Asynchronous Scheduling of Redundant Disk Arrays

Peter Sanders

***Abstract*—Allocation of data to parallel disk using redundant storage and random placement of blocks can be exploited to achieve low access delays. New algorithms are proposed which improve the previously known shortest queue algorithm by systematically exploiting that scheduling decisions can be deferred until a block access is actually started on a disk. These algorithms are also generalized for coding schemes with low redundancy. Using extensive simulations, practically important quantities are measured which have so far eluded an analytical treatment: The delay distribution when a stream of requests approaches the limit of the sytem capacity, the system efficiency for parallel disk applications with bounded prefetching buffers, and the combination of both for mixed traffic. A further step towards practice is taken by outlining the system design for α: *a*utomatically *l*oad-balanced *p*arallel *h*ard-disk *a*rray. Additional algorithmic measures are proposed for α that allow variable sized blocks, seek time reduction, fault tolerance, inhomogeneous systems, and flexible priorization schemes.**

***Index Terms*—Parallel disks, lazy scheduling, asynchronous, random redundant storage, duplicate allocation, soft real time, bipartite matching, queuing theory**

I. Introduction

Ever larger data sets arise in important applications like data mining, electronic libraries, web servers, virtual reality, geographic information systems, or scientific computing. Often, no size limits are in sight. To process these massive data sets with high performance, many disk drives have to be operated in parallel. Consequently, disk arrays have become a multi-billion-dollar business. For cost efficiency and data sharing, a parallel disk server or storage area network has to support requests for data blocks stemming from different applications concurrently. In this situation it is a challenging task to get both high throughput and low access latencies for interactive or time critical applications. New applications like video-on-demand or interactive computer graphics even require soft real time guarantees for the latencies of disk accesses.

Current parallel disk systems use *mirroring*, i.e., replication of the data or parity blocks to achieve fault tolerance. This paper studies how this redundancy which is needed anyway can be exploited to reduce access latencies caused by several requests directed to the same disk. The main issue here is to schedule the request to the disks in such a way that access latencies get small. The achievable improvement is so dramatic compared to traditional techniques like mirroring that a very simple programming model becomes feasible: A parallel disk server with D independent disks is viewed as one big disk with D-fold bandwidth and D-fold capacity that can access D arbitrary blocks concurrently. This parallel random access model is easier to use than the lower level view of D independent disks where the additional issue of appropriate placement of data to disks has to be taken care of. Even if the predominant access patterns are known, this can be a cumbersome task. In difficult cases with unpredictable, dynamically changing access patterns, no good load balance is possible if we do not exploit both redundancy and random placement of data. The model considered here is the asynchronous pendant to the synchronous multi-head parallel disk model of Aggarwal and Vitter [1] which turned out to be very convenient for devising external memory algorithms [2].

Max-Planck-Institute for Computer Science, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, sanders@mpi-sb.mpg.de. Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). An extended abstract of this paper was published in the proceedings of the ACM Symposium of Parallel Algorithms and Architectures (SPAA 2000).

A. Basic Model

We now introduce a simple form of our system model that suffices to understand most of this paper. Generalizations to this model are introduced later when appropriate. The overview in Section I-B gives a few pointers.

Consider a system with D identical disks serving an asynchronous stream of requests for accessing *logical* disk blocks. Logical blocks are stored using *random duplicate allocation* (RDA), i.e., there are two physical copies that are allocated to random *different* disks using hash functions or a RAM resident directory.

Writing for RDA is a relatively easy problem. Previous results for a synchronous model transfer [2].[1] Read requests to the same logical block can be merged into a single request. Read requests to cached blocks or blocks in the write buffer are easier to serve. Therefore, the present study concentrates on read requests to different, uncached logical blocks.

The time $\mathrm{rel}(e)$ when a request becomes known to the system is called the release time of e. Under these assumption our knowledge about the system at a point of time t extends to all the requests e with $\mathrm{rel}(e) \leq t$. We also know which requests have already been served and

[1] The writing algorithms considered there can also be viewed as asynchronous algorithms. To transfer the analytic bounds, we could convert an asynchronous request stream into synchronized batches using an additional buffer of D blocks.

wich are currently being served by a disk. A disk needs time t_e for accessing one of the copies of block e. When not otherwise stated, we assume unit access time $t_e = 1$ for all requests. Since access times are assumed to be known, we can also predict the time $t_{\mathrm{idle}}(d)$ when a busy disk will fall idle again.

A *schedule* has to map every request e to a disk d_e that has a copy of the requested block and to a time interval $[t_{\mathrm{start}}(e), t_{\mathrm{start}}(e) + t_e)$ such that $t_{\mathrm{start}}(e) \geq \mathrm{rel}(e)$ and such that no two intervals overlap on the same disk. The *delay* of a request e is $t_{\mathrm{start}}(e) + t_e - \mathrm{rel}(e)$. We use the term *maximal* delay for the maximal predicted delay of a request currently in the system.

We are most interested in request streams that nearly saturate the system. Often we use periodic arrivals of unit size requests such that request i arrives at time $\mathrm{rel}(i) = (1+\varepsilon)i/D$. In this case, we are particularly interested in the distribution of delays for small ε where the system is nearly saturated. An even simpler case are *batched arrivals* where R request all arrive at time $\mathrm{rel}(e) = 0$. In this case, the maximal delay coincides with the well known measure of *make span* often used for measuring the quality of batched schedules.

B. Overview

The structure of this papers is governed by two main topics. The first topic is to explore the design space of algorithms for asynchronous parallel disk access using random placement and redundancy. The second topic is evaluation of these algorithms. This evaluation is complex because there is an abundance of open theoretical questions and since simulations have to explore a huge space with many parameters. Hence, we restrict the evaluation to duplicate allocation and the simple machine model from Section I-A. However, within this simple model we aim at a comprehensive set of tests.

Section II discusses simple heuristics for improving previously known algorithms which have the advantage to be very fast and parallelizable. A more systematic approach to scheduling is pursued in Section III. Scheduling algorithms based on bipartite matching are introduced that minimize the maximal delay.

To evaluate these new algorithms, we proceed in two steps. First, Section IV proves that fluctuations in the arrival rates of requests have very little influence on latencies assuming a “queuing theory style” load model. Therefore, the experimental evaluation in Section V can get surprisingly general results by simply simulating periodic request arrivals. But Section V also explores the limits of queuing type systems in particular in applications that completely saturate the system. It turns out that closely related variants of our algorithms work well even if request from time critical applications have to coexist with applications saturating the system.

Sections VI and VII generalize the algorithmic results in two directions that are important for practical usefulness. Section VI looks at more flexible storage schemes where logical blocks are encoded in w pieces in such a way that retrieving any r of these pieces suffices to reconstruct the information stored. In particular, the case $w = r+1$ allows a flexible tradeoff between storage overhead and performance.

Additional issues like variable block sizes, multi-zone disks, reducing seek times, fault tolerance, communication delays, inhomogeneous systems, and system tuning are concentrated in Section VII to keep the rest of the paper simple. All these things combined, we get a starting point for the design of a general purpose parallel disks server based on the scheduling algorithms presented here. Section VIII summarizes the results and mentions some open questions.

C. Related Work

An automatic load balancing approach widely used in practice is *striping* [3], [4]. In our terminology that means a logical block size D times larger than the physical block size, where each logical block is dispersed over all disks. This works for scanning large amounts of consecutive data but is of little help for smaller access granularity. Without redundancy, worst case access patterns can direct all request to a single disk resulting in arbitrarily large delays for very small arrival rates of just above one request per time unit. Even redundancy is of limited help as long as the allocation strategy is deterministic. A lower bound by Armen [5] shows that it can take time $\Omega\left(T \frac{\log(N/D)}{\log\log(N/D)}\right)$ to complete a computation on N blocks and performing TD block accesses.

Load balancing by random placement of data is a well known technique (e.g., [6], [7]). In this situation, delays for read requests can be investigated analytically using generating function techniques similar to those used in [2] for write buffering. Average delays behave like $\Theta(1/\varepsilon)$ for periodic arrivals every $(1+\varepsilon)/D$ time units, i.e., they can become rather large as the arrival interval approaches the limit of $1/D$. Nonredundant random placement has been proposed for the parallel file system RAMA [7].

Combining random placement and redundancy has first been considered in parallel computing for PRAM emulation [8] and online load balancing [9]. For scheduling disk accesses, these techniques have been used for multimedia applications [10], [11], [12], [13], [14], [15]. These papers use shortest queue, do not specify the scheduling algorithm, or schedule large batches in a synchronous fashion.

Some RAID arrays use load balancing techniques to spread read requests over primary and mirror disks equally. This approximately halves the observed delays.

In Section V-A we will see that one can do much better.

Even the simplest scheduling heuristics for RDA are quite difficult to treat analytically for asynchronous request arrivals and small ε. Vvedenskaya et al. [16] and Mitzenmacher [17] analyze the shortest queue heuristics as $D \to \infty$ for Poisson arrivals with fixed arrival rate $\lambda = D/(1+\varepsilon)$ and exponentially distributed service times. There are theoretical results on system models which keep the number of requests in the system fixed (e.g., [18]). But so far none of these models seems to be able to approximate the behavior of disk servers. Also, many of the techniques for analyzing RDA seem to be inaccurate by significant constant factors when the number of requests in the system is much larger then D. (A recent analysis of shortest queue for the case of high loads only applies to a batched model [19].) Adler et al. [20] consider an algorithm similar to the lazy queue algorithm based on synchronized rounds of allocation and job consumption. For small arrival rates $(1+\varepsilon \geq 6e > 16)^2$ the expected maximal delay is shown to be bounded by $O(\log\log D)$. This synchronous formulation has the problem that some requests are executed on *both* disks. All these theoretical difficulties led us to adopt a mostly simulation based approach in this paper.

