Algorithms for Scalable Storage Servers*

Peter Sanders

Max Planck Institut für Informatik
Saarbrücken, Germany
sanders@mpi-sb.mpg.de

Abstract. We survey a set of algorithmic techniques that make it possible to build a high performance storage server from a network of cheap components. Such a storage server offers a very simple programming model. To the clients it looks like a single very large disk that can handle many requests in parallel with minimal interference between the requests. The algorithms use randomization, redundant storage, and sophisticated scheduling strategies to achieve this goal. The focus is on algorithmic techniques and open questions. The paper summarizes several previous papers and presents a new strategy for handling heterogeneous disks.

1 Introduction

It is said that our society is an information society, i.e., efficiently storing and retrieving a vast amount of information has become a driving force of our economy and society. Most of this information is stored on hard disks — many hard disks actually. Some applications (e.g., geographical information systems, sattelite image libraries, climate simulation, particle physics) already measure their data bases in petabytes (10^{15} bytes). Currently, the largest of these applications use huge tape libraries, but hard disks can now store the same data for a similar price offering much higher performance [13]. To store such amounts of data one would need about 10 000 disks. Systems with thousands of disks have already been build and there are projects for "mid-range" systems that would scale to 12 000 disks.

This paper discusses algorithmic challenges resulting from the goal to operate large collections of hard disks in an efficient, reliable, flexible, and user-friendly way. Some of these questions are already relevant if you put four disks in your PC. But things get really interesting (also from a theoretical point of view) if we talk about up to 1024 disks in a traditional monolithic storage server (e.g. `http://www.hds.com/products/systems/9900v/`), or even heterogenous networks of workstations, servers, parallel computers, and many many disks. In this paper all of this is viewed as a storage server.

We concentrate on a simple model that already addresses the requirement of user-friendliness to a large extent. Essentially, the entire storage server is

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

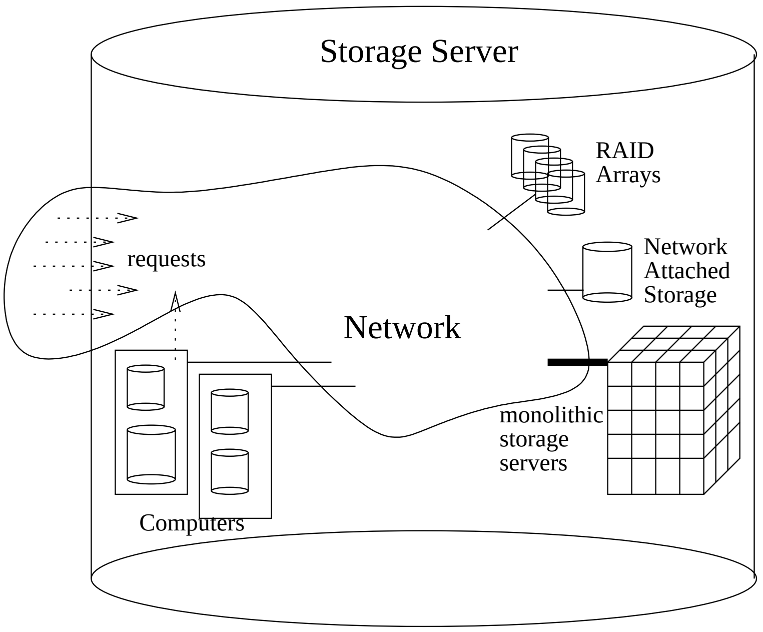

Fig. 1. A storage server appears to the outside world like a huge disk accepting many parallel request from inside and outside the system.

presented to the operating system of the computers that run the applications (*clients*) as a single very large disk (see Fig. 1): There is a single *logical* address space, i.e., an array of bytes $A[0..N-1]$.[1] N is essentially the total cumulative usable capacity of all disks. The clients can submit *requests* for bytes $A[a..b]$ to the storage server. There will be some delay (some milliseconds as in a physical disk) and then data is delivered at a high rate (currently up to 50 MByte/s from a single disk). Otherwise, the clients can behave completely naively: In particular, requests should be handled in parallel with minimal additional delays. Large requests (many megabytes) should be handled by many disks in parallel. If any single component of the system fails, no data should be lost and the effect on performance should be minimal. If the system is upgraded with additional disks, usually larger than those previously present, the logical address space should be extended accordingly and future requests should profit from the the increased cumulative data rate of the system.

