems of virtualization can be mitigated by partitioning or replicating the resources to provide contention-free access to different clients, these approaches raise other problems of their own: first there is no clear way to estimate the requirements and do the partitioning, and secondly over-provisioning can be very expensive in terms of initial setup and maintenance costs. The unique issues that make sharing in storage systems especially difficult and different from other shared environments like networking, has been lucidly described by Wilkes [26]. The problems include the difficulty of predicting response time and throughput due to the strong dependence on data layout and current system state, hot spots and disk conflicts among concurrent flows, heterogeneous disk systems, and the inability to drop requests under heavy load in a storage system.

In this paper, we present a model of fair allocation of disk bandwidth among concurrent clients (flows) sharing a parallel I/O system. Previous work on prefetching and caching in parallel I/O systems has dealt with the problem of a single flow; efficient algorithms are now known that maximize disk system throughput for a single flow (see [1, 16, 18–20] for example). However, the problems of sharing the parallel disks among multiple concurrent flows has not been addressed previously. It has been recently shown that the problem of maximizing the throughput of a parallel disk system for two or more flows is NP-complete, even in the idealized

model where each I/O takes unit time [13]. The problem of allocating disk bandwidth fairly among concurrent competing flows in a parallel I/O system has, to our knowledge, not been formally addressed before.

A major new contribution of this paper is the incorporation of resource contention into the QoS model . None of the previous approaches explicitly address the issue of contention for disks among various flows. For instance, if data is not replicated, several flows may simultaneously contend for overlapping subsets of the disks. The question of how to resolve the contention in an equitable manner that maximizes the system utilization, while still providing fairness (or differentiated service) to the contending flows has not been hitherto addressed. Most of the techniques suggested earlier either handle sharing of a single resource or of multiple uniform resources [17] in which any of the available servers (disks) could be used to satisfy a pending request.

There is a considerable body of existing work [3,4,8,9,11,12,27] on the problem of achieving differentiated services and weight-based proportionate fairness in Internet routers and switches. At a fundamental level, these algorithms use the notion of virtual-time tagging of requests to approximate the idealized Generalized Processor Sharing scheduling strategy to provide weighted QoS in datagram networks. In these models the resource is either a single server or multiple interchangeable (uniform) servers. For storage systems, Façade [21], Stonehenge [15], SFQ(D) [17] also propose virtual-time based scheduling strategies, while incorporating issues specific to storage workloads. These provide valuable system-level approaches and useful heuristics to achieve fair allocation in a uniform server model.

Our model for fair allocation is based on lexicographic minimization of the cumulative allocation vector that tracks the total resource allocation received by a flow. The resource usage may be measured in fixed size units like the number of I/Os performed, or by variable sized quantities such as the service time incurred at the disk.

The rest of the paper is organized as follows. In Section 2 we describe the model and attendant definitions. In Section 3 we present our algorithm, **LexAS** for scheduling with fairness and weighted-QoS for fixed size resources in the presence of resource constraints. Section 4 presents extensions of LexAS to situations where resources are variable-sized or not known a-priori, and discusses some policy issues. Section 5 presents empirical results of simulation experiments. We review related work in section 6 and conclude with section 7.

2. SYSTEM MODEL

The parallel I/O server consists of N independent disks, D_1, D_2, $\cdots$, D_N. The system contains *m flows*, f_1, f_2, $\cdots$ f_m, that seek *fair service* from the storage system[1]. A flow corresponds to requests from a client that has a contractual agreement on the share of the resources it should receive. I/O requests arrive from each flow and are held in *flow queues* until serviced. At a scheduling instant, the scheduler examines the requests in the flow queues and selects a subset of these requests to be dispatched to the disks. For every disk that has a pending request in any of the flow queues, one such request is dispatched by the scheduler. A scheduler with this property is said to be *work conserving*. The dispatched requests are buffered in *disk queues* associated with each disk. When a disk completes servicing its current request it chooses a request from its disk queue to serve next. To avoid idling a disk, the scheduler

[1]In Section 3.1 we generalize the model to handle differentiated service.

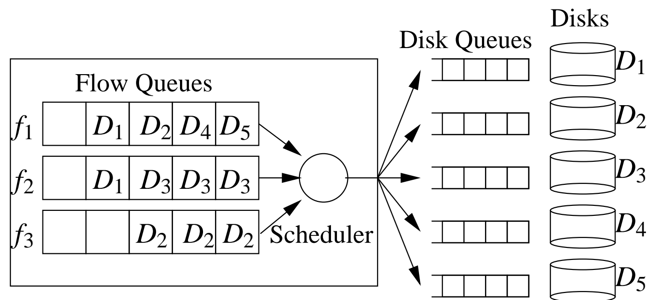

Figure 1: System model

must dispatch requests before any disk queue becomes empty, but otherwise there is considerable flexibility in the invocation of the scheduler. In an idealized model in which the disk service times are uniform for all disks, one can dispense with the disk queues and dispatch requests directly to the disks at each scheduling step.

Figure 1 shows an example of a system with 3 flows and 5 servers. Flow f_1 has requests for each of the the disks D_1, D_2, D_4 and D_5, while f_2 has one request for D_1 and three for D_3. Flow f_3 has three requests, all for disk D_2. The scheduler will chose a set of these requests containing at most one request from each disk, which will be dispatched in the current step. We use the following definitions in this paper.

- The **allocation vector** at scheduling step t, $A(t) = [b_1, b_2, \cdots, b_m]$, where b_j denotes the number of disks assigned to flow j at step t, and $\sum_{1 \leq i \leq m} b_i \leq N$.
- The **cumulative allocation vector** at I/O step t, $CA(t) = [B_1, B_2, \cdots, B_m]$, where B_j is the total number of I/O requests dispatched for flow j up to step t. That is, $CA(t) = \sum_{k=1}^{t} A(k)$.
- The **weight** of a vector is the sum of its components.

At any time a flow has a certain cumulative allocation equal to the number of its requests that have been dispatched. Given the current cumulative allocation and the requests in the flow queues, the scheduler will create a schedule for this step that (a) is work conserving and (b) allocates disks to flows so as to make the cumulative allocation at the end of this step as fair as possible. By fairness we mean as close to an even distribution of requests allocated to each flow. For instance, in the example of Figure 1, suppose that each flow has the same cumulative allocation prior to the start of this step. Since there are 5 disks with pending requests, a work conserving schedule will construct an allocation vector of weight 5. For instance, [4, 1, 0] is a feasible allocation where disks $\{D_1, D_2, D_4, D_5\}$ are allocated to f_1, and $\{D_3\}$ is allocated to f_2. Some other feasible allocation vectors in this case are: [3, 2, 0], [3, 1, 1], [2, 2, 1]. The fairest allocation in this case is [2, 2, 1], and the schedule will dispatch requests from $\{D_4, D_5\}$ for f_1, $\{D_1, D_3\}$ for f_2 and $\{D_2\}$ for f_3. On the other hand, suppose the initial cumulative vector was [10, 12, 12]; then the fairest allocation vector at this step would be [3, 1, 1] leading to a new cumulative vector of [13, 13, 13]. The algorithm which we present will compute the allocation vector that results in the fairest cumulative allocation vector after the current step. We formalize the concepts below.