Several scheduling algorithms are known which reduce the maximal delay for scheduling a batch of $|R|$ requests to $O(|R|/D)$ with high probability [8], i.e., independent of D. Korst [12] explained how an optimal schedule for batches of requests can be computed using maximum flow computations and in [2] it is shown that the maximal delay is bounded by $\lceil |R|/D \rceil + 1$ for optimal schedules with high probability. Further generalizations for batched scheduling including variable block sizes, disk failures, and communication overheads can be obtained using similar techniques [22]. Batched scheduling algorithms can be converted into asynchronous scheduling algorithms by pipelining batched schedules. While one batch is executed on the disks, the newly arriving requests are retained until the previous batch finishes. Then the retained requests are scheduled in a batch and assigned to the disks next. This strategy, applying the bounds from [2], yields an algorithm with maximal delay independent of D. However, the average delay is quite high (e.g. [23]). For practically interesting D, even the maximal delay is higher than for the asynchronous shortest queue heuristics. Originally, we thought that this problem could be solved using essentially the same max-flow based algorithm from [2] modified to update the schedule whenever a new request arrives.[3] However, the overall performance was disappointing because the algorithm cannot distinguish between new and old requests. Old request should be preferred to limit the frequency of large delays. This observation was the motivation for developing algorithms which explicitly handle delays.

Berenbrink et al. [24] also propose a scheduling algorithm for RDA based on bipartite matching. They analyze the competitiveness of several online scheduling algorithms compared to an optimal offline schedule. However, they use a quite different model (synchronous arrivals, deterministic placement, and quality is fraction of requests served within a deadline), so that the competitiveness of the model considered here remains an interesting open question. The contribution of Section III is the observation that time slots need not be synchronized in the online algorithm and the development of an efficient, asynchronous implementation.

[2] This restriction is improved in [21] for $D \to \infty$.

[3] Similar considerations could be made for algorithms which approximate the algorithm from [2], e.g. [23].

II. Simple Algorithms

For the simple scheduling algorithm to be discussed here it is convenient to adopt the well known terminology of graph theory as follows:

At any point in time the known part of the scheduling problem can be can be viewed as an undirected *allocation graph* $G_a = (\{1,\dots,D\}, E)$ where nodes represent disks. A request for a logical block that has copies on disks i and j is represented by an undirected edge $\{i,j\} \in E$. The adjacency lists of the disks are ordered by the release time of the requests, i.e., a newly arriving request $\{i,j\}$ is simple appended to the adjacency lists of disks i and j. Figure 1 gives a simple example for an allocation graph with added information on possible schedules.

Our algorithms focus on the question which copy of a block is actually accessed without explicitly specifying *when* the block is to be accessed. This information is sufficient to derive a complete schedule that is at least as good with respect to maximal and average delays as any other legal schedule that retrieves the blocks from the same disks. The following *FIFO rule* derives such a schedule: For each disk d process the blocks scheduled to d in arrival order and as early as possible. We omit the trivial proof that is based on the observation that removing unnecessary waiting times can only decrease delays and that blocks accessed out of order can be swapped without increasing maximal or average delays.

FIFO-schedules can be compactly represented by a directed *schedule-graph* G_s where a directed edge $e = (u,v)$ indicates that request e is served on disk u. Now, several scheduling algorithms can be described quite compactly. They all have in common that a newly arriving request $\{u,v\}$ is immediately started if u or v are idle (one disk is chosen in some unbiased way if both are idle). In this case, the request is not inserted into the schedule graph or allocation graph.

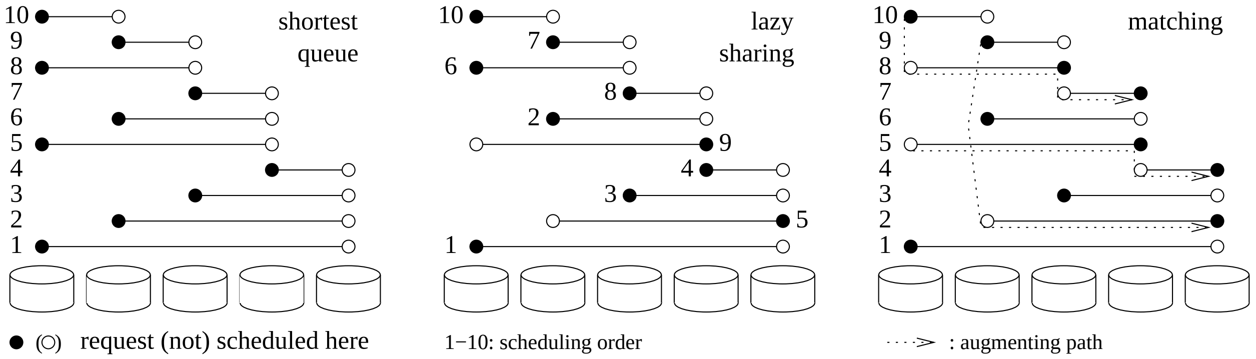

Fig. 1. Example for the performance of shortest queue, lazy sharing, and matching respectively, assuming all request arrive at time 0. For shortest queue ties are broken by picking the leftmost available copy. For lazy sharing it is assumed that the first requests are committed from left to right.

A. Eager Algorithms

We first consider scheduling algorithms which maintain a schedule at all times. The algorithms presented in Section III also have this property but we start with the well known algorithm *shortest queue*:

A new request arriving at time t and schedulable on disks u and v is appended to the adjacency lists of u and v and directed from the disk with smaller out-degree Γ^+. Ties are broken according to $t_{\text{idle}}(u)$ and $t_{\text{idle}}(v)$. An obvious generalization to arbitrary service times chooses the disk with smallest estimated load (= time needed to serve the requests already assigned).

B. Lazy Algorithms

Lazy scheduling algorithms delay decisions to the latest possible moment to have more information available. We start with a very simple implementation of this idea.

Lazy Queue. On a request arrival the edge $\{u,v\}$ is simply appended to the adjacency lists of u and v (as in [20]). When a disk finishes a request, it atomically removes the next request from the local queue and deletes this request from its other queue. In a distributed memory implementation this can be implemented using a single message exchange. [4]

As long as service times are predictable, the lazy queue algorithm behaves identical to the shortest queue algorithm. But when service times are unpredictable, lazy queue behaves like an "omniscient" shortest queue algorithm that knows all the service times. We omit the straightforward proof based on induction over the requests arrivals. This flexibility of lazy algorithms is an important practical advantage since in many cases only inaccurate information about service times is available. Simulations in Section V-F validate this reasoning.

[4] This waiting time can be overlapped with useful work by having more than one request committed to a disk.

Lazy Sharing. The lazy queue heuristics has the disadvantage that some requests may be executed on a highly loaded disk although they could be executed on a disk with lower load. For example, a disk should not grab its first available request (d,d') if disk d' has no other requests that it could work on. More generally, we use the following heuristic modification: When disk d falls idle, let $e_i = (d, v_i)$ denote the i-th member of its adjacency list. Disk d then executes the first request e_i for which $\Gamma(d) - i < \Gamma(v_i)$ where $\Gamma(v)$ denotes the degree of a node v in G_a. Requests 1, 2, 3, 5, 6, 7, 9, and 10 in Figure 1 are scheduled using this rule. A refined lazy sharing algorithm avoids large latencies by terminating the search for a request to be executed when the additional delay suffered by e_i becomes "too large".

In Section V we use the following variant: The search is terminated if request e_i is not the first request on disk v_i. Requests 3, 4, and 8 in in Figure 1 are scheduled using this rule. Furthermore, let e'_i and e'_d denote the request following e_i on disks d and i respectively. The search is also terminated if the maximal delay among e_i, e'_i, and e'_d is minimized by executing e_i on disk d. The delays are computed based on the assumptions that the requests preceding e_i will not be executed on d, that e' will be executed on d, and that e'' will be executed on v_i. Figure 2 gives an example where this rule would apply.

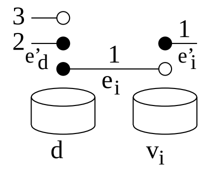

Fig. 2. An example where the last rule in the definition of lazy sharing applies. (Edges are labeled with their release time.)

C. Hybrid Algorithms

A problem of the lazy algorithms presented above is that it is difficult to extract meaningful information from the allocation graph alone. Therefore, *hybrid algorithms* make sense that combine any eager algorithm with a

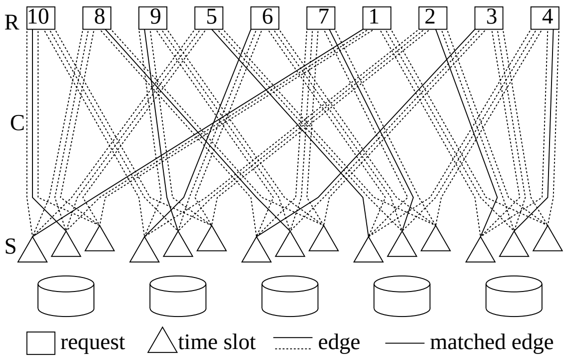

Fig. 3. Representation of a schedule by an R-perfect matching in a bipartite graph.

heuristics which uses the information provided by the schedule graph to further improve the schedule when a disk falls idle. Again, we start with a very simple rule:

Stealing. When a disk d falls idle, and its out-degree in the schedule graph is zero but its in-degree is nonzero, it "steals" the first adjacent request and executes it on d.

