

Scheduling with QoS in Parallel I/O systems

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

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

Abstract— Parallel I/O architectures are increasingly deployed for high performance computing and in shared data centers. In these environments it is desirable to provide QoS-based allocation of disk bandwidth to different applications sharing the I/O system. In this paper, we introduce a model of disk bandwidth allocation, and provide efficient scheduling algorithms to assign the bandwidth among the concurrent applications.

I. Introduction

Parallel I/O architectures consisting of multiple disks connected with high bandwidth interconnect are the norm in high-performance data centers and supercomputing installations. Data is distributed among the disks to enable simultaneous parallel access. Individual applications can potentially speed up their I/O by fetching data blocks in parallel from multiple disks, and buffering them in memory until required. The benefit of prefetching depend on several factors such as the temporal distribution of disk accesses, the amount of buffer memory available to smooth out uneven distributions of requests, and the amount of lookahead available. A substantial body of work exists on the problem of prefetching from multiple disks, dealing both with issues of access prediction and scheduling at the system level [15], [16], [20], [22], as well as scheduling algorithms and their analyses [1], [11], [13], [14]. Many of these issues are fairly well understood at this time.

When several concurrent applications are simultaneously sharing the I/O system, the scheduling of the I/Os becomes more complicated. Two issues of concern in this situation are maximizing utilization of the disk band-

width by multiplexing it among the concurrent tasks, and avoiding starvation of individual tasks. Even when considered separately the problems are challenging. For instance, in contrast to the case of a single access sequence for which efficient scheduling algorithms have been found [1], [11], [13]–[15], the problem of maximizing the disk bandwidth utilization while scheduling a set of n , $n > 1$, reference strings can be shown NP-complete [9]. Besides being computationally intractable and requiring a priori knowledge of the entire set of access sequences, such a minimal-length schedule may be unattractive from the viewpoint of fairness in the allocation of the disk bandwidth to individual applications.

In this extended abstract we present a model for fair disk bandwidth allocation among concurrent tasks in a parallel I/O system, and present efficient scheduling algorithms for implementing several allocation policies. By necessity, our model is an idealized abstraction of a complex parallel I/O system where mechanical disk delays and interconnect scheduling issues need to be considered. Many of these factors can be captured by a more detailed model; this work will concentrate on the higher-level scheduling issues. More details will be provided in the complete paper.

The rest of the paper is organized as follows. In Section II we describe the model and the attendant definitions. In Section III and IV we present our algorithms for scheduling with fairness and *weighted-QoS*. Section V contains some preliminary simulation results.

We review related work in section VI and end with some conclusions in section VII.

II. Overview

A parallel I/O system consists of a set of d independent disks $\mathcal{D} = \{D_1, D_2, \dots, D_d\}$. Data is stored on the disks in units of *blocks*; a block is the unit of access from a disk. In each parallel I/O operation a set of up to d blocks, one from each disk, can be accessed. The blocks fetched from the disks may be buffered in an internal memory buffer until they are required. Prefetching can greatly reduce the overall I/O latency.

A set of n independent applications or tasks is assumed to be concurrently accessing the I/O system. Each task is abstracted by a **reference string** $R_i, 1 \leq i \leq n$; R_i is the ordered sequence of blocks that is required by that application. For each reference string the system has a lookahead of $L - 1$ blocks, $L \geq 1$, beyond the current request from that string. This lookahead enables the application to prefetch blocks that will be accessed in the near future and hold them in the buffer until required.

An **I/O schedule** consists of a sequence of parallel I/O steps: in each step at most one block from each of the d disks is accessed. An I/O schedule is said to be **work conserving** if every I/O step employs maximal parallelism: that is, a disk is busy unless there is no request for that disk or there is not enough buffer space to hold the accessed block. In this paper we will assume that the buffer is large enough to hold the lookahead L of all the strings: hence in a work conserving schedule, the only idle disks are those with no requests at that step. Under this assumption of limited lookahead, it has been shown that a work conserving schedule minimizes the number of I/O steps required for any single reference string in isolation [13].