DEFINITION 1. ***Lexicographic Ordering:*** *Consider two vectors $F = [f_1, f_2, \cdots f_n]$ and $G = [g_1, g_2, \cdots, g_n]$, with non-negative components, such that $\sum_i f_i = \sum_i g_i$, and that $f_i \geq f_{i+1}$ and $g_i \geq$*

g_{i+1}, *for all* $1 \le i \le n-1$ *(i.e. components are arranged in non-increasing order). Then F is lexicographically smaller than G if and only if either (i) F = G or (ii) there is an index* $k, k \ge 1$, *such that* $f_i = g_i, 1 \le i \le k-1$ *and* $f_k < g_k$.

For example consider the following 3-component vectors with weight 7: [3,0,4], [0,4,3], [2,3,2], [0,7,0], [0,5,2], [1,2,4]. When arranged in non increasing order of their component values, these are the distinct vectors [4, 3, 0], [3, 2, 2], [7,0,0], [5,2,0], [4,2,1]. The lexicographically smallest of these vectors is [3,2,2] corresponding to the most balanced distribution of the component values. Lexicographic minimality has been well studied in fairness literature along with maxmin fairness [24].

DEFINITION 2. ***Lexicographic Fairness:*** *Given* $CA(t-1)$ *and a set of requests, find a feasible allocation vector* $A(t)$ *of maximal weight such that* $CA(t) = CA(t-1) + A(t)$ *and for all feasible values of* $A(t)$, $CA(t)$ *is lexicographically minimum.*

3. SCHEDULING ALGORITHMS

In this section we will describe the scheduling algorithm **LexAS** (Lexicographic Allocation of Service) for fair lexicographic scheduling. Given the value of $CA(t-1)$, the scheduler finds a work conserving schedule such that $CA(t)$ is the lexicographically minimum vector. A straightforward algorithm is to try all assignments for $A(t)$ made up of ordered partitions of d into m components, where d is the number of disks with at least one pending request in the flow queues at this step. Each such assignment is checked for feasibility with respect to disk contention, and the feasible vector $A(t)$ that leads to the lexicographically minimum vector $CA(t)$ will be the desired allocation. However, the running time of such an algorithm would be exponential as there are $\theta(d^m)$ possible candidate vectors.

LexAS works in a number of iterations as follows. In each iteration it tries to assign an additional disk to the flow that currently has the smallest cumulative allocation. The aim is to allocate an additional disk to this flow without decreasing the number of disks allocated to any flow in previous iterations. In so doing, the actual set of disks allocated to flows in previous iterations may change, but the algorithm ensures that the number of disks allocated to the other flows does not change.

LexAS works by finding a set of paths in an *augmented* bipartite resource graph $G = (V \bigcup \{\alpha, \omega\}, E \bigcup E_\omega \bigcup E_\alpha)$ defined as follows:

- $V = \{f_1, f_2, \cdots, f_m\} \bigcup \{D_1, D_2, \cdots, D_N\}$ is a set of $m+N$ nodes, one for each flow and disk in the system.
- E is the set of directed edges between nodes representing flows and nodes representing disks: there is an edge (f_i, D_j) whenever there is a request for disk D_j in the queue for flow f_i.
- Distinguished vertices α and ω will serve as the source and sink of paths through the graph.
- $E_\omega = \{(D_j, \omega), 1 \le j \le N\}$, is the set of directed edges from each disk to ω.
- $E_\alpha \subseteq \{(\alpha, f_i), 1 \le i \le m\}$, is a subset of the directed edges from α to flow nodes f_i; this subset changes dynamically as the algorithm proceeds.

Computing the lexicographically fair schedule is mapped to finding a set of paths in a dynamically evolving resource graph beginning with G. Initially the resource graph consists of G defined above with E_α being empty. Our algorithm, formally stated in Algorithm 1 below, will maintain a *priority vector*, $\mathscr{P} = [p_1, p_2, \cdots, p_m] : p_i$ is the priority for flow node f_i. At each step the node f_i with highest-priority is selected, and an edge from α to f_i is added to G. The algorithm then attempts to find a path in the current resource graph from α to ω.

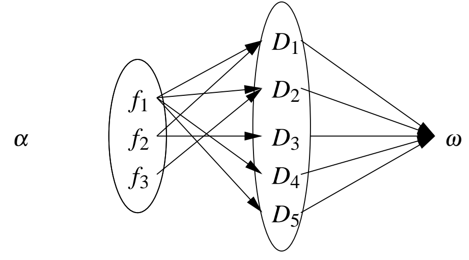

Figure 2: Augmented bi-partite graph

If a path cannot be found, then the currently selected node f_i is marked as *saturated*, and the algorithm will adjust the priority of f_i so that it will not be selected again. A saturated node means that it is not possible to increase the total number of disks allocated to this flow, without reducing the number of disks allocated to one of the other previously scheduled flows. This type of reallocation is undesirable for fairness of the schedule. If the search for a path is successful this means that one additional request made by flow f_i can be satisfied, without decreasing the number of disks already allocated to other flows by the algorithm. The actual assignment of disks to flows might get changed to make this possible, but the *number of disks* assigned to any other flow will *not change* by this reassignment. Thus the total number of disks allocated increases without disturbing the relative allocations needed for fairness. In preparation for the next iteration, the resource graph is modified to reflect the new assignments as described below. This step will be referred to as *path conditioning* in the description of the algorithm.

Path Conditioning: In the current resource graph every edge in *the newly identified path* is reversed except the first edge starting from α (that edge is removed). Formally, every edge (f_a, D_b) is replaced by its reverse edge (D_b, f_a), and similarly every edge (D_r, f_t) in the path is replaced by the reverse edge (f_t, D_r). The last edge on the discovered path from some D_u to ω is also replaced by the reverse edge (ω, D_u). The first edge (α, f_i) is removed from the graph. No changes are made to edges that do not belong to the found path.

The algorithm maintains the following **invariant**: an edge (D_r, f_t) in the resource graph at the start of an iteration means that currently disk D_r is assigned to flow f_t. Suppose the algorithm chooses to allocate a disk to flow f_u at some iteration, and finds a path ($\alpha, f_u, D_{j_1}, f_{j_1}, D_{j_2}, f_{j_2}, \cdots, D_{j_k}, f_{j_k}, D_{j_{k+1}}, \omega$) through f_u. This implies that at the start of the iteration there were at least k assignments: disk D_{j_i} was assigned to flow $f_{j_i}, 1 \le i \le k$. When the edges between flows and disk nodes are reversed by the path conditioning step at the end of the iteration, the $k+1$ new assignments will be: D_{j_1} to f_u, D_{j_2} to f_{j_1}, and so on ending with $D_{j_{k+1}}$ assigned to f_{j_k}. Note that for each of the flow vertices f_{j_i} only the identity of the assigned disk was changed.