The storage server can be implemented as part of the operating software of a monolithic system or as a distributed program with processes on the client computers and possibly on dedicated server machines or *network attached storage*, i.e., disks that are directly connected to the computer. All these components

[1] We use $a..b$ as a shorthand for the range $a, \dots, b$ and $A[a..b]$ stands for the subarray $\langle A[a], \dots, A[b] \rangle$.

communicate over a network. Higher level functionality such as file systems or data base systems can be implemented on top of our virtual address space much in the same way they are today build on top of a physical disk.

We will develop the algorithmic aspects of a storage server in a step by step manner giving intuitive arguments why they work but citing more specialized papers for most proofs. The basic idea is to split the logical address space A into fixed size logical *blocks* that are mapped to *random* disks. Sect. 3 explains that this is already enough to guarantee low latencies for write requests with high probability using a small write buffer. To get basic fault tolerance we need to store the data *redundantly*. Sect. 4 shows that two independently placed copies of each block suffice to also guarantee low read latency for arbitrary sets of block read requests. Sect. 5 demonstrates how we can support accesses to variable size pieces of blocks with similar performance guarantees. We are also not stuck with storing every block twice. Sect. 6 explains how more sophisticated encoding gives us control over different tradeoffs with respect to efficiency, waste of space, and fault tolerance. Up to that point we make the assumption that the clients submit batches of requests in a synchronized fashion — this allows us to give rigorous performance guarantees. In Sect. 7 we lift this assumption and allow requests to enter the storage server independently of each other. Although a theoretical treatment gets more difficult, the basic approach of random redundant allocation still works and we get simple algorithms that can be implemented in a distributed fashion. The algorithms described in Sect. 8 *use* the redundant storage when a disk or other components of the system fails. It turns out that the clients see almost nothing of the fault not even in terms of performance. Furthermore, after a very short time, the system is again in a safe state where further component failures can be tolerated. Sect. 3 assumes that a write operation can return as soon as there is enough space to keep it in RAM memory. In Sect. 9 we explain what can be done if this is not acceptable because a loss of power could erase the RAM. For simplicity of exposition we assume most of the time that the system consists of D identical disks but Sect. 10 generalizes to the case of different capacity disks that can be added incrementally.

2 Related Work

A widely used approach to storage server is RAID [27] (Redundant Arrays of Independent Disks). Different RAID levels (0–5) offer different combinations of two basic techniques: In *mirroring* (RAID Level 1), each disk has a mirror disk storing the same data. This is similar to the random duplicate allocation (RDA) introduced in Sect. 4 only that the latter stores each block independently on different disks. We will see that this leads to better performance in several respects. *Striping* (RAID Level 0) [31] is a simple and elegant way to exploit disk parallelism: Logical blocks are split into D equal sized pieces and each piece is stored on a different disk. This way, accesses to logical blocks are always balanced over all disks. This works well for small D, but for large D, we would get a huge logical block size that is problematic for applications that need fine grained access. Fault tolerance can be achieved at low cost by splitting logical blocks into $D-1$ pieces and storing the bit-wise xor of these pieces in a *parity block* (RAID Levels 3, 5).

Larger storage servers are usually operated in such a way that files or partitions are manually assigned to small subsets of disks that are operated like a RAID array. The point of view taken in this paper is that this management effort is often avoidable without a performance penalty. In applications where the space and bandwidth requirement are highly dynamic, automatic methods may even outperform the most careful manual assignment of data to disks.

Load balancing by random placement of data is a well known technique (e.g., [7, 23]). Combining random placement and redundancy has first been considered in parallel computing for PRAM emulation [18] and online load balancing [6]. For scheduling disk accesses, these techniques have been used for multimedia applications [40, 41, 19, 24, 8, 36]. The methods described here are mostly a summary of four papers [35, 32, 33, 16]. Sect. 10 describes new results.

There are many algorithms explicitly designed to work efficiently with coarse-grained block-wise access. Most use the model by Vitter and Shriver that allows identical parallel disks and a fixed block size. Vitter [42] has written a good overview article. More overviews and several introductory articles are collected in an LNCS Tutorial [22].

3 Write Buffering

3.1 Greedy Writing

Consider the implementation of an operation *write*(a, B, i) that writes a client array $a[0..B-1]$ to the logical address space $A[i..i+B-1]$. The main observation exploited in this section is that *write* can in principle be implemented to return almost immediately: Just copy the data to a buffer space.[2] The matching read operation *read*(B, i) returns the cached data without a disk access.

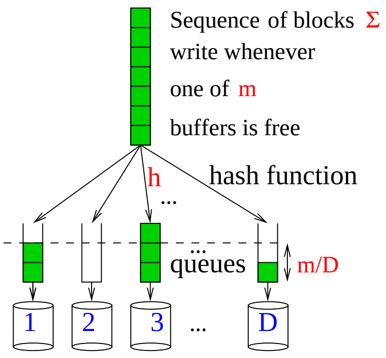

Fig. 2. Optimal Writing.