Local Optimization. When a disk d falls idle, it scans its adjacency list (both incoming and outgoing edges) and reverses edges if this "improves" the schedule. For example, we have implemented a variant which commits a request if executing it on the other possible disk would lead to a large delay or larger average load.

III. SCHEDULING USING BIPARTITE MATCHINGS

We now explain how bipartite matchings can be used to find schedules which are optimal in the sense at any point in time the maximal delay of the requests known to the system is minimized. Hence, schedules are globally optimal over all disks although they are still only locally optimal with respect to the time scale.

Scheduling options are now represented by a bipartite graph $G_2 = (R \dot\cup S, C)$. In the example of Figure 3, the set of known requests R forms the upper side of the graph. The lower side S represents *time slots* of disks, i.e., Slot (d,i) represents the time interval $[t_{\mathrm{idle}}(d)+i-1, t_{\mathrm{idle}}(d)+i)$ on disk d. Edges of G_2 represent possible assignments of requests to time slots. A request $\{u,v\}$ is connected to all slots of the form (u,i) and (v,i) for $i \in \mathbb{N}$. We associate a *weight* w with an edge $(r,(d,i))$ that corresponds to the delay suffered by r when scheduled to time slot (d,i), i.e., $w = t_{\mathrm{idle}}(d) + i - \mathrm{rel}(r)$.

Any legal schedule that does not keep a disk superfluously idle can be described by a matching M in G_2 that matches all request in R to exactly one time slot in S. In graph theory such a matching is called *R-perfect matching*. The matching in Figure 3 corresponds to the optimal schedule shown in Figure 1. This matching representation can now be used to optimize schedules in several respects starting with well known algorithms for computing matchings [25].

A. Minimizing the Maximal Delay

If there is a schedule with maximal delay $\hat{L}$, it can be found by constraining the bipartite graph G_2 to those edges with weight at most $\hat{L}$. As a side effect, this also limits the potentially infinite set of slots to a set of size at most $D\hat{L}$. Since there can be at most $D|R|$ different delays in a FIFO schedule, the minimal maximal delay $\hat{L}^*$ can be found by binary search using $O(\log(D \cdot |R|))$ computations of maximum cardinality bipartite matchings.

Although the algorithm presented so far runs in polynomial time, actually computing a schedule in real time for each new request might appear like a formidable task. However, there are many optimizations which make it quite practical. We maintain the invariant that G_2 contains one unmatched slot per disk, i.e., at any point in time only $|R|+D$ slots need to be explicitly represented. We compute the matching incrementally. When a new request $e = \{u,v\}$ arrives and hence becomes active, e is provisionally mapped to the free slot of disk u or v using the shortest queue heuristics. If the delay suffered by this decision is small, say below the estimate $|R|/D$ for the average delay, nothing needs to be done. Otherwise, we know from matching theory, that it suffices to consider *augmenting paths* from e to a free slot s which alternate between unmatched and matched edges. For any such path P, we get a new R-perfect matching by converting all matched edges on P to unmatched edges and all unmatched edges to matched edges. Call the maximum weight of an unmatched edge on P the *weight* of P. We get a matching with optimal $L_{\max}$ by picking an augmenting path with minimum weight. This minimum weight path can be found using a variant of Dijkstra's algorithm.

So far, we have an $O((|R|+D)\log(|R|D)+|R|^2/D)$ time algorithm for scheduling a single request assuming Fibonacci heaps for the priority queue used in Dijkstra's algorithm and an estimate of $|R|^2/D$ for the number of edges in the bipartite graph. First simulations with this algorithm indicated that for large D it is still too slow by an order of magnitude. Therefore, several additional optimizations were exploited. First, the $\Omega(R)^2/D$ edges of the bipartite graph need not be stored explicitly. The current implementation only stores a list of time slots for each disk. Since a request is always matched to a time slot, it is stored in the same object as the time slot. These measures not only decrease the number of pointer dereferences during the search but also make the algorithm more cache efficient by increasing locality and by making it more likely that the entire data set (of size $O(|R|+D)$) fits into cache.

Furthermore, we use a faster priority queue data structure which is not comparison based and a relaxation which allows non-optimal paths if their maximum weight is small. The last optimization requires more explanation. Consider a bipartite graph with large $|R|$ so that a large average delay is to be expected. Then most requests are connected to many occupied slots which promise an unattainably small delay. These connections form many long paths which have small weight but do not lead to a free slot. A strict implementation of Dijkstra's algorithm would have to explore all these paths to verify that none of them leads to a free slot. This problem can be largely avoided by exploring low weight edges in a breadth first fashion. This way the search will usually find a fairly low weight path with few edges without exploring a large part of the graph. Both optimizations can be combined by replacing the priority queue by an array of *buckets*. Each bucket B_i stores a FIFO queue of nodes reachable using paths of weight in the interval $(x_{i-1}, x_i]$. We set $x_0 = -\infty$, and $x_i = \alpha + i\beta$ for appropriate parameters α and β.

We find a path of approximately minimum weight (at most a bucket width away from the true minimum) by approximating the weight of a path by the index of the bucket containing its interval. In the simulations from Section V we have $\alpha = |R|/D - \beta$ and $\beta = 0.1$, i.e., paths with weight below average are all in bucket 0 so that time consuming search for augmenting paths with weight below average is avoided. Measurements discussed in Section V-A indicate that this implementation approach is fast enough to serve hundreds of disks by a centralized processor.

B. Minimizing the Total Delay

The sum of all delays of requests in the system can be minimized by computing a minimum weight maximum cardinality matching in G_2. Similar to the maximal delay case explained above, this can be done using a shortest path calculation for each new request. However, augmentation is more expensive now since it not only involves a shortest paths but the shortest path tree of all nodes reached during a shortest path search (refer to [26, Section 7.8] for details).

Finally, we can find a schedule with minimal total delay among all schedules which minimize the maximal delay by first optimizing the maximal delay, then constraining the edges to those which conform to this delay and then finding a schedule with minimum total delay in this pruned bipartite graph.

IV. Nonperiodic Request Arrivals

In a real system, requests will usually not arrive with a fixed interarrival time. We argue that this has little influence on the performance of the server if these fluctuations are not very large.

From the point of view of queuing theory, the most natural model would be a Poisson stream of requests with arrival rate $\lambda = D/(1+\varepsilon)$. The following theorem shows that the fluctuations in this system have only marginal influence on the average delay if D is large.

Theorem 1: Assume a scheduling algorithm which can serve a periodic stream of requests with interarrival time $(1+\varepsilon)/D$ such that the average delay is bounded by $f(1/\varepsilon)$ for some differentiable concave[5] function f. Then there is a scheduling algorithm which can service a Poisson stream of requests with arrival rate $D/(1+\varepsilon)$ with average delay at most

$$\bar{L} \leq f\left(\frac{1}{\varepsilon}\right) + O\left(\sqrt{\frac{f'(1/\varepsilon)}{\varepsilon^2 D}}\right) \quad .$$

Note that we do not need to know f since the main value of the theorem is to tell us that knowledge about periodic arrivals (e.g., via simulations) transfers to system with fluctuations in arrival rates.

Proof: We can convert[6] the Poisson stream into a periodic stream with interarrival time $(1+\varepsilon')/D$ by putting a "leaky bucket" [27, Section 5.3.3] between the request stream and the server. The bucket forwards a request every $(1+\varepsilon')/D$ time units. We are free to choose the parameter ε'. The leaky bucket behaves as an $M/D/1$ queuing system. Its average delay is

$$\bar{L}_1 = \frac{1+\varepsilon'}{D}\left(\rho + \frac{\rho^2}{2(1-\rho)}\right)$$

where $\rho = (1+\varepsilon')/(1+\varepsilon)$ [28, Equation 7.39]. If the bucket is empty, dummy requests can be injected. Exploiting the concavity of f, the average delay in the server is at most

$$\bar{L}_2 \leq f\left(\frac{1}{\varepsilon'}\right) \leq f\left(\frac{1}{\varepsilon}\right) + \left(\frac{1}{\varepsilon'} - \frac{1}{\varepsilon}\right) f'\left(\frac{1}{\varepsilon}\right) \quad .$$

We set

$$\varepsilon' = \varepsilon - \varepsilon\sqrt{\frac{1+\varepsilon}{2f'(1/\varepsilon)D}}$$

and estimate the total delay $\bar{L} = \bar{L}_1 + \bar{L}_2$.

Using $\varepsilon' \leq \varepsilon$ and $\rho \leq 1$, the average delay in the leaky bucket is

$$\begin{aligned}\bar{L}_1 &= \frac{1+\varepsilon'}{D}\left(\rho + \frac{\rho^2}{2(1-\rho)}\right)\\ &\leq \frac{1+\varepsilon}{D}\left(1 + \frac{1}{2(1-\rho)}\right)\\ &= \frac{1+\varepsilon}{\varepsilon}\sqrt{\frac{f'(1/\varepsilon)(1+\varepsilon)}{2D}} + \frac{1+\varepsilon}{D}\end{aligned}$$

[5] This assumption could be lifted at the price of a more complicated formulation of the theorem. However, theoretical considerations and all our measurements suggest that f is indeed concave.

[6] In practice, we will directly inject the requests into the system. At least for the shortest queue algorithm it is easy to show that this can only decrease delays.