Figure 2 shows the set of vertices V and edges E for the resource graph constructed for the example of Figure 1. The graph consists of 3 flow vertices and 5 disk vertices; there is an edge (f_i, D_j) if there is a request for disk D_j in the queue of flow f_i. Assume for purpose of exposition that the $CA(t-1)$ = [0,2,2] at this

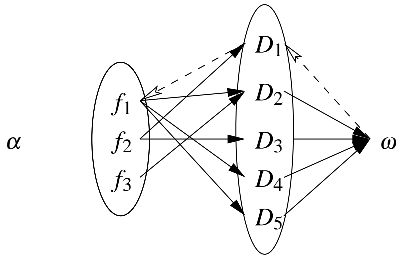

Figure 3: G_1, graph after one iteration

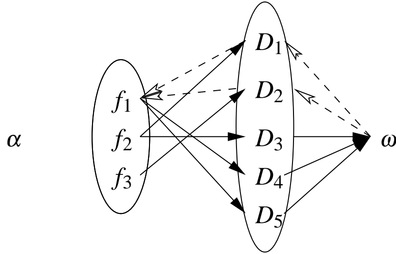

Figure 4: G_2, graph after 2 iterations

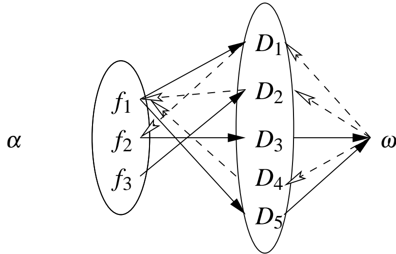

Figure 5: G_3, graph after 3 iterations

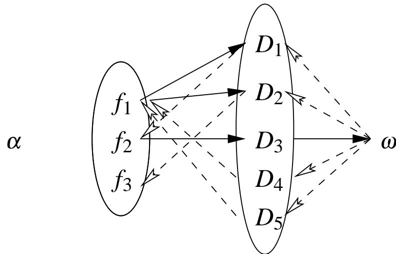

Figure 6: G_4, graph after 4 iterations

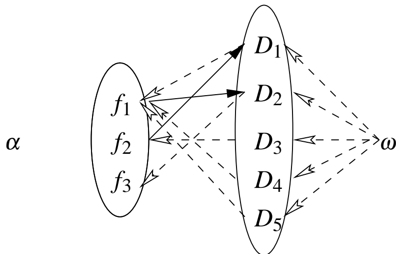

Figure 7: G_5, graph after 5 iterations

time. LexAS will first try to assign a disk to flow f_1, since it is the smallest component. Suppose it finds the path $(\alpha, f_1, D_1, \omega)$ through f_1. Following the path conditioning step, the modified graph G_1 is shown in Figure 3; the edge (D_1, f_1) indicates that currently D_1 is assigned to flow f_1. The current allocation vector $A(t)$ = [1, 0, 0]. The minimum component of $CA(t-1)+A(t)$ = [1, 2, 2] is still f_1, and the algorithm looks for another path through f_1 in graph G_1. If the path $(\alpha, f_1, D_2, \omega)$ is found, then, following the path conditioning step, we have the modified graph G_2 shown in Figure 4. Note that both D_1 and D_2 are assigned to f_1 and $CA(t-1)+A(t)$ = [2, 2, 2]. All f_1, f_2 and f_3 are candidates for the next selection, and we assume that f_2 is chosen arbitrarily. Suppose that the path through f_2 in G_2 that is found by LexAS is $(\alpha, f_2, D_1, f_1, D_4, \omega)$. Following path conditioning the graph G_3 is shown in Figure 5. Note that D_1 has been reassigned from f_1 to flow f_2, and that D_4 has been assigned to f_1. The number of disks assigned to f_1 is still 2, while f_2 now has a disk assigned to it. The vector $CA(t-1)+A(t)$ = [2, 3, 2]. For the next iteration, assume that f_3 is selected. The path $(\alpha, f_3, D_2, f_1, D_5, \omega)$ after path conditioning leads to the graph shown in Figure 6. Note that f_1 has two disks $\{D_4, D_5\}$ assigned, f_2 has the assignment $\{D_1\}$ and f_3 has the assignment $\{D_2\}$. $CA(t-1)+A(t)$ = [2, 3, 3], so the final flow to be selected is f_1. The path $(\alpha, f_1, D_1, f_2, D_3, \omega)$ following path conditioning is shown in Figure 7. The final allocation vector $A(t)$ = [3, 1, 1], leading to $CA(t)$ = [0, 2, 2] + [3, 1, 1] = [3, 3, 3]. The final assignment is disks $\{D_1, D_4, D_5\}$ to f_1, disk $\{D_3\}$ to f_2 and disk $\{D_2\}$ to f_3.

Note that although related to the problems of bipartite graph matching and determining maximum flow, there are subtle but significant differences that preclude direct application of either of these algorithms to our problem [13]. Our algorithm is inspired by the ideas of augmenting paths and residual graphs employed in the Ford Fulkerson algorithm for maximum flow in a network, but has been suitably refined for the problem at hand.

```
1  allocations = 0;
2  d = number of disks with at least one request pending;
3  G is augmented resource graph with E_α = φ;
4  Let priority vector 𝒫 = [p_1, p_2, ··· , p_m], where p_i = B_i, at
   the time of scheduling;
5  Mark all p_i's as non-saturated;
6  while (allocations < d) do
7      Choose the minimum non-saturated element p_i of 𝒫
       with ties broken arbitrarily. Add edge (α, f_i) to G;
8      Search for a path from α to ω that includes (α, f_i) ;
9      if (no path is found) then
10         Mark p_i as saturated in 𝒫;
11     else
12         p_i = p_i + 1;
13         allocations = allocations + 1;
14     end
15     Update graph G by path conditioning;
16 end
```

Algorithm 1: LexAS algorithm

The following lemma formally asserts that the choice of flow node at each iteration of LexAS leads to a lexicographically minimal cumulative allocation vector.

LEMMA 1. *Given vector* $X = [x_1, x_2, \cdots x_i, \cdots x_m]$*, such that* $x_1 \geq x_2 \cdots \geq x_m$*. Let* $Y_i = [x_1, x_2, \cdots x_i + 1, \cdots x_m]$*, and let* $\hat{Y}_i$ *denote the vector obtained by arranging the components of* Y_i *in non-increasing order. Then* $\hat{Y}_m$ *is lexicographically minimum of all the vectors* $\hat{Y}_i, i = 1, \cdots, m$*.*

LEMMA 2. *Once LexAS allocates a disk node to some flow node, the disk will never become free (although it may be reassigned to a different flow node).*

LEMMA 3. *LexAS finds the minimum lexicographical allocation of d disk blocks to flows.*

LEMMA 4. *The running time of LexAS algorithm is $O((m+d)|E|)$.*

Lemma 1 is proved later on as a special case of Lemma 5. The proofs of Lemmas 2, 3 and 4 are not presented due to lack of space. The proofs are available in [13].

3.1 Differentiated Service

In this section we describe how LexAS can be generalized to handle the problem of providing *differentiated service* to the set of flows. In this model, each flow f_i has an associated *weight* w_i, $0 \le w_i \le 1$ and $\sum_{i=1}^{m} w_i = 1$. Flow f_i should be allocated a fraction w_i of the total number of I/O requests dispatched at any time.