An obvious limitation of the buffering strategy is that we will eventually run out of buffer space without a good strategy for actually *outputting* the data to the disks. We postpone the question how data is mapped to the disks until Sect. 3.2 because the following *greedy writing* algorithm works for any given assignment of data to the disks. We maintain a queue of output requests for each disk. Whenever a disk falls idle, one request

[2] We can even do without a copy if we “steal” a from the client and only release it when the data is finally copied or output.

from this queue is submitted to the disk. Fig. 2 illustrates this strategy. In some sense, greedy writing is optimal:

Theorem 1 ([16]). *Consider the I/O model of Vitter and Shriver [43] (fixed block size, fixed output cost). Assume some sequence of block writes is to be performed in that logical order and at most m blocks can be buffered by the storage server. Then greedy writing minimizes the number of I/O steps needed if the disk queues are managed in a FIFO (first in first out) manner.*

Proof. (Outline) An induction proof shows that greedy writing is optimal among all output strategies that maintain queues in a FIFO manner. Another simple lemma shows that any schedule can be transformed into a FIFO schedule without increasing the I/O time or the memory requirement. ■

Things get more complicated for more realistic I/O models that take into account that I/O times depend on the time to move the disk head between the position of two blocks.

Open Problem: 1 *Can you find (approximately) optimal writing algorithms for the case that I/O costs depend on the position of blocks on the disks? Even for fixed block size*[3] *and cost estimates only dependent on seek time little is known if the buffer size is limited.*[4]

3.2 Random Allocation

Theorem 1 is a bit hollow because performance can still be very bad if all blocks we need to write have to go to the same disk. We would like to have an allocation strategy that avoids such cases. But this seems impossible — for any given mapping of the address space to the disks, there will be sets of requests that all go to the same disk. Randomization offers a way out of this dilemma. We allocate logical blocks to random disks for some fixed (large) block size B. (Sect. 10 discusses details how this mapping should actually be implemented.) Random mapping makes it very unlikely that a particular set of blocks requested by the clients reside on the same disk. More generally, we get the following performance guarantee for arbitrary sequences of *write* requests:

Theorem 2 ([35, 16]). *Consider the I/O model of Vitter and Shriver [43] (D disks, fixed block size, fixed output cost). Assume some sequence of n randomly mapped different blocks are to be written and at most m blocks can be buffered by the storage server. Then greedy writing accepts the last block after an expected number of $(1 + \mathcal{O}(D/m))\frac{n}{D}$ output steps. After the last block has been accepted, the longest queue has length $\mathcal{O}\big(\frac{m}{D}\log D\big)$.*

[3] Variable block sizes open another can of worms. One immediately gets NP-hard problems. But allowing a small amount of additional memory removes most complications in that respect.

[4] For infinite buffer size, the problem is easy if we look at seek times only (just sort by track) or rotational delays only [39]. For both types of delays together we have an NP-hard variant of the traveling salesman problem with polynomial time solutions in some special cases [5].

It can also be shown that longer execution times only happen with very small probability.

Proof. (Outline) The optimal greedy writing algorithm dominates a "throttled" algorithm where in each I/O step $(1 - D/m)D$ blocks are written. The effect of the throttled algorithm on a single disk can be analyzed using methods from queuing theory if the buffer size is unlimited. The average queue length turns out to be bounded by $m/2D$ and hence the expected sum of all queue lengths is bounded by $m/2$. More complicated arguments establish that large deviations from this sum are unlikely and hence the influence of situations where the buffer overflows is negligible. ■

Open Problem: 2 *The number of steps needed to write n blocks and flush all buffers should be a function that decreases monotonically with m. Prove such a monotonic bound that is at least as good as Theorem 1 both for large and small n.*

3.3 Distributed Implementation

We now explain how the abstract algorithms described above can be implemented in a real system. We will assume that one or several disks are connected to a computer where one ore several processors share a memory. Several computers are connected by a communication network to form the storage server. Disks directly attached to the network are viewed as small computers with a single disk attached to them. The client applications either run on the same system or send requests to the storage server via the network. Let us consider the possible routes of data in the first case: When a *write* operation for an array s is called, the data is located in the client memory on computer S. Array s contains data from one or several randomly mapped blocks of data. Let us focus on the data destined for one of the target disks t that is attached to a server machine T. The ideal situation would be that disk t is currently idle and the data is shipped to the network interface card of T which directly fowards it to disk t, bypassing the processor and main memory of T. Since this is difficult to do in a portable way and since t may be busy anyway, the more likely alternative is that S contacts T and asks it to reserve space to put s into the queue of t. If this is impossible, the execution of the *write* operation blocks until space is available or S tries to buffer s locally. Eventually, the data is transferred to the main memory of T. When the request gets its turn, it is transmitted to the disk t which means that it ends up in the local cache of this disk and is then written.