A parallel I/O step will be represented by a

fetch vector, $F = [b_1, b_2, \dots, b_n]$, where b_i is the number of blocks from reference string R_i that are fetched in that I/O step. Note that since at most d blocks can be fetched in any parallel I/O step, $\sum_{1 \leq i \leq n} b_i \leq d$. The **cumulative fetch vector**, $CF = [B_1, B_2, \dots, B_n]$, where B_i is the total number of blocks from reference string R_i fetched so far. Note that CF can be obtained by the component-wise addition of the fetch vectors for each of the preceding I/O steps. For weighted allocation of bandwidth, we let w_i indicate the relative priority assigned to a reference string R_i . The **proportionate vector** $V_p = [v_1, v_2, \dots, v_n]$ where $v_i = B_i/w_i$. Finally, we refer to the sum of the components of a vector as its **weight**.

We will be interested in constructing work conserving schedules since such schedules will maximize the disk utilization at every I/O step. In addition, we would like the schedule at each step to be **fair**: by fairness we intuitively mean that we try to fetch as evenly as possible for all the reference strings. Formally, we say that the schedule at a step is fair if the fetch vector F at that step is lexicographically smallest among all n -vectors with the same weight. We formally state the definition of lexicographic minimum below: Consider two vectors $F = [f_1, f_2, \dots, f_n]$ and $G = [g_1, g_2, \dots, g_n]$, 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 $1 \leq i \leq n - 1$. Then F is lexicographically smaller than G if and only if there is some index $k, k \geq 1$, such that $f_i = g_i, 1 \leq i \leq k - 1$ and $f_k < g_k$. For example consider the following 3-component vectors with weight 5: $[2, 0, 3]$, $[0, 3, 2]$, $[1, 2, 2]$, $[0, 5, 0]$, $[0, 4, 1]$, $[1, 1, 3]$. When arranged in non increasing order of their component values, these are the distinct vectors $[3, 2, 0]$, $[2, 2, 1]$, $[5, 0, 0]$, $[4, 1, 0]$, $[3, 1, 1]$. The lexicographically smallest of these vectors is $[2, 2, 1]$ corresponding to the most balanced distribution of the component values.

We define three different policies for scheduling.

- 1) **Locally Fair Allocation:** Achieve a fair schedule at each I/O step that is work conserving.
- 2) **Globally Fair Allocation:** Achieve a fair work-conserving schedule based on the cumulative numbers of blocks accessed by each string.
- 3) **Weighted Allocation:** Achieve a work conserving schedule in which the cumulative number of blocks accessed after any step are in proportion to the relative priorities of the reference strings.

A. Overview of the Algorithms

At any time, there are a set of candidate blocks from each reference string that can be fetched in this I/O step. These are the unfetched blocks in the lookahead for that string. The system is modeled by an augmented bipartite resource graph $G = (V \cup \{s, t\}, E \cup E_t \cup E_s)$ defined as follows:

- $V = \{D_1, D_2, \dots, D_d\} \cup \{R_1, R_2, \dots, R_n\}$ is a set of $n+d$ nodes, one for each reference string and disk in the system.
- E is the set of directed edges between nodes representing reference strings and nodes representing disks: there is an edge (R_i, D_j) whenever there is a request to disk D_j in the current lookahead window of reference string R_i .
- Distinguished vertices s and t will serve as the source and sink of paths through the graph.
- $E_t = \{(D_j, t), 1 \leq j \leq d\}$, is the set of directed edges from each disk node to t .
- $E_s \subseteq \{(s, R_i), 1 \leq i \leq n\}$, is a subset of the directed edges from s to string nodes R_i ; this subset changes dynamically as the algorithm proceeds.

We will show that the scheduling problems defined above can be mapped to finding a set of paths in a dynamically evolving resource

graph. Initially the resource graph consists of G defined above with E_s being empty. Our algorithms will maintain a **priority vector**, $\mathcal{P} = [p_1, p_2, \dots, p_n]$: p_i is the priority for string node R_i . At each step the node with highest-priority is selected, and an edge from s to this node is added to E_s . The algorithm then attempts to find a path in the current resource graph from s to t using the edge between s and the selected node.

If a path cannot be found through the currently selected node R_i , then the node is marked as **saturated**, and the algorithm will adjust the priority of R_i so that it will not be selected again. A saturated node means that it is not possible to increase the total number of scheduled disks by making a further assignment to this reference string (*i.e.* the weight of the fetch vector will not increase). An assignment to a saturated node will come at the expense of reducing the assignment to one of the previously scheduled nodes. The choice of priorities is such that this reallocation is undesirable.