Let $CA(t) = [B_1, B_2, \cdots B_m]$ be the cumulative allocation vector at time step t, and let its weight be denoted by W. Then under differentiated service, flow f_i should do a fraction w_i of the total number of I/Os performed: that is $B_i = W w_i$, $i = 1, \cdots, m$. It follows, that under differentiated service: $B_1/w_1 = B_2/w_2 = \cdots = B_i/w_i = \cdots = B_m/w_m$. We therefore also refer to differentiated service as *proportional service*.

Define a **normalized cumulative allocation vector** $\mathcal{N}(t)$, consisting of weight-scaled values of the cumulative allocation vector, $CA(t) = [B_1, B_2, \cdots, B_m]$. That is, $\mathcal{N}(t) = [\eta_1, \eta_2, \cdots \eta_m]$, where $\eta_i = B_i/w_i, i = 1, \cdots, m$.

We modify algorithm LexAS for the case of differentiated service as follows. The *priority vector* $\mathscr{P}$ in step 4 of the algorithm is initialized to $\mathcal{N}(t)$ at the start. Step 7 that chooses the flow to be allocated a disk is modified as follows: among all non-saturated flow nodes, we select the one with the minimum value of $\eta_i + (1/w_i)$. Finally in step 12, we increment p_i by $1/w_i$. For completeness the algorithm with these changes is presented as Algorithm 2.

To show that the choice of flow node in step 7 leads to a lexicographically minimal normalized cumulative allocation vector, we argue as follows.

LEMMA 5. *Given vector $X = [x_1, x_2, \cdots x_i, \cdots x_m]$, such that $x_1 \ge x_2 \cdots \ge x_m$. Let $Y_i = [x_1, x_2, \cdots x_i + 1/w_i, \cdots x_m]$ and let $\hat{Y}_i$ denote the vector obtained by arranging the components of Y_i in non-increasing order. Let t be the index such that $x_t + 1/w_t \le x_k + 1/w_k$, for all k, $1 \le k \le m$. Then $\hat{Y}_t$ is lexicographically minimum of all the vectors $\hat{Y}_k, k = 1, \cdots, m$.*

PROOF. (Sketch) Let $u_t = x_t + 1/w_t$ and $u_k = x_k + 1/w_k$. Since $u_k \ge u_t > x_t$, the rank[2] of u_k in $\hat{Y}_k$, is less than t. Similarly, since $u_t > x_t$, the rank of u_t in $\hat{Y}_t$ is no more than t. Consider the elements with the first t ranks of vector $\hat{Y}_t$ and of vector $\hat{Y}_k$. They differ in at most two elements: u_k and x_t in $\hat{Y}_k$ and u_t and x_k in $\hat{Y}_t$. Since $u_t \le u_k$, it follows by looking at the first t ranks that $\hat{Y}_t$ is lexicographically smaller than $\hat{Y}_k$. □

In the special case where all weights are equal, the lemma simplifies to Lemma 1.

[2]The rank of a component is equal to the number of elements in the vector that are no smaller than it.

```
1  allocations = 0;
2  d = number of disks with at least one request pending;
3  G is augmented resource graph with E_α = φ;
4  Let priority vector 𝒫 = [p_1, p_2, ..., p_m], where p_i = η_i, at
   the time of scheduling;
5  Mark all p_i's as non-saturated;
6  while (allocations < d) do
7      Choose the non-saturated element p_i of 𝒫 with the
       minimum value of p_i + 1/w_i. Break ties arbitrarily.
       Add edge (α, f_i) to G;
8      Search for a path from α to ω that includes (α, f_i) ;
9      if (no path is found) then
10         Mark p_i as saturated in 𝒫;
11     else
12         p_i = p_i + 1/w_i;
13         allocations = allocations + 1;
14     end
15     Update graph G by path conditioning;
16 end
```

Algorithm 2: LexAS algorithm for weighted allocation

4. MODEL EXTENSIONS

In this section we consider extensions to the model of the previous section to deal with some practical issues and generalizations.

4.1 Variable sized Resources

The first extension will deal with allocating variable-sized resources. For specificity, we will consider service time as the resource since different requests will in general, incur varying amounts of service times at a disk. We show below that the problem of finding a fair allocation in this case is NP-Complete. For this situation we generalize the definition of the allocation vector $A(t) = [b_1, b_2, \cdots, b_m]$ in the natural way, so that b_j is the sum of service times of the dispatched requests (instead of the number of dispatched requests) from flow j at step t.

DEFINITION 3. *Fairness with Variable Service Times (FVST) problem: The input is a set of requests $\{r_{i,j} : 1 \le i \le m, 1 \le j \le N\}$; $r_{i,j}$ is a request from flow f_i for a resource R_j, with associated service time $c_{i,j}$. Find a work-conserving schedule with the lexicographically minimum allocation vector $A(t)$. The decision problem asks whether there exists a work-conserving schedule in which all components of $A(t)$ are equal?*

LEMMA 6. *FVST is NP-Complete.*

PROOF. By reduction from *2-Partition*. Consider an instance of 2-partition with input set $\mathscr{A} = \{a_1, a_2, \cdots, a_n\}$. Create an instance of FVST with $m = 2$ flows and $N = n$ resources; create a set of $2n$ requests $\{r_{i,j}, 1 \le i \le 2, 1 \le j \le n\}$, and $c_{i,j} = a_j$ for all $1 \le i \le 2$. It is easy to see that there exists a 2-partition of $\mathscr{A}$ if and only if there exists a work-conserving schedule in which both components of the allocation vector are equal to $1/2 \sum_{i=1}^{n} a_i$. □

In addition to the issue of computational complexity raised above, it is notoriously difficult in practice to make accurate a-priori estimates of the resource requirements such as the disk service time of a request [26]. Hence a practical scheme will have to work around this limitation in some way if the goal is to assign cumulative service time in a proportional manner. We propose using a feedback-based adaptive scheme in conjunction with LexAS as described below.

The cumulative allocation vector $CA(t)$ is maintained by empirical monitoring and feedback from the resource controller to the scheduler. The actual service time of each request is monitored at the resource and, after the request completes, is used to update the accumulated service time of the flow to which the completed request belongs. The system also maintains the average service time of the requests, which will be used as an estimate for the service time of the new requests by LexAS. When LexAS is invoked at time t, it uses the measured values of $CA(t)$ to initialize the priority vector $\mathscr{P}$ used in the scheduling algorithm. The average value of the service time c^* is assumed for each request during the scheduling. Hence after every allocation of a resource, the appropriate component of $\mathscr{P}$ is increased by c^* or in the case of differentiated service by c^*/w_i.

4.2 Minimizing Schedule Length

We next consider the issue of the length of the overall schedule. To formulate the problem precisely, we consider the static (off-line) situation where the entire set of requests in the queues of all the flows is scheduled before the next batch of requests is admitted. In the model as described previously, the scheduler is free to select any of the pending requests from any of the flows; the accessed block is immediately returned to the application driving that flow which may internally buffer the block if desired. In this model, not only is LexAS a fair algorithm, it also *maximizes the I/O throughput*, due to its work-conserving property.