For the average delay after the leaky bucket we exploit

$$\frac{1}{\varepsilon'} - \frac{1}{\varepsilon} = \frac{\varepsilon - \varepsilon'}{\varepsilon^2} + O\left(\frac{(\varepsilon - \varepsilon')^2}{\varepsilon^3}\right)$$

and get

$$\bar{L}_2 \leq f\left(\frac{1}{\varepsilon}\right) + \frac{1}{\varepsilon}\sqrt{\frac{f'(1/\varepsilon)(1+\varepsilon)}{2D}} + O\left(\frac{1}{\varepsilon D}\right) \quad .$$

Summing the bounds for $\bar{L}_1$ and $\bar{L}_2$ yields

$$\bar{L} \leq f\left(\frac{1}{\varepsilon}\right) + \frac{2+\varepsilon}{\varepsilon}\sqrt{\frac{f'(1/\varepsilon)(1+\varepsilon)}{2D}} + O\left(\frac{1}{\varepsilon D}\right) \quad .$$

which simplifies to the claimed bound. ■

We could now generalize the above results to arbitrary distributions with finite variance using queuing theory. However, we choose the simpler and more powerful model of *adversarial queuing theory* [29], [30]. This model makes no hard to justify assumptions like independence and is very simple: Any sequence of event arrivals is allowed, as long as within any time window of extent W, at most $\lfloor WD/(1+\varepsilon)\rfloor$ requests arrive.[7] In other words, the request stream is macroscopically smooth but allows arbitrary fluctuations within a time window. The scheduling algorithm works online, i.e., it has no information about request arrivals in the future.

Theorem 2: A server with D disks can service a stream of requests controlled by an adversary with rate $D/(1+\varepsilon)$ and window extent W with average delay at most $\bar{L} + W \cdot (1+\varepsilon)$ if $\bar{L}$ is the average delay achieved by a system with periodic requests arrivals with interarrival time $(1+\varepsilon)/D$.

Proof: We use the same basic approach as in the proof of Theorem 1. Consider a leaky bucket queue with service time $(1+\varepsilon)/D$. This queue can never get longer than DW and hence the additional delay is it most $DW \cdot (1+\varepsilon)/D = W \cdot (1+\varepsilon)$. ■

V. SIMULATIONS

This section studies the performance of a number of scheduling algorithms and allocation strategies (RDA if not otherwise stated). The starting point are periodic arrival times and constant service times. This simple model has the advantage that only two parameters – the number of disks D and the interarrival time $(1+\varepsilon)/D$ – need to be considered to analyze performance. In such a situation, simulations can partially replace analytical results. The expected performance can be found quickly and accurately if the random number generator works. Even the tails of the distribution can be well approximated since the system is so simple that many events can be simulated. Section V-A studies different interarrival times for fixed $D = 64$ and also discusses the running time of the matching heuristics. Section V-B argues that D has little influence on the behavior of the system. In Section V-C it is demonstrated that a failed disk has little influence on performance. Sections V-D and V-E generalize the load model to traffic patterns that cannot be modeled by open queuing systems and are typical for applications that saturate the system until some resource is exhausted. Section V-F considers the case of varying, unpredictable service times. For this case, additional algorithms make sense and the performance differences change.

The following algorithms are in the main focus:

Nonredundant: Accesses to random nonredundantly stored blocks. In our simple system there is nothing to schedule here.

Mirror: A block stored on disk i is also stored on disk $D-i-1$. Scheduling is done using the shortest queue heuristics. Other Algorithms cannot do better in this case.

Shortest queue: Scheduling using the shortest queue heuristics from Section II-A.

Lazy Sharing: The refined lazy algorithm described in Section II-B which tries to reduce average and maximal delays.

Matching: The fast implementation of the algorithm based on bipartite matching from Section III-A; using approximate shortest path search with bucket width 0.1 for paths with weight exceeding $|R|/D$ where $|R|$ is the number of requests currently in the system. We additionally reorder the slots by FIFO order and use the stealing heuristics.

Not shown here or only discussed in places where something interesting can be noted are the following further algorithms and allocation strategies that we now quickly survey roughly in order of increasing merit:

Nonrandom worst case: All requests are directed to the same disk. In the worst case all copies are on the same disk hence the system will be extremely inefficient regardless of the scheduling strategy. Even if two copies go to different disks, all the work might have to be done by two disks.

Ring allocation: A block stored on disk i is also stored on disk $i+1 \bmod D$ where i is chosen randomly [31], [32]. Aerts et al. [32] propose this scheme because they can give an efficient algorithm for finding optimal schedules for batches of requests. Although this scheme is better than plain mirroring, it is significantly worse than general RDA even if we compare optimal scheduling for ring allocation with the very fast and simple shortest queue algorithm for RDA.

Shortest queue with stealing: As shortest queue but using the stealing refinement from Section II-C, i.e.,

[7] We need the additional technical assumption that the random locations chosen for block allocations are independent of request sequence, i.e., we assume an *oblivious* adversary.

disks without scheduled request steal the first incident edge in FIFO order. This refinements only makes a difference for unpredictable service times and hence is only discussed in Section V-F.

Lazy Queue: The basic algorithm from Section II-B which always schedules the first incident edge in FIFO order. Since this algorithm is equivalent to shortest queue for predictable service times, we also only discuss it in Section V-F.

Local Search Hybrid: A combination of the shortest queue heuristic with several local optimization heuristics as described in Section II-C. Some of these algorithms are able to achieve an improvement over shortest queue but so far they are beaten by the simpler and more elegant lazy sharing algorithm.

Flow: We have implemented an asynchronous adaptation of the maximum flow based algorithm from [2] mentioned in Section I-C. For not too small ε it achieves better performance than shortest queue in particular with respect to average delays. However, considering that this algorithm is complicated and computationally intensive, it is disappointing.

A. The Influence of Arrival Rates

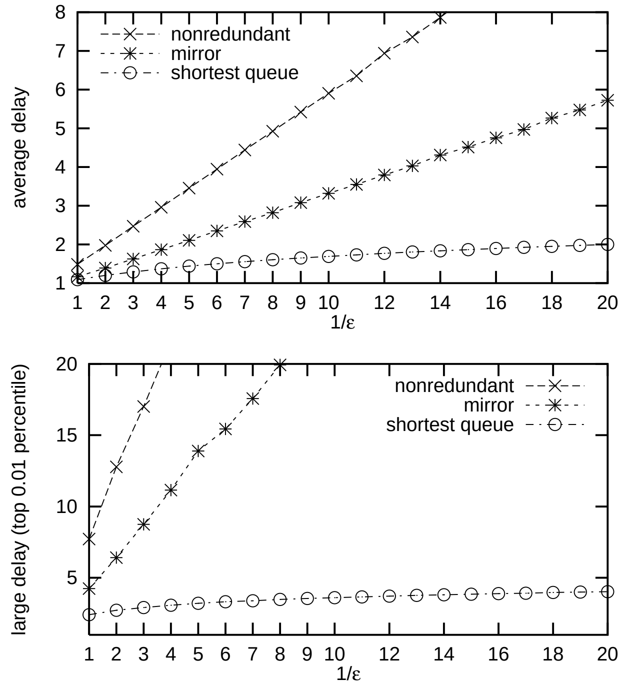

Fig. 4. Comparison of the shortest queue heuristics with randomized algorithms which do not use RDA. Average delay and large delays (occurring once in 10000 requests) for 64 disks and 10^7 requests with interarrival time $(1+\varepsilon)/D$ as a function of $1/\varepsilon$.

Figure 4 shows the average delay and large delays for the shortest queue algorithm and those randomized algorithms which do not use RDA. Nonredundant allocation has average delay linear in $1/\varepsilon$. This can be proven using queuing theory. Mirroring halves the delays. The other algorithms show that once redundancy is allowed we can do much better than mirroring in particular for arrival rates which use as much as 90% of the peak performance of the system. With respect to large delays, the difference between nonredundant placement or mirroring and RDA is even more dramatic. This phenomenon is theoretically well understood. We stress it here because nonredundant allocation is still predominant in practice and a lot of application specific load balancing strategies might become dispensable by using RDA.

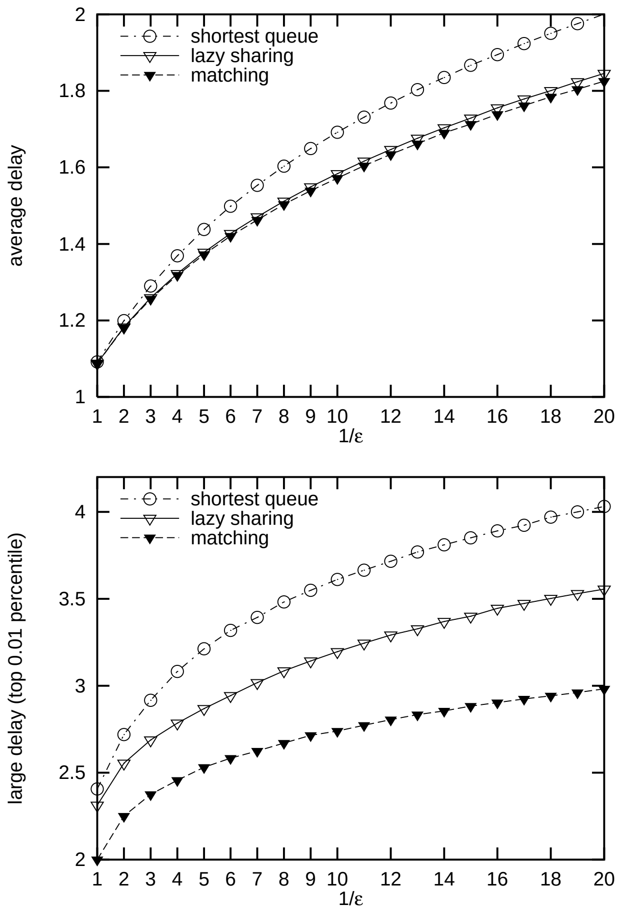

Fig. 5. Comparison of new scheduling algorithms with the shortest queue heuristics.