This scenario deviates in several points from the theoretical analysis:

- The nice performance bounds only hold when all disks share the same *global* pool of buffers whereas the implementation makes use only of the local memories of the computer hosting the target disk t. It can be shown that this makes little difference if the local memories are large compared to $\log D$ blocks. Otherwise, one could consider shipping the data to third parties when neither S nor T have enough local memory. But this makes only sense if the the network is very fast.

– Theorem 2 assumes that all written blocks are different. Overwriting a block that is still buffered will save us an output. But it can happen that overwriting blocks that have recently been output can cause additional delays [35]. Again, this can be shown to be unproblematic if the local memory is large compared to $\log D$ blocks. Otherwise dynamically remapping data can help.
– The logical blocks used for random mapping should be fairly large (currently megabytes) in order to allow accesses close to the peak performance of the disks. This can cause a problem for applications that less rely on high throughput for consecutive accesses than on low latency for many parallel fine grained accesses. In that case many consecutive small blocks can lie on the same disk which then becomes a bottleneck. In this case it might make sense to use a separate address space with small logical blocks for fine grained accesses.

4 Random Duplicate Allocation

In the previous section we have seen that random allocation and some buffering allow us to write with high throughput and low latency. The same strategy seems promising for reading data. Randomization ensures that data is spread uniformly over the disks and buffer space can be used for *prefetching* data so that it is available when needed. Indeed, there is a far reaching analogy between reading and writing [16]: When we run a writing algorithm "backwards" we get a reading algorithm. In particular, Theorem 1 transfers. However this reversal of time implies that we need to know the future accesses in advance and we pay the $\mathcal{O}(m \log(D)/D)$ steps for the maximum queue length *up front*.

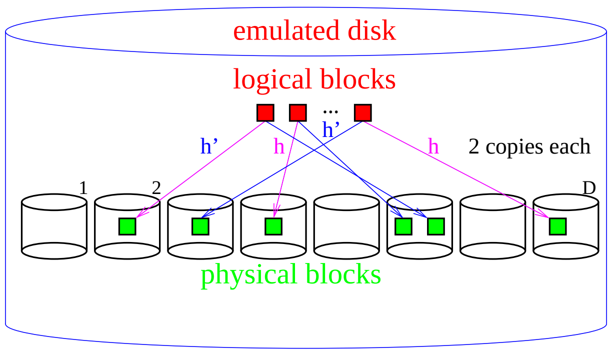

Fig. 3. The concept of (R)andom (D)uplicate (A)llocation.

Therefore, we now bring in an additional ingredient: Each logical block is stored redundantly. Figure 3 illustrates this concept. For now we concentrate on the simple case that each block is allocated to two randomly chosen disks.

Sect. 6 discusses generalizations. Redundancy gives us flexibility in choosing from where to read the data and this allows us to reduce read latencies dramatically. By choosing two *different* disks for the copies we get the additional benefit that no data is lost when a disk fails.

We begin with two algorithms for scheduling a batch of n requested blocks of fixed size that have been analyzed very accurately: The *shortest queue* algorithm allocates the requests in a greedy fashion. Consider a block e with copies on disks d and d' and let $\ell(d)$ and $\ell(d')$ denote the number of blocks already planned for disks d and d' respectively. Then the shortest queue algorithm plans e for the disk with smaller load. Ties are broken arbitrarily. It can be shown that this algorithm produces a schedule that needs

$$k = \frac{n}{D} + \log\ln D + \Theta(1)$$

expected I/O steps [9]. This is very good for large n but has an additive term that grows with the system size.

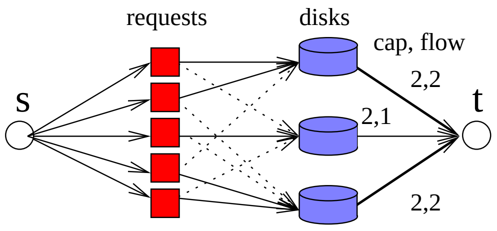

Fig. 4. A flow network showing how five requests are allocated to three disks. The flow defined by the solid lines proves that the requests can be retrieved in two I/O steps.