An alternative model seeks to restrict the order in which requests from a flow can be serviced. In this model the requests from a flow are consumed sequentially, and there is a maximum buffer of size M blocks available for any flow. Thus, for each flow at most M blocks including the earliest unfetched block of that flow, are candidates for I/O at any step. Being work-conserving is no longer sufficient to guarantee that the schedule has minimal length. Depending on which flow was allocated a disk at a time step, the set of new candidates for scheduling at the next time step changes. In this situation, the problem of constructing an I/O schedule that *minimizes the number of I/O steps* to service all requests has been shown to be NP-complete [13] even when $M = 1$ and all service times are equal. We omit the details except to mention that a suitable reduction from *3-Partition* is used to prove NP-completeness.

Both types of workloads occur in practice. The first models transactional workloads consisting of independent disk requests. In this case LexAS optimizes both the system throughput as well as providing fair (or proportionate) allocation to individual flows. The latter situation models streaming workloads where there is a fixed order in which blocks are required, and the buffer size restricts arbitrarily deep prefetches. In this case LexAS optimizes on fairness, giving each flow proportional service. Of course, this may result in a schedule that is longer than the minimum length schedule. However note that finding the minimum length schedule is computationally intractable, and is not guaranteed to be fair.

4.3 Server Allocation Policies

We now discuss two models for proportionate resource allocation, which will be referred to as *Fair Window Scheduling* (abbreviated FWS) and *Don't Use And Lose* (abbreviated DUaL) respectively. The models differ with respect to the time interval over which the allocation is measured, and are used to capture the notion of average or instantaneous quality-of-service respectively.

4.3.1 FWS policy

Under the FWS policy, the scheduler attempts to provide flows with fair or proportionate allocation of server resources over a long run of the system. At any time t, the FWS scheduler tries to ensure that the cumulative accumulation $CA(t)$ from the start of the window at time 0, is in proportion to the weight of a flow. LexAS will provide a flow with the smaller of its total service demand and its proportional share.

Such a model is useful in situations involving the concurrent execution of multiple batched applications, such as accessing multiple variable-bit rate multimedia streams, or in scientific codes that alternate between I/O-intensive and compute-bound phases of activity. Such applications may go through periods where the demands on the I/O system are low, followed by periods of high I/O activity. In FWS scheduling, a flow implicitly gains service credits during the periods in which it makes less than its fair share of the demand on the server. When the flow switches to a phase of heavy demand, the scheduler redeems those accumulated credits and temporarily provides it more than its proportionate share of the server resources, in order to increase the average allocation to its proportional share.

4.3.2 DUaL policy

The alternative DUaL policy is based on the notion of "use it or lose it". Under this policy it is considered unfair for a flow to receive service credits for failing to fully utilize its share of the resources. This is motivated by the potentially severe short-term impact on other flows that can occur when a flow recovers accumulated credits. Such a flow may monopolize the servers until its service allocation approaches its proportional share, starving other flows during this interval. The DUaL policy attempts to provide proportionate service to flows f and g over every window when both flows are backlogged.

Fair scheduling schemes [11, 12, 17] are explicitly designed to enforce this policy. Although both models have been considered in the literature and it has been shown that both the policies can be implemented [27] in practice , the latter has been preferred for scheduling network traffic due to the nature of the workload. Clearly no work conserving schedule can guarantee proportionate resource allocation both in every backlogged time window $[s,t]$ as well as in the time window $[0,t]$ (unless flows are never idle). For instance, a flow which has been idle from time 0 to s will receive very different allocations in a window $[s,t]$ under the two policies. This behavior is illustrated in Figures 8 and 9 in section 5. The appropriate policy should be decided based on an understanding of the workload and performance objectives.

The LexAS algorithm facilitates the use of a simple mechanism that can be tuned to either of these policies. By appropriate setting of a parameter value, LexAS can either give no credit at all to a flow during its under utilized periods as in the DUaL policy, or give complete credit to a flow for its under utilized period as in the FWS policy, or any amount in between. The description earlier was for the FWS policy, since no allowance was made for an idle flow. For the DUaL policy the only change is in the handling of flows that have been idle for some time. The scheduler will artificially increase the cumulative allocation of an idle flow, so that it cannot get credit for its idle period. The algorithm needs to keep track of when an idle flow (say f_r) is reactivated, and increment its cumulative allocation B_r to $(B_{min}/w_{min}) \times w_r$, where B_{min} is the smallest cumulative accumulation of any currently backlogged flow. Essentially, a newly awakened flow restarts at an equal footing with other flows in competing for service at this time. Recall, that the DUaL policy does not permit a flow to gain credit during its idle periods.

In the DuAL policy in a window where two flows f and g are both backlogged the request sequences can be so skewed that one of the flows gets an arbitrarily larger amount of service than the other. For instance one flow may have requests only to one disk

(lets say disk 1), while the other flow may have requests for all disks. This means that $|B_f(t) - B_g(t)|$ can grow without bound. On the other hand, the ratio $B_f(t)/B_g(t)$ can be shown to be bounded, since LexAS will ensure that the skewed flow is not starved of service on disk 1. These observations are summarized in the Lemmas below. The proofs are omitted due to lack of space.

THEOREM 1. *For any work conserving schedule, there exist request sequences for which the difference $|B_f(t) - B_g(t)|$ grows with t, even though f and g are continuously backlogged in the interval $[0,t]$.*

THEOREM 2. *Let c_{max} be the maximum service time of a request. In any sufficiently long time interval T in which flows f and g are backlogged during the entire interval we have:*

$$\frac{\eta_f(T)}{\eta_g(T)} \leq N \left(\frac{w_g}{w_f\,(w_g - c_{max}/T)} \right)$$

5. SIMULATION-BASED EVALUATION

In this section we present the results of evaluating LexAS using simulation on synthetically generated workloads. The workloads are synthesized from *random*, *restricted random* and *skewed* access patterns as follows. In the first case, every request of a flow is to a disk chosen randomly with equal probability. In case of *restricted random*, a flow randomly accesses a subset of disks: restriction parameter ϕ represents the fraction of disks accessed by the flow. We use $\phi = 0.5$ in our experiments. In the skewed model successive accesses are correlated. In this model, the ith request belongs to the same disk as the $(i-1)^{th}$ request with some probability p; p starts with a certain maximum value (close to 1) and keeps on decreasing linearly. This produces skewed access patterns for flows, where there is a burst of requests to a disk, followed by a burst on another disk, and so on. This behavior captures locality in disk requests. We also used *striped skew* pattern where the next access is for the next numbered disk (with wraparound) with high probability.

In the simulation, we show two basic characteristics of LexAS. First is the ability to *isolate the performance* of different flows in achieving the desired *weighted allocation* of resources. This isolation prevents a badly-behaving flow from negatively impacting the behavior of other flows. Secondly, we show that LexAS works well in practical situations in which the service times are non-uniform. We also compare LexAS with some common schemes such as Random and Round-robin to better illustrate the challenges in QoS scheduling.