The performance differences between nonredundant allocation and RDA are so large that we decided to compare the RDA algorithms separately in Figure 5 allowing a better resolution. Both in terms of average and large delays there is a consistent ranking of achieved scheduling quality, matching > lazy sharing > shortest queue. Lazy sharing almost reaches matching for average delays and lies somewhere in the middle with respect to large delays.

The additional time needed for computing matchings was 7.3μs per request using an efficient but not highly tuned implementation. The system used was GNU C++ on a 300MHz Ultra-SparcIIi processor. For $D = 256$, scheduling time per request grows to about 9.6μs per request, i.e., apparently the growth is sublinear in D. Even the fastest disks currently have access latencies around 5ms so that on a 64 disk system the scheduling processor would only spend 10% of its capacity on scheduling. Fast interconnection networks with low latency communication libraries achieve latencies small enough to become no bottleneck. So currently a matching based centralized scheduler should be quite feasible for system with hundreds of disks. When the gap between processing speed and disk latency widens further, even larger systems become feasible.

B. Scaling D

The asymptotic analysis of the shortest queue algorithm [16], [17] already hints that D might have little influence on the distribution of request latencies. Figure 6 shows the dependence of average and large delays for fixed ε and varying D. Considering the large variation in D, the changes are small. The D-dependence of average delays is much smaller than observed in the simulations of [17]. The main difference[8] in the setup is that we simulate request streams which are 100 times longer. One can observe that for large D, short sequences with 100000 request are not long enough to approximate the equilibrium state of the system so that by increasing D the average delay appears to decrease. This effect disappears for longer request sequences.

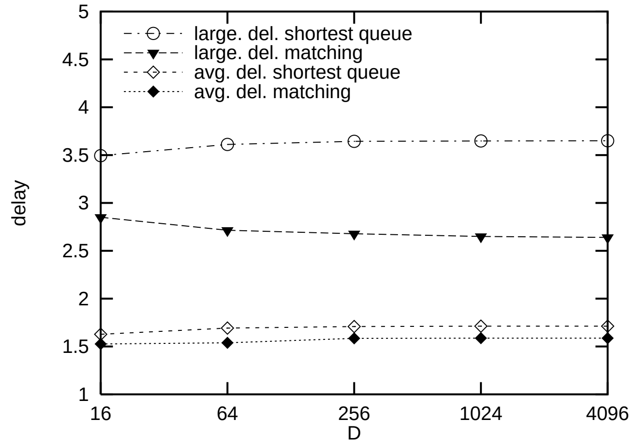

Fig. 6. Scheduling quality of the matching and shortest queue algorithm for $\varepsilon = 0.1$ and varying D.

[8] To exclude the differences for the values of D and ε used we have also made simulations with the values from [17] ($D \in \{100, 500\}$ and $\varepsilon = 0.01$).

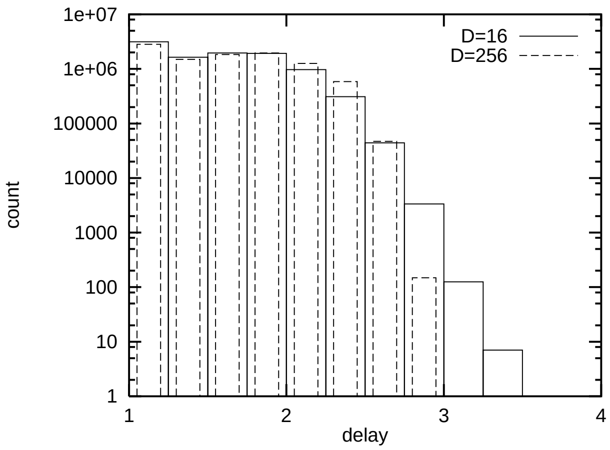

Fig. 7. Delay distribution the matching algorithm, $\varepsilon = 0.1$, and $D \in \{16, 256\}$.

The most interesting effect is that large delays slightly decrease with D for the matching algorithm whereas they increase for the shortest queue heuristics. The best explanation for the decrease in the case of the matching algorithm is the common phenomenon of sharp concentration in probabilistics that gets more pronounced as the system size increases. This effect can be demonstrated in more detail in the histograms shown in Figure 7. A small system with 16 disks has slightly longer tails in the distribution of delays than a large system with 256 disks.

We can also see that the matching algorithm produces very short tails in both cases. This is important for some real time time applications which require very low failure probabilities. Even in the 16 disk system, the probability that the delay exceeds 3.25 is less than 10^{-6}. Much lower probabilities for slightly larger delays are likely but would require very long simulations. However, once delays due to random placement are so unlikely, other bottlenecks like temporary failures of disks may become the dominating factor anyway.

C. Faults

When a disk fails, the system throughput decreases by a small factor of $1/D$. In addition, requests which have a copy on the faulty disks lose their scheduling flexibility. Since only few requests are affected, load balancing still works well. Measurements not shown here demonstrate that the delays increase only slightly. The advantage of the matching based algorithm over shortest queue is somewhat increased.

D. Beyond Open System Traffic

The periodic request streams from the previous sections and the queuing model from Section IV assume a so called *open system* where request arrivals are independent of the behavior of the server. For many applications this assumption is not warranted and can lead to wrong predictions. When we analyze the performance of disk systems, we are particularly interested in I/O limited applications that are most of the time waiting for requests to be fulfilled. In this case, answering a request triggers new request. Thus, request arrivals depend on service times and result in very bursty traffic if the system is flooded with requests until some resource like buffer space is used up.

To explore how scheduling algorithms perform on resource limited traffic, we report simulations based on a simple traffic pattern that produces all these effects and actually appears in applications based on sorting or offline prefetching [33], [34]. Note that sorting is the most I/O intensive part of many data base operations like index construction or certain joins.

Given m block buffers and a sequence of requests S to be consumed by the application in that order. The application posts the next request from S whenever a buffer is available, i.e., initially the buffers are flooded with a burst of m requests. When a block is made available to the application, it is not immediately consumed but buffered until all the blocks preceding it in S have been consumed.

In such a situation, we are not so much interested in delays of individual blocks but in the overall *efficiency* of the system, i.e., the ratio between the data volume $|S|$ actually accessed and the volume $D \cdot T$ that could be transferred in a total execution time T. Similar to our use of ε to describe arrival rates close to the system limit, we now also use the term *inefficiency*$= 1 -$efficiency that should be small to be close to a perfect system.

Figure 8 shows that RDA and in particular the matching algorithm are useful in this situation. Even with a small buffer pool of size $3D$, the matching algorithm achieves an efficiency of more than 99 %.

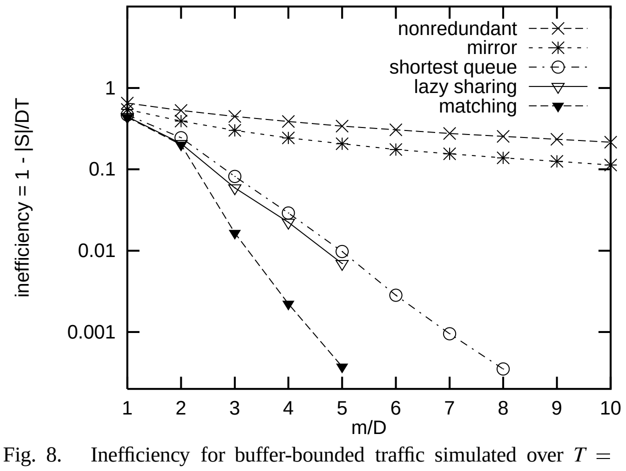

Fig. 8. Inefficiency for buffer-bounded traffic simulated over $T =$ 100000 time units.

E. Mixed Traffic

We have seen that RDA achieves both low delays for continuous request arrivals and high efficiency for traffic with bounded prefetching buffer. But delays can get rather large in the latter case. Therefore, we seemingly face a dilemma if both kinds of traffic are mixed. The dense traffic produced by buffer limited applications delays continuous (e.g. periodic) request arrivals that may stem from a delay sensitive application. Figure 9 shows that we get much larger latencies than in a system without buffer limited traffic (compare Figure 5). Not shown here are the average delays which are affected similarly. The observed delays are well approximated by $\frac{m}{D(1-x)}$ where x is the fraction of the system resources used up by periodic requests (i.e. interarrival time is D/x). Intuitively, this can be explained by observing that the bounded buffer traffic keeps about m requests in the system. Periodic traffic contributes a number of requests that is proportional to its share in system load. Hence, there are usually about $m/(1-x)$ requests in the system. Hence, it is not astonishing that delays are of this magnitude divided by D.

This dilemma can be solved by giving priority to delay sensitive requests. The implementation used in our simulations achieves this by a quite simple measure. All our scheduling algorithms base their decisions on *delays* that have so far been computed as anticipated delivery time minus release time. Now we simply replace release time by an *anticipated finishing time*. For continuous request arrivals we set this time to one plus release time whereas for buffer limited traffic with buffer size m we set it to release time plus m/D. The bottom curves in Figure 9 show that with this measure delays of periodic requests are small and almost independent of the buffer size of the

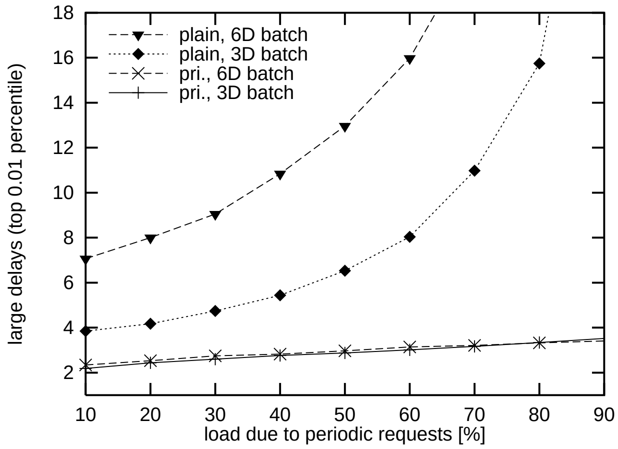

Fig. 9. Large delays (occurring once in 10000 request) suffered by periodic requests if they have to compete with buffer limited traffic. If all requests are treated equal (plain), delays are quite big. With priorization, delays are comparable to a system with pure real time traffic. All curves show the performance for the matching algorithm.