We will see that *optimal* schedules do not have this problem — we can do better by not committing our choices before we have seen all the requests. Optimal schedules can be found in polynomial time [12]: Suppose we want to find out whether k steps suffice to retrieve all requests. Consider a flow network [2] that consists of four layers: A source node in the first layer is connected to each of n request nodes. Each request node is connected to two out of D disk nodes — one edge for each disk that holds a copy of the requested block. The disk nodes are connected to a sink node t. The edges between disk nodes and t have capacity k. All other nodes have capacity 1. Now it is easy to see that a flow saturating all edges leaving the source node exists if and only if k steps are sufficient. A schedule can be read of an integral maximum flow by reading request r from disk d if and only if the edge (r, d) carries flow. Figure 4 gives an example. The correct value for k can be found by trial and error. First try $k = \lceil n/D \rceil$ then $k = \lceil n/D \rceil + 1, \ldots$, until a solution is found. Korst [19] gives a different flow formulation that uses only D nodes and demonstrates that the problem can be solved in time $\mathcal{O}\big(n + D^3\big)$. If $n = \mathcal{O}(D)$ it can be shown that the problem can be solved in time $\mathcal{O}(n \log n)$ with high probability [35].

Theorem 3. *[35] Consider a batch of n randomly and duplicately allocated blocks to be read from D disks. The optimal algorithm needs at most*

$$\left\lceil \frac{n}{D} \right\rceil + 1$$

steps with probability at least $1 - \mathcal{O}(1/D)^{\lceil n/D \rceil + 1}$.

Proof. (Outline) Using a graph theoretical model of the problem, it can be shown that the requests can be retrieved in k steps if and *only if* there is no subset Δ of disks such that more than $|\Delta|k$ requested blocks have both their copies allocated to a disk in Δ [38]. Hence, it suffices to show that it is unlikely that such an *overloaded subset* exists. This is a tractable problem mostly because the number of blocks allocated to Δ is binomially distributed. ■

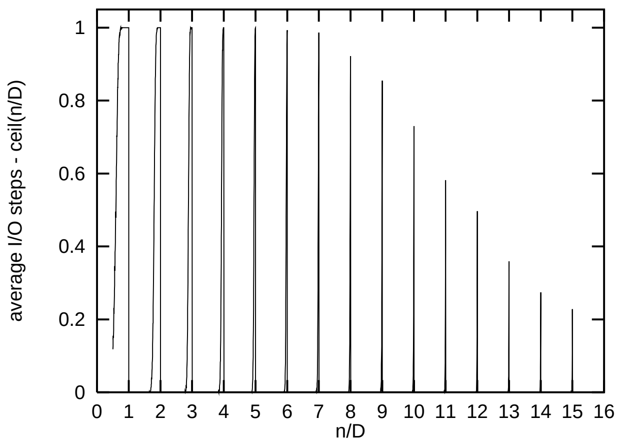

Fig. 5. The number of I/O steps (minus the lower bound $\lceil n/D \rceil$) needed by optimal scheduling when scheduling $n \in 256..4096$ blocks on $D = 256$ disks.

Figure 5 shows the performance of the optimal scheduling algorithm. We only give the data for $D = 256$ because this curve is almost independent of the number of disks. We see that the performance is even better than predicted by Theorem 3: We can expect to get schedules that need only $\lceil n/D \rceil$ steps except if n is a multiple of D or slightly below. For example, when $n = 3.84D$, we almost always get a schedule with 4 steps, i.e., we are within 4 % of the best possible performance. We also see that for large n we can even get perfect balance when n is a multiple of D.

Open Problem: 3 *Is there a threshold constant c such that for $n > cD \log D$, optimal scheduling finds a schedule with $\lceil n/D \rceil$ I/O steps even if n is a multiple of D?*

4.1 The Selfless Algorithm

One problem with optimal scheduling is that we do not have good performance guarantees for scheduling large sets of blocks efficiently. Therefore it makes sense to look for fast algorithms that are close to optimal. Here we describe linear time algorithm that produces very close to optimal solutions [11].

The *selfless algorithm* distinguishes between *committed* and uncommited requests. Uncommitted requests still have a choice between two disks. Committed requests have decided for one of the two choices. Initially, all requests are uncommitted. A disk d is called committed if there are no uncommited requests that have d as a choice. Let the *load* $\ell(d)$ of disk d denote twice the number of requests that have committed to d plus the number of uncommitted requests that have d as an option. The load could be viewed as twice the expected number of committed requests of a disk if all uncommitted requests would choose randomly between one of their two choices. The selfless algorithm is based on two simple rules:

1. If there is an uncommitted disk with at most $\lceil n/D \rceil$ remaining incident requests, we commit all of them to this disk.
2. Otherwise, we choose an uncommitted disk d with minimum load, choose an uncommited request with d as an option, and commit it to d.