5.1 Uniform Service times

The first set of simulation experiments assumed uniform service times for the requests, and scheduled successive I/Os using the basic LexAS algorithm described in Figure 1. Each flow queue was filled with 64 requests at the scheduling instant. For these experiments we chose 64 disks, and we experimented with 5 flows. 3 of the flows were generated using the random model and the other 2 consisted of interleaved sections of random and skewed requests generated as follows: if L is the total number of requests in the flow, requests in the segment $[0, L/5]$ are skewed, $[L/5, 2L/3]$ is generated randomly, $[2L/3, 4L/5]$ is generated again using the skewed model, and $[4L/5, L]$ is generated randomly. We chose these segment lengths to be able to highlight the difference in behavior of DUaL and FWS policies.

We ran LexAS (DUaL) algorithm on these 5 flows and the result is shown in Figure 8. The weights of all flows were assumed to be equal, so we were looking for a fair allocation. Note that we plot

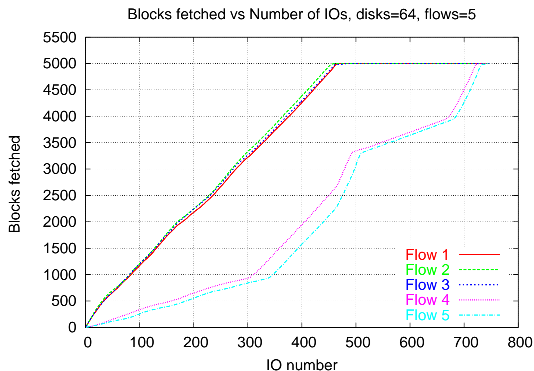

Figure 8: Fair allocation by LexAS(DUaL), skewed inputs

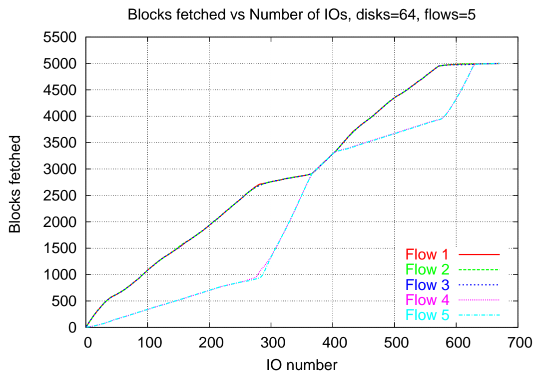

Figure 9: Fair allocation by LexAS(FWS), skewed inputs

cumulative number of blocks accessed (B_i) for each flow on the y-axis and the I/O step number on the x-axis. Thus the slope of the curve indicates the number of blocks fetched per I/O for that flow. We observe that till about 320 I/Os, flows 4 and 5 get fewer blocks per I/O because their accesses in this range are skewed. When these flows are in their random phase (between I/O numbers 320 and 480), all flows get a similar number of blocks per IO as expected in the DUaL policy. Note the slopes of all five flows are equal in this range. After step 480, the slope for flows 4 and 5 increases slightly because flows 1, 2 and 3 have completed. In the interval from 500 to 680 IO steps, the second skewed region for flows 4 and 5 is reached, as shown by their decreased slope. The rest of the region is generated with the random model and is scheduled with higher slope than the earlier random section, since only two flows are contending for resources.

Figure 9 shows the results obtained by running LexAS (FWS) on the same input. There are 3 main points to note here: (*i*) the initial skewed segment in flows 4 and 5 completes in about 280 IOs rather than 320 in case of DUaL (*ii*) between I/O numbers 280 and 360, the slope of curves 4 and 5 is much higher than the slope of curves 1, 2 and 3 because FWS favors the flows lagging behind in their fair share of number of blocks, and (*iii*) around step number 360 all flows have an equal number of blocks and their slope is equal from that point till the flows 4 and 5 enter their second skewed phase.

To test the LexAS algorithm for a weighted allocation, we experimented with 4 flows where flows 1 and 2 are generated randomly, and flows 3 and 4 have an initial 5% region generated in a skewed manner; at a random later point again a certain number of

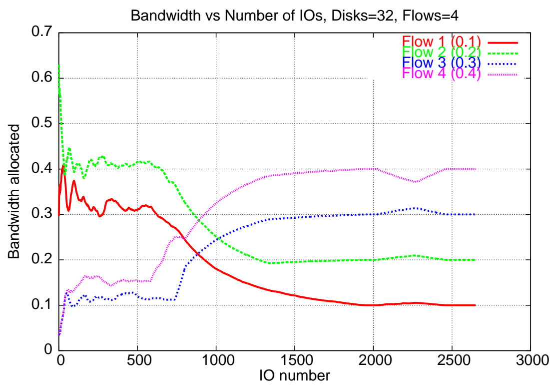

Figure 10: Weighted allocation by LexAS(FWS), skewed inputs

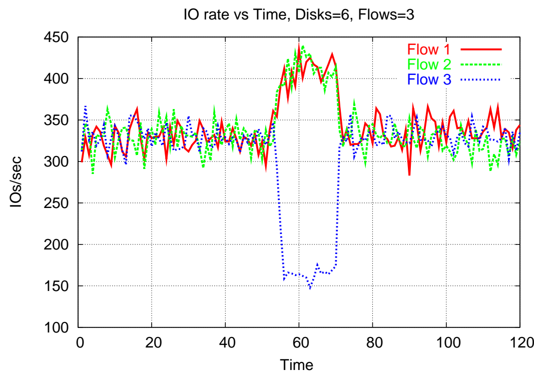

Figure 11: Fair allocation by Random Scheme

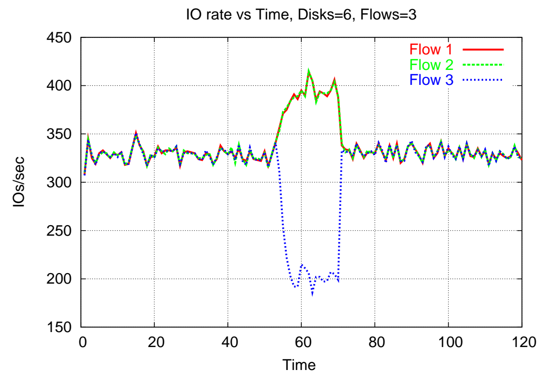

Figure 12: Fair allocation by Round-robin Scheme

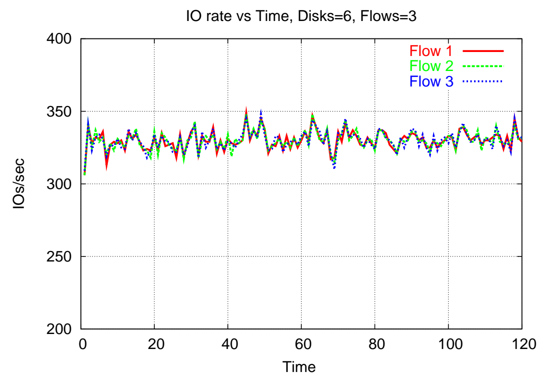

Figure 13: Fair allocation by LexAS (FWS)

requests are skewed (to measure the sensitivity of the algorithm). The weights for the flows were chosen to be 0.1, 0.2, 0.3, 0.4 respectively. We observe in Figure 10 that initially flows 3 and 4 (the two lower curves) get less than their weighted share due to the skewness (note that these two flows have higher weight), and flows 1 and 2 get more than their fair share, since the work conserving property allocates the share unused by flows 3 and 4 to the other flows. Also notice that flow 2 gets a larger share of the free capacity than 1 due to its higher weight, and 4 gets a larger share than 3. As soon the hot spot activity passes, flows approach their proportionate share of bandwidth. Later on near step 2000, there is again a short hot spot activity in flow 4 that is shown by the slight fluctuation in the curves.