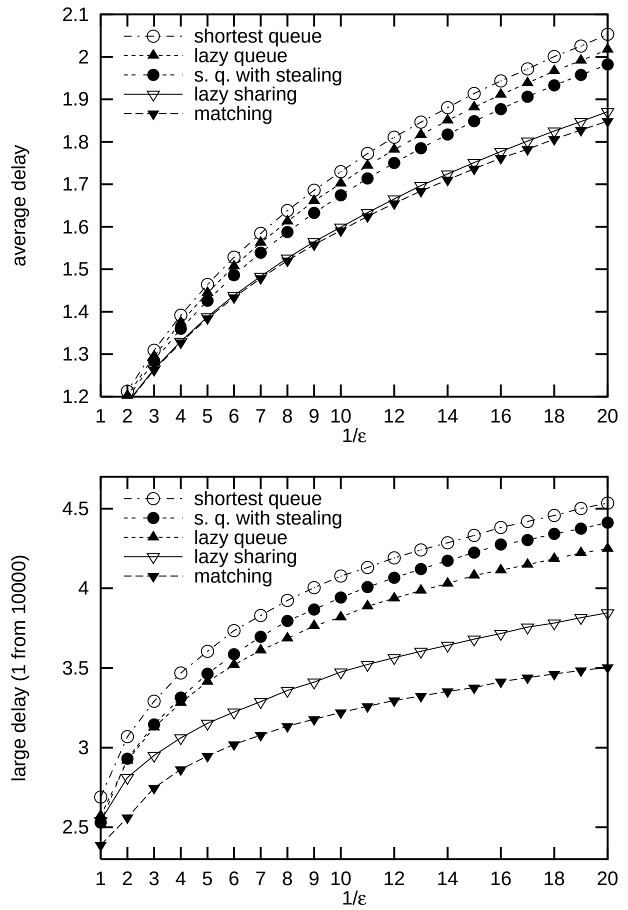

Fig. 10. Comparison of five scheduling algorithms for uniformly distributed service times between 0.7 and 1.3. Average delay and large delays delay (occurring once in 10000 request) for 64 disks and 10^7 requests with interarrival time $(1+\varepsilon)/D$ as a function of $1/\varepsilon$.

bursty traffic.

Further measurements not shown here indicate that the price we have to pay for priorizing periodic traffic over buffer limited traffic in terms of efficiency is rather low except for very small buffer sizes.

F. Unpredictable Service Times

In contrast to fluctuations in event arrival, unpredictable service times can have a significant impact on performance. Therefore, we decided to perform a number of simulations. Also refer to Section VII for several refinements of the model. To keep the number of parameters of the model small, random service times are used that have an identical distribution for all requests. From the point of view of queuing theory, an exponential distribution would be the easiest choice. However, this would give an artificial advantage to algorithms which ignore the information when a request started to execute on a disk (the exponential distribution is memoryless, i.e., the expected remaining service time is always the same). Santos and Muntz [15] find that a quite narrow normal distribution works well in certain situations. Since this might be too optimistic when rotational delays have a bimodal distribution due to “near misses”, we conservatively choose a rather wide uniform distribution with mean one.

Figure 10 shows the results using the same parameters as in Figure 5 except that the service times are now uniformly distributed between 0.7 and 1.3 and are unknown to the scheduler. Now the lazy queue algorithm and the shortest queue algorithm with stealing become different from the plain shortest queue algorithm. As expected, both perform better than shortest queue. Interestingly, there is no clear winner between these two algorithms. Stealing has smaller average delays and the lazy algorithm is better with respect to large delays. Although the matching algorithm is based on an oversimplification of the system model, it still performs best. However, the lazy sharing algorithm comes even closer to the performance of the matching algorithm now.

VI. Beyond Replication Coding

Instead of simply replicating logical blocks, we can more generally encode a logical block which has r times the size of a physical block into w subblocks of unit physical size such that reading any r out of the w subblocks suffices to reconstruct the logical subblock. Perhaps the most important case is $w = r+1$. Using *parity-encoding*, r of the subblocks are simply pieces of the logical block and the last subblock is the exclusive-or of the other subblocks. A missing subblock can then be reconstructed by taking the exclusive-or of the subblocks read. Parity encoding is the easiest way to reduce redundancy compared to RDA while maintaining some flexibility in scheduling. Its main drawback is that the physical blocks being read are a factor r smaller than the logical blocks so that high bandwidth can only be expected if the logical blocks are fairly large. As long as $r \ll D$, even a disk failure will not severely inhibit performance since only a fraction of $(r+1)/D$ of the requests will come without scheduling flexibility.

Choosing $w > r+1$ can be useful if more than one disk failure is to be tolerated, if a single disk failure should inhibit performance even less or if insufficient battery buffered RAM for buffered writing is available. In the latter case, one writes less than w subblocks and uses the additional flexibility for explicitly scheduling write accesses using the same methods as described here for reading. A disadvantage of codes with $w > r+1$ is that they are computationally more expensive than parity-encoding [35], [36], [37], [38], [39], [14].

It makes sense to use different values for r and w concurrently on the same system for different purposes. Data which is rarely read and can be reconstructed otherwise, e.g., checkpointing data for scientific computing, could be stored nonredundantly. Frequently accessed data, in particular if read with small block sizes should be replicated, perhaps even more than twice (i.e., $r = 1$, $w > 2$). High volume data which is read in larger chunks could use parity encoding with rather large r, etc. All the scheduling algorithms described here can work with such a mixed workload.

A. Simple Algorithms

The simple scheduling algorithms from Section II are straightforward to adapt. The allocation graph G_A now becomes a hypergraph. Hyperedges connect w nodes and are marked with the number r of edges to be read. For the schedule graph G_S we can use a generalized notion of directed hyperedge which points to $w-r$ of the connected nodes. For parity encoding, where $w-r=1$, even the notion of a directed path of nodes makes sense. It still suffices to consider FIFO schedules and the lazy queue algorithm is still equivalent to an omniscient shortest queue algorithm.

Theoretical results for the case $r > 1$ are quite sparse and so far limited to synchronous algorithms (e.g., the optimal batched scheduling algorithm from [2] works for $w = r+1$). The *delay tree technique* originally developed to analyze *collision protocols* used in PRAM emulation works for general r and w. So it might be possible to apply delay trees to asynchronous algorithms.

We have applied the asymptotic approach from [16], [17] for modeling the behavior of the shortest queue algorithm to general r and w. This yields a recurrence relation for the queue length distribution. However, this relation seems to have no closed form solution for $r > 1$ so that it is mostly useful for numeric evaluation so far.

B. A Constrained Bipartite Matching Formulation

We now generalize the matching algorithm. A schedule can be represented as an R-perfect matching in a bipartite graph $G_2 = (R \dot\cup S, C)$ where S represents time slots as before and where $(e,i) \in R$ represents the i-th *subrequest* ($i \in \{1,\ldots,r\}$) of request e. All subrequests of a request are connected to the same set of slots on w disks. Unfortunately, only those matchings represent *legal* schedules which match all subrequests of a request to slots of different disks. Therefore, the shortest path search described in Section III-A is modified to consider only augmenting paths which maintain this property. We constrain the search for augmenting paths to *legal* edges connecting R and S where $\{(e,j),(d,i)\}$ is legal with respect to the current matching M if $\neg\exists i', j' : \{(e,j'),(d,i')\} \in M$. The following Theorem shows that this algorithm yields schedules that minimize maximal delays.

Theorem 3: Incremental construction of matchings using only legal augmenting paths leads to legal R-perfect matchings whenever legal R-perfect matchings exist.

Proof: Consider a legal matching M which is not R-perfect and an unmatched node $v = (e,j) \in R$. If a legal R-perfect matching M^* exists, we construct a legal augmenting path which leads from v to a free node in S.

Since M^* is R-perfect there is exactly one edge in M^* incident to any node in R. Consider the edge $e = \{v,(d,i)\} \in M^*$. If e is legal we start the path with e. If e is illegal, by definition of illegal edges, there is an edge $\{(e,j'),(d,i')\} \in M$ with $i' \neq i$. Now consider the edge $\{(e,j'),(d_2,i_2)\} \in M^*$. Since all subrequests of e are adjacent to the same set of slots, there must be an edge $\{v,(d_2,i_2)\}$. If this edge is legal, we start our augmenting path with it. Otherwise, we go on following edges in M and M^* until a slot (d'',i'') is found such that $\{v,(d'',i'')\}$ is legal. This must eventually happen: Since M^* is a legal R-perfect matching, all subrequests e are matched by M^* to slots on different disks. But since v is unmatched in M, not all these slots can be illegal for a connection with v. Figure 11 gives an example.