If the search for a path is successful this means that the weight of the fetch vector can be increased by assigning an additional disk to the i^{th} string. The previous assignment of disks to strings might get changed to make this possible, but the *number of disks* assigned to any other string will *not change* by this reassignment. Thus the weight of the fetch vector is increased without disturbing the relative allocations needed for fairness. 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 algorithms.

Path Conditioning: In the current resource graph every edge (R_i, D_j) in the discovered path is replaced by its reverse edge (D_j, R_i) , and similarly every edge (D_s, R_t) in the path is replaced by the reverse edge (R_t, D_s) . The last edge on the discovered path from some

D_u to t is also replaced by the reverse edge (t, D_u) .

The invariant maintained by the algorithm is that the presence of an edge (D_s, R_t) in the resource graph at the start of an iteration means that currently disk D_s is assigned to string R_t . Suppose the path going through selected node R_u at this iteration is $(s, R_u, D_{j_1}, R_{j_1}, D_{j_2}, R_{j_2}, \dots, D_{j_k}, R_{j_k}, D_{j_{k+1}}, t)$. This implies that at the start of the iteration there were k assignments: disk D_{j_i} was assigned to string R_{j_i} , $1 \leq i \leq k$. When the edges between string and disk nodes are reversed, the $k+1$ new assignments will be: D_{j_1} to R_u , D_{j_2} to R_{j_1} , and so on ending with $D_{j_{k+1}}$ assigned to R_{j_k} . Note that for each of the string vertices R_{j_i} only the identity of the assigned disk was changed. Note that although related to the problems of bipartite graph matching and determining maximum flow [7], there are subtle differences that preclude direct application of either of these algorithms to our problem [9]. 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.

III. Local fairness

Problem definition: Find an assignment of disks to reference strings such that the **fetch vector**, $F=[b_1, b_2, \dots, b_n]$ has *maximal weight* and is *lexicographically minimum*.

In the resource graph G defined earlier let \hat{D} be the number of disk nodes that have at least one incident edge. Then the maximal weight of the fetch vector is \hat{D} , *i.e.* a work-conserving schedule will assign \hat{D} disks.

A. LFS algorithm

A straightforward algorithm to obtain local fairness can try all assignments made up of ordered partitions of \hat{D} into n components, in

increasing lexicographic order until a feasible schedule is found. However, the running time of such an algorithm would be exponential as there are $\theta(\hat{D}^n)$ possible partitions of \hat{D} into n parts, where the order of parts is important.

The algorithm LFS shown in figure 1 gives the maximal weight, lexicographically minimum **fetch vector** in $O((n+D)|E|)$ time. The algorithm has the structure described in overview. The priority vector begins with all components 0; at every iteration either the weight of the priority vector increases by 1, or one component (string node) is saturated. The (non saturated) component with smallest value will have the highest priority in the vector \mathcal{P} . The use of the dynamic graph created by the path conditioning step ensures that the assignments of disks at some stage do not preclude reassignment at a later stage. This observation and the pruning of saturated nodes allows the algorithm to terminate in no more than $\hat{D} + n$ iterations; each iteration requires an $O(|E|)$ path searching algorithm.

We present some lemmas related to LFS algorithm (the proofs are omitted due to lack of space).

Lemma 1: Once a disk node has been assigned to some string node it will never become free (although it may be reassigned to a different string node).

Lemma 2: LFS finds the minimum lexicographical allocation of \hat{D} disk blocks to reference strings.

Lemma 3: The running time of LFS algorithm is $O((n+D)|E|)$.

IV. Global Fairness and QoS Scheduling

The local fairness metric aims to distribute disk accesses fairly at every I/O step. There are situations where such an approach may be inadequate. If an application has a burst of requests to just a few disks, it will receive a

- (1) **LFS(Locally Fair Scheduling) algorithm:**
- (2) $\text{weight} = 0$
- (3) G is augmented resource graph, with $E_s = \phi$
- (4) Let priority vector $\mathcal{P} = [0, 0, \dots, 0]$.
Mark all elements of the vector as non-saturated.
- (5) **while** ($\text{weight} < \hat{D}$)
- (6) Choose the lowest-valued non-saturated element p_i of \mathcal{P} with ties broken arbitrarily.
Add an edge (s, R_i)
- (7) Find any path from s to t that includes the edge (s, R_i) .
- (8) **if** (no path is found)
- (9) Mark p_i as saturated in \mathcal{P} .
- (10) **else**
- (11) $\text{weight} = \text{weight} + 1$; $p_i = p_i + 1$
- (12) Update graph G by **path conditioning**