5.2 Non-Uniform Service Times

In this section we assume non-uniform service times uniformly distributed between t_{min} and t_{max}, and Poisson arrivals. We chose $t_{min} = 0.013$ msec and $t_{max} = 12.11$ msec.[3]. LexAS is invoked whenever a disk becomes free. The length of the disk queues was fixed at 4: hence LexAS ignores disks that currently have 5 outstanding requests at that disk, when constructing its dispatch schedule. Note that with a queue size of zero, LexAS simply dispatches a request from the flow with a request for the free disk with minimum cumulative allocation. Façade [21] has shown that throughput increases with increased disk queue length, because requests can be re-ordered to support lower-level optimizations. Thus zero queue length would provide poorer throughput in practice as there would be no option for the low-level scheduler to optimize for seek and rotational delays.

First we compared our approach to some simpler schemes such as **Random** and **Round-robin**. Upon completion of a request, **Random** immediately assigns the free disk to a randomly-chosen flow with a request for that disk. **Round-robin** assigns the free disk to the next flow with a request for the disk in a round-robin traversal of the flows. We used three randomly generated flows with equal weights, with average arrival rates of 600 IOs/sec each. Flow 3 accesses only half ($\phi = 0.5$) the disks in a random manner from 30 to 70 seconds. This restricted behavior is introduced to check the capability of Random and Round-robin schemes. Figures 11 and 12 shows the achieved IO rate (IOs/sec) for each of the flows for Random and Round-robin schemes respectively. Observe that the share of flow 3 decreases during the skewed portion and the other two flows get a larger share of the system bandwidth. Figure 13 shows the behavior of LexAS which achieves fairness even in presence of skew. If requests are randomly distributed to all the disks for all flows, then Random and Round-robin also achieve fairness to a large extent. We experimented with various values of t_{min}, t_{max} and ϕ and observed similar results. Lower values of ϕ increases unfairness in distribution and higher values make the other schemes closer to a fair distribution. This happens because Random and Round-robin schemes allocate fair share only among the disks accessed by a flow, and they do not compensate for lack of accesses to some disks.

We tested LexAS for weighted distribution, skewed behavior and mixed arrival patterns. The storage system consists of 8 disks and has a capacity of approximately 1400 IOs/sec. This is deliberately

[3]These values are chosen based on the description of disk characteristics given in chapter seven of [14].

set less than the total input rate (2100 IOs/s) to see the weighted distribution; otherwise all the flows will get service rates equal to their input rates due to the work conserving property of LexAS. First we experimented with three flows having weights 0.2, 0.3 and 0.5 respectively. In this experiment flow 1 accesses all disks in a random order, flow 2 has skewed accesses to all disks from 0 to 10 sec and again from 60 to 70 sec, flow 3 randomly accesses half the disks from 30 to 70 sec and all the disks during the rest of the time. Figure 14 shows the achieved IO rate (IOs/sec) for the flows, with continuous average IO rates of 700 IOs/sec each. We observe that even with variable service times and input skews, LexAS allocates IOs/sec proportional to the flow weights.

We also experimented with striped skew and on-off arrivals. Figure 15 shows the distribution of IOs/sec for 4 flows which are generated as follows: Flow 1 and 3 are generated in a striped skew manner, flow 2 has restricted random accesses from 20 to 60 sec and random otherwise, flow 4 has a periodic on-off arrival pattern with 40 sec on-time followed by 10 sec off-time. Flows 1, 2 and 3 have arrival rates of 600 IOs/sec and flow 4 has an arrival rate of 500 IOs/sec. Figure 15 shows that all flows get IOs proportional to their weights. Note that during the off-time of flow 4, spare capacity is also distributed in a weighted manner. There is a small drop in IOs given to flow 1 from 50 to 60 sec because flow 1 received more than its share during the off-time due to lack of requests from other flows and FWS scheduling policy keeps track of previous history for future allocations. Flow 4 did not receive any extra share once it came back (at time = 50 sec), because its arrival rate is almost equal to its service rate and there are no pending requests in its queue. Also at 90 sec, flow 4 gets idle again and spare capacity is shared in a weighted mannner by the remaining flows.

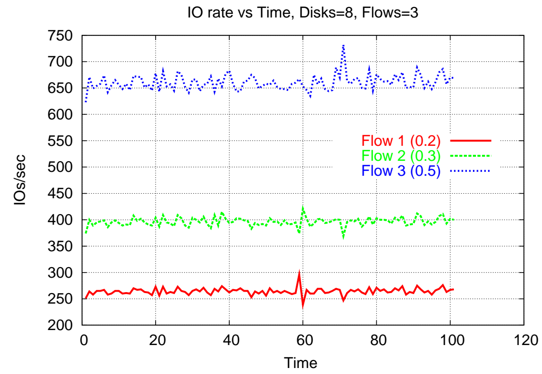

Figure 14: Weighted allocation by LexAS(FWS)

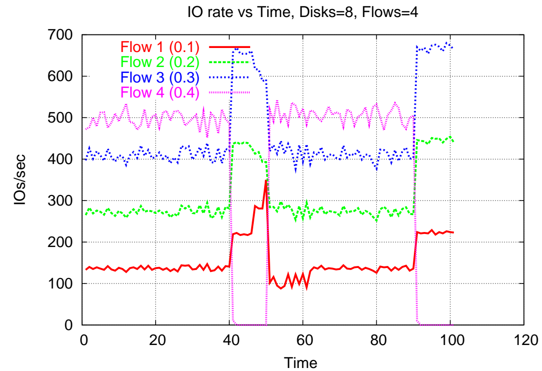

Figure 15: Weighted allocation by LexAS(FWS)

6. RELATED WORK

The work related to LexAS can be divided into three major categories: (i) virtualization systems for shared storage such as Interposed Proportional sharing [17], Façade [21], and Stonehenge [15], (ii) QoS and fairness in network scheduling and (iii) Fair allocation of a single disk.

Façade [21] provides a virtualization approach to fulfill *Service Level Objectives* (SLOs) of independent workloads accessing a storage system by doing EDF (earliest deadline first) scheduling. Although EDF is work conserving, it may not be able to provide performance isolation in case of sudden rise in demand from certain flows. Façade assumes the availability of a capacity planner for admission control, so that all the SLOs are satisfied (unfairness of EDF is shown in [17]). Our approach is work conserving and provides a finer control over allocation of disks to workloads in each I/O, to achieve the desired bandwidth allocation. Interposed proportional sharing algorithms are suggested in [17] that assign virtual start and finish times to requests based on their cost, weight of the client queue and arrival time. They show their algorithm to be fair as well as work conserving. However, the model in [17] assumes a single server or multiple uniform (interchangeable) servers. In contrast, we attain fairness even when flows can request service from specific servers and the set of requested servers can vary dynamically.