This algorithm can be implemented to run in linear time using fairly standard data structures: Disks are viewed as nodes of a graph. Uncommitted requests are edges. Using an appropriate graph representation, edges can be removed in constant time (e.g., [21]). When a disk becomes a candidate for Rule 1, we remember it on a stack. The remaining nodes are kept in a priority queue ordered by their load. Insert, decrement-priority and delete-minimum can be implemented to run in amortized constant time using a slight variant of a bucket priority queue [14].

If we would plot the performance of the selfless algorithm in the same way as in Figure 5 it would be absolutely impossible to see a difference, i.e., with very high probability the selfless algorithm finds an optimal schedule. This will be proven in an upcoming paper [11] using differential equation methods that have previously been used for the mathematically closely related problem of *cores* of random graphs [28].

5 Variable Size Requests

We now drop the assumption that we are dealing with fixed size jobs that take unit time to retrieve. Instead, let $\ell_i \leq 1$ denote the time needed to retrieve request i. This generalization can be used to model several aspects of storage servers:

- We might want to retrieve just parts of a logical block
- Disks are divided into *zones* [30] of different data density and correspondingly different data rate — blocks on the outer zones are faster to retrieve than blocks on the inner zones. We assume here that both copies of a block are stored on the same zone.

The bad news is that it is strongly NP-hard to assign requests to disks so that the I/O time is minimized [1]. The good news is that optimal scheduling is still possible if we allow request to be split, i.e., we are allowed to combine a request from pieces read from both copies. We make the simplifying assumption here that a request of size $\ell = \ell_1 + \ell_2$ stored on disks d_1 and d_2 can be retrieved by spending time ℓ_1 on disk d_1 and time ℓ_2 on disk d_2. This is approximately true if requests are large.

Even the performance guarantees for random duplicate allocation transfer. We report a simplified version of a result from [33] that has the *same* form as Theorem 3 for unit size requests:

Theorem 4. *Consider a set R of request with total size $n = \sum_{r \in R} \ell_r$ of randomly and duplicately allocated requests to be read from D disks. The optimal algorithm computes a schedule with I/O time at most*

$$\left\lceil \frac{n}{D} \right\rceil + 1 \text{ with probability at least } 1 - \mathcal{O}(1/D)^{\lceil n/D \rceil + 1}$$

The proofs and the algorithms are completely analogous. The only difference is that the maximum flows will now not be integral and hence require splitting of requests. Splitting can also have a positive effect for unit size requests since it eliminates threshold effects such es the steps visible in Fig. 5. A more detailed analysis indicates that the retrieval time becomes a monotonic function of the number of requests [33].

In a sense, Theorem 4 is much more important than Theorem 3. For unit size requests, we can relatively easily establish the expected performance of an algorithm by simulating all interesting cases a sufficient number of times. Here, this is not possible since Theorem 4 holds for a vast space of possible inputs (uncountably big and still exponential if we discretize the piece sizes).

6 Reducing Redundancy

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 physical blocks such that reading any r out of the w blocks suffices to reconstruct the logical block. Perhaps the most important case is $w = r + 1$. Using *parity-encoding*, r of the blocks are simply pieces of the logical block and the last block is the exclusive-or of the other blocks. A missing block can then be reconstructed by taking the exclusive-or of the blocks 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. The special case $r = D - 1, W = D$ yields the coding scheme used for RAID levels 3 and 5.

Choosing $w > r + 1$ can be useful if more than one disk failure is to be tolerated (see Sect. 8) or if we additionally want to reduce output latencies (see

Sect. 9). A disadvantage of codes with $w > r+1$ is that they are computationally more expensive than parity-encoding [20, 15, 29, 10, 4, 8].

Most of the scheduling algorithms for RDA we have discussed are easy to generalize for more general coding schemes. Only optimal scheduling needs some additional consideration. A formulation that is a generalization of bipartite matching [32] yields a polynomial time algorithm however.

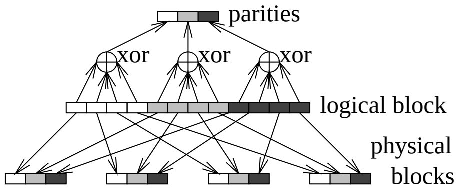

Fig. 6. Encoding of a logical block of size 12 into 4 physical blocks and one parity block of size 3 such that aligned logical requests of size $4s$, $s \in \{1, 2, 3\}$ can be fulfilled by retrieving any 4 out of 5 physical blocks of size s.