Fig. 1. $O((n+D)|E|)$ algorithm for local fair scheduling

smaller fraction of the bandwidth; it may be desirable to favor these strings once the hot spot activity passes. In other cases, a systematic albeit small difference in local allocation at each step, may result in a significant spread of the cumulative bandwidth allocation for long running applications. In order to handle such situations we define a global fairness criterion, and describe how we can adapt our basic algorithm for this purpose. Intuitively the globally-sensitive algorithm called GFS, keeps the history of accesses in the **cumulative fetch vector** CF. In the current I/O assignment, GFS will try and equalize the components of CF, by favoring strings which are lagging in the total number of blocks fetched so far.

The outline of the algorithm is quite similar to the LFS algorithm in Figure 1. As in LFS, **weight** tracks the number of disks allocated at any step of the algorithm. We let $CF = [B_1, B_2, \dots, B_n]$ be the CF vector at the start of the I/O step. Initialize the priority vector $\mathcal{P} = [p_1, p_2, \dots, p_n]$, where $p_i = B_i$ (in step 4): p_i tracks the total number of blocks that have been fetched for the i^{th} string. In the main loop, strings are scheduled in increasing order

of blocks accessed, so that strings which have the largest slack (or minimum B_i) at any step are given a chance to catch up (in step 6). The rest of the algorithm remains the same. The running time of GFS is also $O((n+D)|E|)$.

A. Weighted allocation

For QoS, we want to assign unequal proportions of resources to the tasks needing them. We study the problem of assigning disk bandwidth in a skewed manner to multiple reference strings. Let the **proportionate vector** $V_p = [v_1, v_2, \dots, v_n]$, where $v_i = B_i/w_i$ for i^{th} string. The problem can be stated more formally as follows:

Problem definition: Find an assignment of disks to reference strings that maximizes the weight of the fetch vector and lexicographically minimizes the proportionate vector $V_p = [v_1, v_2, \dots, v_n]$.

The algorithm for weighted allocation called WeAB, keeps the history of accesses in the **cumulative fetch vector** CF. Initialize the priority vector $\mathcal{P} = [p_1, p_2, \dots, p_n]$ as $p_i = B_i/w_i$. Intuitively p_i indicate the proportionate bandwidth given to the i^{th} string so far. In the current I/O step, WeAB will assign disks to try and equalize the components of P , by favoring strings which are lagging in their proportion of blocks fetched. Note that assigning a disk to R_i increases p_i by $1/w_i$ instead of 1. In the main loop, strings are scheduled in decreasing order of proportionate slack, so that strings which have the largest proportionate slack at any step are given the first chance to decrease their slack. Mathematically, the string with the minimum value of $(B_i + 1)/w_i$ has the highest priority. At each step either a string node gets saturated or the weight increases by 1. It continues until \hat{D} disks are scheduled. A more formal description of the algorithm is given in figure 2. The WeAB algorithm also runs in $O((n+D)|E|)$ time.

- (1) **WeAB algorithm:**
- (2) weight = 0
- (3) G is augmented resource graph with $E_s = \phi$
- (4) Let *priority vector* $P = [p_1, p_2, \dots, p_n]$, where $p_i = B_i/w_i$. Mark all p_i 's as non-saturated
- (5) **while** (weight < \hat{D})
- (6) Choose the non-saturated element p_i of \mathcal{P} such that $(B_i + 1)/w_i$ is minimum with ties broken arbitrarily. Add an edge (s, R_i)
- (7) Find any path from s to t that includes the edge (s, R_i) .
- (8) **if**(no path is found)
- (9) Mark p_i as saturated in \mathcal{P} .
- (10) **else**
- (11) weight = weight + 1; $B_i = B_i + 1$; $p_i = B_i/w_i$
- (12) Update graph G by **path conditioning**