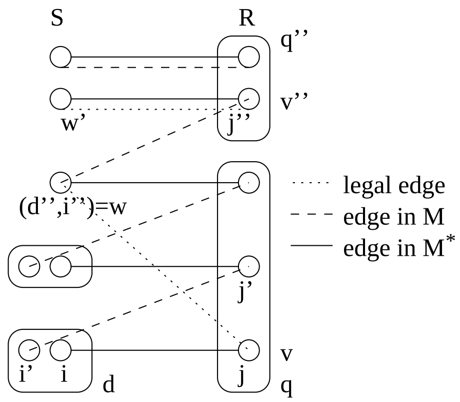

Fig. 11. Example how to find legal edges for an augmenting path.

When an augmenting path under construction has reached a node $w \in S$ (e.g., after the first step, $w = (d'',i'')$) and w is free, the augmenting path is complete. Otherwise, the path is extended using the edge $\{w, v'' = (e'',j'')\} \in M$. Then, a legal edge $\{v'',w'\}$ is found using the same techniques as before. Now the reason why such an edge must exist is that the subrequest to which w is matched in M^* is not a subrequest of e''. Therefore, $\exists s \in S, k : \{s,(e'',k)\} \in M^* \wedge \forall k' : \{s,(e'',k')\} \notin M$.

This construction is continued until a free node is reached. This must eventually happen since we can never get back to v which is unmatched in M. ■

VII. α: Automatically Load-Balanced Parallel Hard-Disk Arrays

We believe that the load balancing algorithms introduced so far could be an ingredient to build easy to use

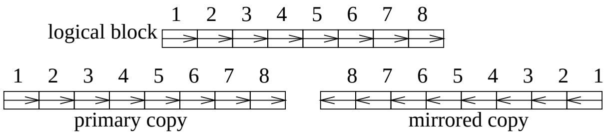

Fig. 12. RDA using a mirrored copy can be used to read a large block from both copies in an adaptive way.

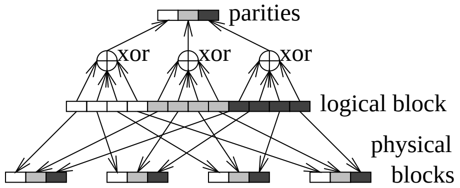

Fig. 13. Encoding of a logical block of size 12 into 4 physical blocks and one parity block of size 3 such that aligned logical requests of size $4s$, $s \in \{1,2,3\}$ can be fulfilled by retrieving any 4 out of 5 physical subblocks of size s.

α-system which can be used as if a single very high performance disk were available. However, to get a practical general purpose system, some refinements are necessary and the new algorithms need to be reconciled with some established technologies. In the following we sketch how this could be done. Section VII-A discusses how irregularity in the request size and the service time of the disks can be accommodated. Interactions between the global scheduling of parallel disks used so far and well studied scheduling algorithms for minimizing seek times are added in Section VII-B. In Section VII-C we outline how redundancy can be exploited to get fault tolerance at high efficiency. Section VII-D explains how to cope with different disks in the system. Finally, Section VII-E investigates priorization and fairness issues. Complementary to these informal discussion for asynchronous systems there is a paper [22] that discusses similar issues for the less practicable but analytically more tractable case of synchronous batched accesses.

A. Variable Request Sizes and Multizone Disks

In a general purpose system not all requests are to logical blocks of the same size. In addition, even for applications where this is the case, the actual access times will depend on the track where the physical data is stored — more data sectors fit on the outer zones of a hard disk so that at constant angular velocity the bandwidth changes.

The simple scheduling algorithms from Section II can be adapted to such a scenario relatively easily, the lazy queue algorithm does not even need an accurate model of the disks to make good decisions.

Finding substantially better or even optimal schedules is a difficult problem however. For example, even for $D = 2$, random allocation, and identical release times for all requests, finding a currently optimal schedule without allowing preemption is NP-hard. This problem is equivalent to the partition problem [40].

Since many hard disks do not allow to preempt a request once started we seem to be stuck. However, a good compromise is to split all requests into subrequests of estimated equal access time. Then we can use algorithms based on bipartite matchings again. The overhead involved is not as large as one might expect. Since modern disks use caching and read ahead, accessing a large block in k pieces costs about as much as accessing the whole block plus k times the overhead for initiating a transmission between the on-disk-cache and the server. This overhead is small compared to the mechanical delays of the disk.

If we use FIFO schedules, scheduling a request of size s located on disks u and v results in accessing k subblocks on u and $s-k$ subblocks on v without interruption by other accesses. In addition, we can always reschedule the subblocks in such a way that the subblocks on u and v are consecutive without affecting the delay of other requests. This approach works particularly well if we take the term "mirroring" literally, i.e., we store the second copy of a block in the reverse physical order. This way, both disks start accessing the data from both logical ends and we stop when all the data is present. Figure 12 gives an example. Summing up, we pay an additional seek delay and some overhead for blocking and get improved load balancing, and the opportunity to use highly optimized scheduling algorithms. For large requests, this looks like a promising option. If we deviate from FIFO scheduling it is additionally possible to guarantee short latencies for small accesses which would be impossible if small requests would always have to wait for large requests with earlier release time. Refer to Section VII-E for more details.

We also have to discuss how data should be allocated to the disks to allow both fine grained and coarse grained access to the same file. A good compromise is to use large physical blocks, e.g., currently around 1MB [41] to allow high bandwidth access with large requests. Later we are free to read smaller blocks. For general r out of w encoding this requires some further explanations. The obvious way — chopping a large physical block into r contiguous subblocks — does not work. Instead, we chop it into small pieces, e.g., 512 byte sectors or even machine words, and code groups of r pieces into w pieces. Figure 13 gives an example. Now small requests can be serviced with the same scheduling flexibility as large requests.

B. Scan Scheduling

To achieve high throughput in the case of high system load one should deviate from the strict FIFO discipline to reduce the overall seek time (e.g., [42]). Sorting the requests by track number is usually a good approach for a single disk if we do not care about large delays for some requests. Good compromises between low delays and high performance are an active area of research even for single disk.

Now we have the additional complication of parallel disks. For example, service times now get dependent on the presence of other requests. So far, we only have partial answers to this problem. First, the shortest queue algorithm reduces the overall scheduling problem to single disk scan scheduling. We keep a current schedule and an estimate of its cost for each disk and use this estimate to commit a newly arriving request.

The other simple algorithms from Section II need only slightly more general information. We must be able to decide how the schedule changes if the disk retrieving one particular request is changed.

The matching algorithm from Section III can be adapted similarly. For extending an augmenting path we only need to know how the schedule of a particular disk changes if one request is replaced by another. Although we loose optimality guarantees by these changes, the simulations from Section V-F indicate that even if the the load model is inaccurate, the matching based scheduler is good at reducing large delays.

The above considerations give some evidence that the ranking of the scheduling algorithms determined for the simple model used in this paper could remain the same for a more realistic model if the algorithms are augmented with an appropriate scan scheduling heuristics.

C. Fault Tolerance

An α-system can be implemented without a single point of failure. We use a system with at least w independent processors. Data allocation is done in such a way that the subblocks of a request are allocated on disks attached to different processors. A write request may be confirmed as soon as all subblocks have been stored in the write buffers. The power supply has a battery backup providing enough energy to flush all write buffers. The processors need to be connected by a fast redundant network.[9] The centralized scheduler needed for the matching based algorithms is no problem if all PEs are regularly informed about their local schedule. When the processor responsible for scheduling fails, a new processor takes over this task and reconstructs the current schedule using the local schedules of the non-faulty processors.

Using *virtual spares*, even with r out of $r+1$ coding, we can tolerate multiple disk failures as long as between disk failures there is enough time to remap the data of the failed disk. Appropriate hash functions for this purpose are described in [14]. Except for the bandwidth of the missing disks, the system performance is uninhibited once the reconfiguration is complete. This way there is no need to exchange failed disks immediately so that expensive 24h service contracts may become cheaper or avoidable. For many applications one will not need hot swapable disks any more since failed disks only need to be replaced in the regular service intervals if a sufficient number of disks is still intact.

[9] If cost is more important than performance in the case of a fault, an asymmetric configuration with one fast network and a cheap slow network may be used.

D. Inhomogeneous Systems

Within the lifespan of a parallel disk server, hard disk technology usually changes so much that the disks originally bought are likely to be outdated after some time. It may be very difficult or expensive to buy the same type of disks when faulty disks are to be replaced or when the system is upgraded. Therefore, it is better to upgrade the server with the disks which are currently most economical. In a traditional system without automatic load balancing, it would become quite complicated to avoid the slowest disks to become a bottleneck. With load balancing, speed imbalances are equalized as much as possible automatically. The lazy queue algorithm does not even need to know the speed of the disks. The other algorithms need information about capacity and speed for the different zones of the new disk. This data can be found automatically by running a small benchmark when a new disk is configured into the system.

Things get more complicated when the new disks also have larger capacity. As long as the speed improvement of a new disk is halfway proportionate to the capacity improvement, we can modify the hash function to map more data to this disk. It is less advisable to use disks where capacity grows much faster than bandwidth (this easily happens if one insists on always buying the largest disks available). In this case, it would be easiest to use some of the space for storing files which are rarely accessed.

E. Priorities and Different Application Types

Section V-E already gave an example how the needs of different applications and application types can be coordinated by some form of priorization. For a practical system is is important to come up with schemes that do not need much hand tuning or knowledge of many system and application parameters. For example, it is probably feasible to assume that real time requests are recognizable and that an admission control policy outside the server makes sure that real time requests alone can never overload the system. But it would be quite complicated to mark requests with such application specific parameters as the buffer size defined in Section V-D. We expect that this is not necessary either. For example, the system could monitor the load stemming from different sources of requests and compute the "modified delays" needed in Section V-E based on these statistics.