A small trick also allows us to use general coding schemes for arbitrary request sizes: As before, data is allocated for large logical blocks whereas actual requests may retrieve parts of these blocks. But the coding is done in very small pieces (say sectors of size 512) and the encoded pieces are stored in the physical blocks in an interleaved fashion. Figure 6 gives an example.

7 Asynchronous Access

In Sect. 3.3 we have already explained how writing can be implemented in an asynchronous, distributed way by providing one thread for each disk. We now explain how this can be generalized for read accesses in the presence of redundant allocation. Client requests for a block of data arrive individually in an asynchronous fashion. The clients want to have these requests answered quickly, i.e., they want small delays. The algorithms described in Sect. 4 can be generalized for this purpose [34]. For example, the shortest queue algorithm would commit the request to the disk that can serve it fastest.

However, we loose most of the performance guarantees. For example, it is easy to develop an algorithm that minimizes the maximum delay among all *known* requests but it is not clear how to *anticipate* the impact of these decisions on requests arriving in the future.

Open Problem: 4 *Give theoretical bounds for the expected latency of any of the asynchronous scheduling algorithms discussed in [34] as a function of D and ϵ in the following model: A block access on any of the D disks takes unit time. A request for a block arrives every $(1+\epsilon)/D$ time units. For non-redundant random allocation it can be shown that the expected delay is $\Theta(1/\epsilon)$. Experiments* *and heuristic considerations suggest that time $\mathcal{O}(\log(1/\epsilon))$ is achievable using redundancy.*

Asynchrony also introduces a new algorithmic concept that we want to discuss in more detail: *Lazy decisions.* The simplest lazy algorithm — *lazy queuing* — queues a request readable on disks d and d' in queues for *both* d and d'. The decision which disk actually fetches the block is postponed until the last possible moment. When a disk d falls idle, the thread responsible for this disk inspects its queue and removes one request r queued there. Then it communicates with the thread responsible for the other copy of r to make sure that r is not fetched twice. Lazy queueing has the interesting property that it is equivalent to an "omniscient" shortest queue algorithm, i.e., it achieves the same performance even if it does not know how long it takes to retrieve a request.

Theorem 5. *Given an arbitrary request stream where a disk d needs $t(d, r)$ time units to serve request r. Then the lazy queue algorithm produces the same schedule as a shortest queue algorithm which exactly computes disk loads by summing the $t(d, r)$-values of the scheduled requests.*

The only possible disadvantage of lazy algorithms compared to "eager" algorithms such as shortest queue is that a simple implementation can incur additional communication delays at the performance critical moment when a disk is ready to retrieve the next request (asking another thread and waiting for a reply). This problem can be mitigated by trying to agree on a *primary* copy of a request r before the previous request finishes. The disk holding the primary copy can then immediately fetch r and in parallel it can send a confirmation to the thread with the other copy.

Figure 7 shows that RDA significantly outperforms traditional output schemes. Even mirroring that has the same amount of redundancy produces much larger degrees when the storage server approaches its limits. Measurements not shown here indicate that the gap is much larger when we are interesting in the largest delays that are encountered sufficiently often to be significant for real time applications such as video streaming.

It can also be shown that fluctuations in the arrival rate of requests have little impact on performance if the the number of requests arriving over the time interval of an average delay is not too big. Furthermore, the scheduling algorithms can be adapted in such a way that applications that need high throughput even at the price of large delays can coexist with applications that rely on small delays [32].

8 Fault Tolerance

When a disk fails, the peak system throughput decreases by a small factor of $1/D$. Furthermore, requests which have a copy on the faulty disks lose their scheduling flexibility. Since only few requests are affected, load balancing still works well [33]. In addition, there are now logical blocks that have less redundancy than the

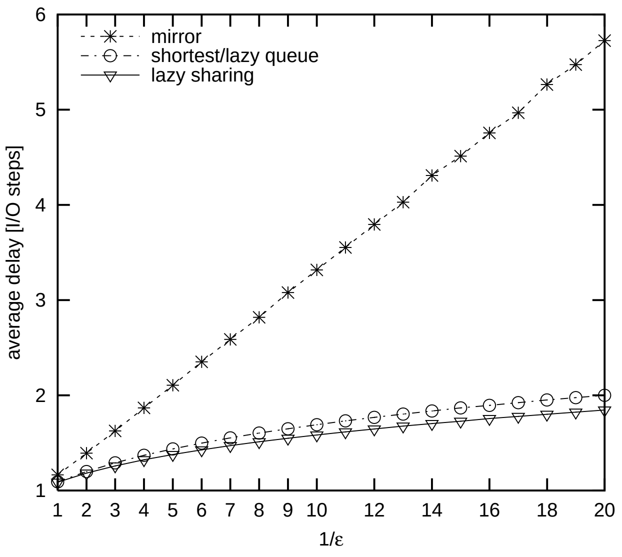

Fig. 7. Average delays for 10^7 requests to $D = 64$ disks arriving at time intervals of $(1+\epsilon)/D$. The mirror algorithm uses random allocation to a RAID-1 array. Lazy sharing is a refinement of lazy queue — an idle disk d leaves a request r to the alternative disk $d'(r)$ if the queue of d' is shorter than the queue of d.