Fig. 2. $O((n+D)|E|)$ algorithm for weighted allocation

V. Results

We evaluate the algorithms for *global and proportional allocation* by simulations. For **global fairness**, we performed an experiment with 4 reference strings (generated randomly) and 64 disks. The amount of lookahead is equal to number of disks for each reference string. Figure 3 shows the actual distribution of bandwidth among the strings. The figure confirms that the bandwidth is equally distributed among all four strings. Initial fluctuations are related to the distribution of disk requests in the reference strings and the work conserving nature of the algorithm, that prevents any disk from idling if some request for that disk can be scheduled. For weighted allocation, we performed experiments with 3 randomly-generated reference strings and 64 disks. Strings R_1, R_2 and R_3 have been assigned relative priorities of 0.5, 0.3 and 0.2 respectively. Figure 4 shows the actual bandwidth allocated to the strings using *WeAB* algorithm. The result shown in the figure indicates that *WeAB* algorithm allocates differentiated bandwidth to various strings according to the weights assigned to them.

We also experimented with skewed input

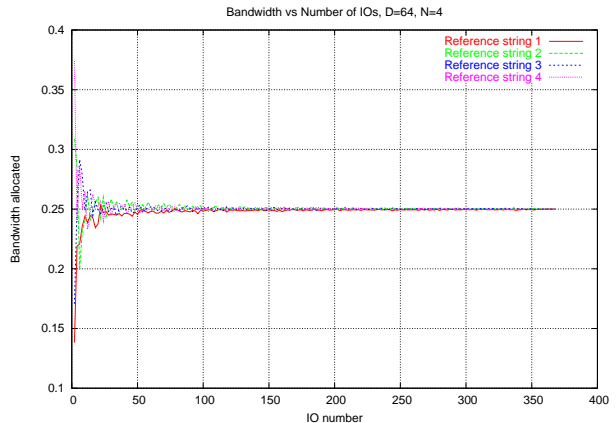


Fig. 3. GFS:fair allocation for 4 strings

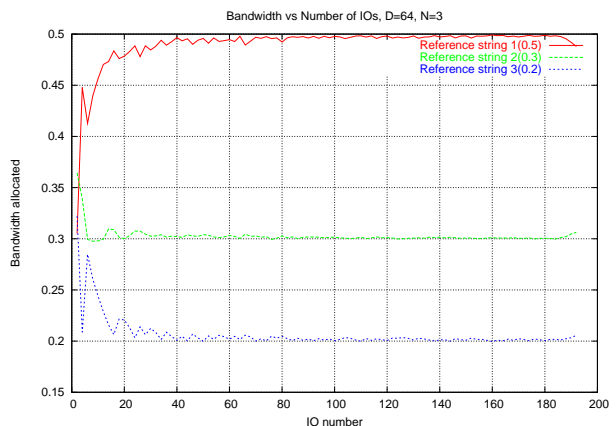


Fig. 4. *WeAB* with weights 0.5,0.3,0.2

models. We are currently extending the evaluation to incorporate physical disk parameters and variable access times. A more comprehensive evaluation of the algorithms will be presented in the full paper. We also experimented with skewed input models and we will be presenting a more comprehensive evaluation of algorithms in the full paper.

VI. Related work

Several disk scheduling policies have been proposed for single as well as multiple disks. Haritsa et al. [10] have presented efficient and fair scheduling algorithms for a single disk. They try to achieve fairness at the cylinder

level by proposing strategies for disk head movement after a request has been serviced. The YFQ [4] algorithm which is an extension to generalized processor sharing (GPS) model of proportional resource allocation, allows applications to reserve a fixed share of disk bandwidth. Cello [21] provides a scheduling framework for a heterogeneous mix of applications accessing the disk. It employs a 2-level scheduling framework, where requests from applications are put into different queues based on their needs at the first level and a common scheduler reorders the requests selected from these queues to minimize seek and rotational delays at the second level. Our approach however, provides bandwidth allocation across multiple disks. By batching requests that belong to a single disk many of the lower level optimizations suggested in these works can be employed along with our higher level framework.

Lumb et al. presented Facade [6] that is an approach to fulfill *Service Level Objective* (SLO) of independent workloads accessing a storage system. Facade provides a dynamic trade-off between the IO rate and average latency by real-time scheduling and feedback based control of disk queue lengths. Their approach assumes the availability of a capacity planner for admission control. Our approach 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 [12] that assign virtual start and finish times to requests based on their cost, client queue and arrival time. In [12], the server is considered as a single resource shared by all the clients, whereas we attain fairness for multiple input streams requesting multiple resources (disks).