VIII. CONCLUSION

The techniques developed in this paper form an algorithmic toolbox for running parallel disk systems that achieve high performance and are easy to use. The combination of random placement via hashing and redundant storage provides us with a general data placement strategy that works well for arbitrary access patterns. Random duplicate allocation (RDA) greatly outperforms nonredundant placement and even mirroring that has the same storage cost. The new scheduling algorithms lazy sharing and matching improve on the previously known shortest queue algorithm for RDA. Lazy sharing is relatively simple and easy to implement in a parallel system. The matching algorithm gives a further reduction of large delays because it minimizes maximal delays in some precise sense. Although matching needs a centralized scheduler, hundreds of disks can by served by our matching algorithm that uses an implicit graph representation and fast incremental computation of augmenting paths.

Although Sections IV–VII discuss rather different topics they serve the common purpose to substantiate the usefulness of the general approach (random redundant placement and clever asynchronous scheduling algorithms). Section IV proves that request streams with fluctuating arrival rates are not much different from streams with periodic arrivals. Hence, the the space of parameters in our model is further reduced. Only the two scalar parameters system size and arrival rate have to be varied. This reduction in parameter space makes it possible to cover this space fairly well by the simulations given in Section V. Section V-D further generalizes the set of system inputs to a natural class of applications that cannot be modeled by any open system. Section V-E closes a gap between the above two models by explaining how both types of traffic can coexist efficiently.

The following Sections VI and VII generalize the model to cover a wider range of practical situations. The limitation to duplicate storage is lifted in Section VI by explaining how the scheduling algorithms can be generalized to allow lower redundancy, more fault tolerance, or both. The generalized matching algorithm might be of independent theoretical interest since it solves a problem in polynomial time that is more general than matching. The generalized matching algorithm is even useful for the special case of batched arrivals since the batched scheduling algorithm from [2] does not work for $w \geq r+2$.

Sections VII introduces further generalizations of the algorithms and allocation strategies to cope with variable access times, seek times, inhomogeneous systems and different application types. Sections VI and VII stay on the algorithmic level since the larger space of possible parameters makes it much more difficult to cover this space by simulations. Perhaps future work should directly make the step to implementing the algorithms on an existing system (real or simulated) and measure the performance for existing application benchmarks like SPEC SFS[10] or the TPC benchmarks [11].

Acknowledgements: Many thanks to Petra Berenbrink, Andre Brinkmann, Sebastian Egner, Jan Korst, Panagiota Fatourou, Christian Scheideler, Roberto Solis Oba, Kay Salzwedel, Jeff Vitter, Berthold Vöcking and Gerhard Weikum for very helpful discussions on the topic.

REFERENCES

[1] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[2] P. Sanders, S. Egner, and J. Korst, "Fast concurrent access to parallel disks," in *11th ACM-SIAM Symposium on Discrete Algorithms*, 2000, pp. 849–858.

[3] K. Salem and H. Garcia-Molina, "Disk striping," *Proceedings of Data Engineering'86*, 1986.

[4] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *Proceedings of ACM SIGMOD'88*, 1988.

[5] C. Armen, "Bounds on the separation of two parallel disk models," in *IOPADS*, Philadelphia, May 1996, pp. 122–127, ACM Press.

[6] R. D. Barve, E. F. Grove, and J. S. Vitter, "Simple randomized mergesort on parallel disks," *Parallel Computing*, vol. 23, no. 4, pp. 601–631, 1997.

[7] E. L. Miller and R. H. Katz, "RAMA: An easy-to-use, high-performance parallel file system," *Parallel Computing*, vol. 23, pp. 419–446, 1997.

[8] R. M. Karp, M. Luby, and F. Meyer auf der Heide, "Efficient PRAM simulation on a distributed memory machine," in *24th ACM Symp. on Theory of Computing*, May 1992, pp. 318–326.

[9] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," in *26th ACM Symposium on the Theory of Computing*, 1994, pp. 593–602.

[10] W. Tetzlaff and R. Flynn, "Block allocation in video servers for availability and throughput," *Proceedings Multimedia Computing and Networking*, 1996.

[11] R. Tewari, R. Mukherjee, D.M. Dias, and H.M. Vin, "Design and performance tradeoffs in clustered video servers," *Proceedings of the International Conference on Multimedia Computing and Systems*, pp. 144–150, 1996.

[12] J. Korst, "Random duplicate assignment: An alternative to striping in video servers," in *ACM Multimedia*, Seattle, 1997, pp. 219–226.

[13] R. Muntz, J.R. Santos, and S. Berson, "A parallel disk storage system for real-time multimedia applications," *International Journal of Intelligent Systems*, vol. 13, pp. 1137–1174, 1998.

[14] P. Berenbrink, A. Brinkmann, and C. Scheideler, "Design of the PRESTO multimedia storage network," in *International Workshop on Communication and Data Management in Large Networks*, Paderborn, Germany, October 5 1999.

[15] J. R. Santos, R. R. Muntz, and B. Ribeiro-Neto, "Comparing random data allocation and data striping in multimedia servers," in *ACM SIGMETRICS*, 2000, to appear.

[16] N. D. Vvedenskaya, R. L. Dobrushin, and F. I. Karpelevich, "Queueing system with selection of the shortest of two queues: An asymptotic approach," *Problems of Information Transmission*, vol. 32, no. 1, pp. 15–29, 1996.

[17] M. Mitzenmacher, *The Power of Two Choices in Randomized Load Balancing*, Ph.D. thesis, UC Berkeley, 1996.

[10] `http://www.spec.org`
[11] `http://www.tpc.org/`

[18] R. Cole, A. Frieze, B. M. Maggs, M. Mitzenmacher, A. W. Richa, R. K. Sitaraman, and E. Upfal, "On balls and bins with deletions," in *2nd RANDOM*. 1998, vol. 1518 of *LNCS*, pp. 145–158, Springer.

[19] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking, "Balanced allocations: The heavily loaded case," in *32th Annual ACM Symposium on Theory of Computing*, 2000.

[20] M. Adler, P. Berenbrink, and K. Schröder, "Analyzing an infinite parallel job allocation process," in *6th European Symposium on Algorithms*. 1998, number 1461 in LNCS, pp. 417–428, Springer.

[21] P. Berenbrink, A. Czumaj, T. Friedetzky, and N. D. Vvedenskaya, "On the analysis of infinite parallel job allocation processes via differential equations," in *11th ACM Symposium on Parallel Algorithms and Architectures*, 2000.

[22] P. Sanders, "Reconciling simplicity and realism in parallel disk models," in *12th ACM-SIAM Symposium on Discrete Algorithms*, Washington DC, 2001, pp. 67–76.

[23] J. Korst and P. Coumans, "Asynchronous control of disks in video servers," Tech. Rep. NL-MS-19.649, Philips Research Laboratories, Eindhoven, the Netherlands, 1998.

[24] P. Berenbrink, M. Riedel, and C. Scheideler, "Simple competitive request scheduling strategies," in *11th ACM Symposium on Parallel Architectures and Algorithms*, 1999, pp. 33–42.

[25] R. K. Ahuja, R. L. Magnanti, and J. B. Orlin, *Network Flows*, Prentice Hall, 1993.

[26] K. Mehlhorn and S. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press, 1999.

[27] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, 3 edition, 1996.

[28] K. S. Trivedi, *Probability & Statistics with Reliability, Queueing and Computer Science Applications*, Prentice Hall, Englewood Cliffs, 1982.

[29] A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. P. Williamson, "Adversarial queing theory," in *28th Annual ACM Symposium on Theory of Computing*, 1996, pp. 376–385.

[30] W. Aiello, E. Kushilevitz, R. Ostrovsky, and A. Rosén, "Adaptiv packet routing for bursty adversarial traffic," in *30th Annual ACM Symposium on Theory of Computing*, 1998, pp. 359–368.

[31] George P. Copeland and Tom Keller, "A comparison of high-availability media recovery techniques," in *ACM SIGMOD International Conference on Management of Data*, 1989, pp. 98–109.

[32] J. Aerts, J. Korst, and S. Egner, "Random duplicate storage for load balancing in multimedia servers," *Information Processing Letters*, 2000, accepted.

[33] R. D. Barve and J. S. Vitter, "A simple and efficient parallel disk merge sort," in *11th ACM Symposium on Parallel Algorithms and Architectures*, 1999, pp. 232–241.

[34] M. Kallahalla and P. J.Varman, "Optimal read-once parallel disk scheduling," in *IOPADS*, 1999, pp. 68–77.

[35] F.J. MacWilliams and N.J.A. Sloane, *Theory of error-correcting codes*, North-Holland, 1988.

[36] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson, "Coding techniques for handling failures in large disk arrays, csd-88-477," Tech. Rep., U. C. Berkley, 1988.

[37] M. O. Rabin, "Efficient dispersal of information for security, load balancing and fault tolerance," *Journal of the ACM*, vol. 36, no. 2, pp. 335–348, 1989.

[38] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994, pp. 245–254.

[39] Guillermo A. Alvarez, Walter A. Burkhard, and Flaviu Cristian, "Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, New YOrk, June 2–4 1997, vol. 25,2 of *Computer Architecture News*, pp. 62–72, ACM Press.

[40] M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman and Company, New York, 1979.

[41] H. G. P. Bosch, *Mixed-Media File Systems*, Ph.D. thesis, University of Twente, Netherlands, 1999.

[42] G. Nerjes, P. Muth, M. Paterakis, Y. Romboyannakis, P. Triantafillou, and G. Weikum, "Incremental scheduling of mixed workloads in multimedia information servers," *Journal of Multimedia Tools and Applications*, 1999, to appear.