Stonehenge [15] proposes a *virtual clock* based two-level disk scheduling architecture. At the first level every request goes to a central controller that determines a deadline for the request, and at the second level a CVC disk scheduler is used at each disk. The drawback of Stonehenge is that it does not explicitly address disk contention. In addition a virtual stream can get an unfair share of bandwidth if it gets idle for some time and comes back. This is because the *virtual time* of the stream will lag behind as compared to active flows. We believe that our scheme can be used at the central scheduler in a system like Stonehenge while taking into consideration the contention among physical disks for different virtual streams.

Many schemes have been developed for fair scheduling both in datagram and packet switched networks [3,4,9,11,12,27]. Most of them are variants of the Generalized Processor Sharing (GPS) algorithm and differ based on fairness guarantees, delay bounds and cumulative service provided to flows [12]. Some of them([3,4,8]) try to approximate GPS very closely and do elaborate calculations that are prohibitive to implement in high speed networks. Others [9,11] try to find fast approximations thereby loosing in terms of bounds they can provide for delay and cumulative service during any interval. Our approach when applied to their scenario matches the lowest bound achieved by them for cumulative bandwidth obtained by different flows. These algorithms are not applicable to our scenario where multiple resources are shared by flows in the presence of resource constraints on each request. In case of Internet switches, schemes such as PIM [2], iSLIP [22], shakeup [10] match input-output ports in a probabilistic, round-robin or random manner to achieve high throughput and fairness. iFS [23] assigns virtual start and finish times to each packet and matching is done based on bandwidth requirements of the flow. These schemes obtain a one-to-one mapping at each step, whereas we need one-to-many at each IO step. Iteratively applying bipartite matching suggested by them prevents reassignment in the future, so fairness is no longer guaranteed. Other mechanisms such as QoSBox [7], and Diffservs [5] achieve service differentiation by packet dropping, traffic shaping and reordering. Most of these techniques cannot be adapted to work in storage systems [26].

Several disk scheduling policies have been proposed for fairness in case of single disks. Silberschatz et al. introduced YFQ [6] disk scheduling algorithm that assigns a *start and finish time* to each request so as to approximate generalized processor sharing (GPS) algorithm for resource sharing. YFQ schedules requests in order of finish times, and also performs various optimizations to reach the efficiency of seek/rotational delay based algorithms. Cello [25] suggests a two-level scheduling framework that includes a coarse-grained scheduler for each application that works according to the application needs and a fine-grained common scheduler that re-orders requests from different schedulers to optimize for seek and rotational delays. Our approach however, provides weighted fairness across multiple disks. Furthermore, multiple requests can be sent to each disk to get the benefits of the lower level optimizations.

7. CONCLUSIONS

In this paper we proposed an algorithm for QoS scheduling for parallel disks. Our approach can be used to support an economic model deployed at data centers where clients pay according to their service agreements with the SSP (storage service provider), and different flows are isolated from each other. The results show that LexAS provides weighted-fairness to different flows and is work conserving. A benefit of our approach is that it works well even when the service times of requests are not known a priori, by using empirically measured feedback to drive the algorithm. In the absence of such mechanisms, the algorithm can still work using estimates of the service time.

LexAS is the first approach that handles allocation of shared resources in the presence of explicit resource constraints. Previous approaches to QoS scheduling have concentrated on single resources or multiple uniform (indistinguishable) resources. Our results have applicability in a broader context of multiple heterogeneous servers, where clients may require specific servers at different times.

Acknowledgments

Support for the first author was provided by the National Science Foundation under Grant CCR-0105565. Research time and travel support during IPA assignment of the second author was provided by the National Science Foundation under the IR/D program. The support is gratefully acknowledged.

We also thank the anonymous reviewers whose pertinent comments have resulted in many improvements.

8. REFERENCES

[1] S. Albers, N. Garg, and S. Leonardi. Minimizing stall time in single and parallel disk systems. *J. ACM*, 47(6):969–986, 2000.

[2] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High speed switch scheduling for local area networks. *ACM Transactions On Computer Systems*, 11:319–352, November 1993.

[3] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM (1)*, pages 120–128, 1996.

[4] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. IETF RFC 2475, December 1998.

[6] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems Volume II-Volume 2*, page 400. IEEE Computer Society, 1999.

[7] N. Christin and J. Liebeherr. The QoSbox: A PC-Router for Quantitative Service Differentiation in IP Networks. Technical Report CS-2001-28, University of Virginia, 2001.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–26, September 1990.

[9] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOMM'94*, pages 636–646, April 1994.

[10] M. W. Goudreau, S. G. Kolliopoulos, and S. B. Rao. Scheduling algorithms for input-queued switches:randomized techniques and experimental evaluation. *IEEE INFOCOM*, 3:1634–1643, March 2000.

[11] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. Technical Report CS-TR-96-02, UT Austin, January 1996.

[12] A. G. Greenberg and N. Madras. How fair is fair queuing. *J. ACM*, 39(3):568–598, 1992.

[13] A. Gulati. Scheduling with QoS in parallel I/O systems. Master's thesis, Rice University, Department of Computer Science, June-Nov. 2004.

[14] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[15] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Perform. Eval. Rev.*, 32(1):14–24, 2004.

[16] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. In *Proceedings of the 9th Annual European Symposium on Algorithms*, pages 62–73. Springer-Verlag, 2001.

[17] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. *SIGMETRICS Perform. Eval. Rev.*, 32(1):37–48, 2004.

[18] M. Kallahalla and P. Varman. Optimal read-once parallel disk scheduling. *Proc. 6th ACM Workshop on I/O in Parallel and Distributed Systems (IOPADS'99)*, pages 68–77, April 1999.

[19] M. Kallahalla and P. J. Varman. Optimal prefetching and caching for parallel I/O systems. In *13th ACM Symposium on Parallel Algorithms and Architectures*, pages 219 – 228, July 2001.

[20] T. Kimbrel, P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Integrated parallel prefetching and caching. In *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 262–263. ACM Press, 1996.

[21] C. Lumb, A. Merchant, and G. Alvarez. Faç ade: Virtual storage devices with performance guarantees. *File and Storage technologies (FAST'03)*, pages 131–144, March 2003.

[22] N. McKeown. The iSLIP Scheduling Algorithm for Input-Queued Switches. *IEEE/ACM Transactions On Networking*, 7:188–201, April 1999.

[23] N. Ni and L. N. Bhuyan. Fair scheduling in Internet routers. *IEEE Transactions On Computers*, pages 686–701, June 2002.

[24] S. Sarkar and L. Tassiulas. Fair bandwidth allocation for multicasting in networks with discrete feasible set. *IEEE Trans. Comput.*, 53(7):785–797, 2004.

[25] 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*, pages 44–55. ACM Press, 1998.

[26] J. Wilkes. Traveling to rome: Qos specifi cations for automated storage system management. In *International Workshop on QoS*, pages 75–91, June 2001.

[27] L. Zhang. VirtualClock: A new traffi c control algorithm for packet switching networks. In *SIGCOMM 90*, pages 19–29, September 1990.