Many schemes have been developed for fair scheduling in Internet routers. Schemes such as iSLIP [17], shakeup [8] try to achieve

high throughput and fairness by looking for maximal bipartite matching between inputs and outputs at each point of scheduling. iFS [18] provides fairness and QoS guarantees by assigning virtual start and finish times to each packet. These algorithms try to obtain an 1-to-1 mapping between input and output ports instead of 1-to-many mapping required for our problem. Iteratively applying bipartite matching does not guarantee a fair schedule in our case because it gives an irrevocable mapping that cannot be changed in future iterations. Many other architectures have been proposed for differentiated allocation of bandwidth to flows such as diffservs [19] and QoSBox [5]. None of them can be used for QoS in automated storage systems [23]. As stated in [23], performance of storage systems depends a lot on the current state and storage protocols are not amenable to packet dropping and traffic shaping.

Automated tools such as Hippodrome [3], Minerva [2] can be used to design storage systems, if the capacity requirements and workload characteristics are known a priori. These tools go through the iterative process of designing and evaluating the system. Once the workloads or any other requirement changes the whole process needs to be repeated again. Our approach can handle short term fluctuations and provide weighted-QoS even in the presence of unpredictable workloads.

VII. Conclusions

In this paper we explored scheduling schemes to provide bandwidth allocation in parallel I/O systems. LFS algorithm obtains local fairness at each I/O step. GFS algorithm provides fair allocation over larger time windows by keeping a history of blocks accessed by each string, and WeAB is a general scheme to obtain an I/O schedule with prioritized allocation of disk bandwidth to various strings. WeAB can support an economic model of

services provided by a data center that need to provide differentiated bandwidth to various customers. We demonstrated via simulations that the WeAB algorithm matches the bandwidth allocated to various application to their desired priority and GFS obtains a fair schedule. All these algorithms are work conserving and provide high throughput. We plan to implement our algorithms in an actual system to get a better idea about performance and scalability of our approach in future.

Acknowledgments

Support for the first author was provided by the National Science Foundation under Grant CCR-0105565. Research time during IPA assignment of the second author was provided by the National Science Foundation under the IRD program.

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] G. A. Alvarez and et al. Minerva: an automated resource provisioning tool for large-scale storage systems. In *ACM Transactions on Computer Systems*, pages 483–518, November 2001.
- [3] E. Anderson and et al. Hippodrome: running circles around storage administration. In *File and Storage Technology (FAST'02)*, pages 175–188, January 2002.
- [4] 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.
- [5] N. Christin and J. Liebeherr. The QoSbox: A PC-Router for Quantitative Service Differentiation in IP Networks. Technical report, 2001.
- [6] A. M. Christopher Lumb and G. Alvarez. Facade: Virtual storage devices with performance guarantees. *File and Storage technologies (FAST'03)*, pages 131–144, March 2003.
- [7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. *The MIT Press*, pages 579–598, 1999.
- [8] 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.
- [9] A. Gulati. Scheduling with QoS in parallel I/O systems. *Master's thesis in progress, Rice University*, June 2004.
- [10] J. R. Haritsa and T. Pradhan. Fair disk schedulers for high performance computing systems. *Proc. of International Conf. on High Performance Computing*, December 1995.
- [11] 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.
- [12] 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.
- [13] 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)*, April 1999.
- [14] 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.
- [15] 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.
- [16] D. Kotz and R. Jain. I/O in parallel and distributed systems. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 40, pages 141–154. Marcel Dekker, Inc., 1999.
- [17] N. McKeown. The iSLIP Scheduling Algorithm for Input-Queued Switches. *IEEE/ACM Transactions On Networking*, 7:188–201, April 1999.
- [18] N. Ni and L. N. Bhuyan. Fair scheduling in internet routers. *IEEE Transactions On Computers*, pages 686–701, June 2002.
- [19] K. Nichols, S. Blake, F. Baker, and D. Black. An architecture for differentiated services. Technical report, December 1998.
- [20] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95. ACM Press, 1995.
- [21] 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.
- [22] G. A. S. Whittle, J.-F. Paris, A. Amer, D. D. E. Long, and R. Burns. Using multiple predictors to improve the accuracy of file access predictions. In *20th IEEE Conference on Mass Storage Systems and Technologies*, April 2003.
- [23] J. Wilkes. Traveling to rome: Qos specifications for automated storage system management. In *International Workshop on QoS*, pages 75–91, June 2001.