others so that additional disk failures could now lead to a loss of data. To get out of this dangerous situation, the lost redundancy has to be reestablished. This can be achieved without exchanging any hardware by dispersing these blocks over unused space of the other disks. This can be done very quickly because the data read and written for this purpose is uniformly distributed over all disks. If we are willing to invest a fraction of ϵ of the peak performance of the system, the reconstruction can finish in a fraction of about $1/\epsilon(D-1)$ of the time needed to read and write one disk. For example, in a large system with 10 000 disks with 100GByte each and a disk I/O rate of 50MByte/s we could in principle reconstruct a failed disk in as little as ten seconds investing 4 % of our peak I/O rate. Section 10 will explain how a random mapping of the data is maintained in this situation.

Failures of disk controllers or entire machines can be handled in a similar manner if the random duplicate allocation is modified in such a way that different copies are allocated to different pieces of hardware. The ultimate realization of this strategy divides the storage server into halves that are physically so far apart that even a fire or another catastrophe is unlikely to destroy both halves at the same time. The limiting factor here are the costs of a high speed interconnection between the halves so that for such systems one may consider to have more than two copies of each block and to send data to the remote half only occasionally.

A major challenge in practical fault tolerance is that it is very difficult to test the behavior of distributed software under faults. Since the data losses can be really expensive, storage servers might therefore be a prime candidate for formal verification:

Open Problem: 5 *Define a useful abstract model of a storage server and its software and* prove *that it operates correctly under disk failures, power loss, ...*

9 Reducing Write Latency

Somewhat paradoxically, there are many applications where *write*s are much more frequent than *read*s, i.e., much of what is written is never read. The reason is that a lot of the data needed by a client can be cached in main memory (by the storage server or by the application). One could argue that one would not have to output this data at all but this neglects that many applications must be able to recover their old state after a power loss.

There are several ways to handle this situation. One would be to make sure that write buffers are in memory with enough battery backup that they can be flushed to disk at a power loss. The next step on the safety ladder makes sure that the data is buffered in *two* processors with independent power supply before a *write* operation returns. But some applications will still prefer to wait until the data is actually output. In this situation, the strategy from Section 3 leads to fairly long waiting time under high load.

In this situation, the generalized coding schemes outlined in Sect. 6 can be used. If a logical block can be reconstructed from any r out of w pieces, we can return from a *write* operation after r' pieces are output ($r \leq r' \leq w$) and we get a flexible tradeoff between write latency and fault tolerance. For example, for $r = 1$ and $w = 3$ we could return already when two copies have been written. Which copies are written can be decided using any of the scheduling algorithms discussed above, for example the lazy queueing algorithm from Sect. 7. The remaining copy is not written at all or only with reduced priority so that it cannot delay other time critical disk accesses.

10 Inhomogeneous Dynamically Evolving Storage Servers

A storage server that operates reliably 24h a day 365 days a year should allow us to add disks dynamically when the demands for capacity or bandwidth increase. Since technology is continuously advancing, we would like to add new disks with higher capacity and bandwidth than the existing disks. Even if we would be willing to settle for the old type, this becomes infeasible after a few years when the old type is no longer for sale. In Sect. 8 we have already said that we want to be able to remove failed disks from the system without replacing them by new disks. The main algorithmic challenge in such systems is to maintain our concept of load balancing by randomly mapping logical blocks to disks. We first explain how a single random mapping from the virtual address space to the disks is obtained.

Inhomogeneity can be accommodated by mapping a block not directly to the D inhomogeneous disks but first to D' *volumes* that accommodate $N/D'B$ blocks each. The volumes are then mapped to the disks in such a way that the ratio $r(d) = c(d)/v(d)$ between the capacity $c(d)$ of a disk d and the number of volumes $v(d)$ allocated to it is about the same everywhere. More precisely, when a volume is allocated, it is greedily moved to the disk that maximizes $r(d)$. If $D'/D \gg D \max c(d)/\sum_d c(d)$, we will achieve a good utilization of disk capacity.[5]

When a disk fails, the volumes previously allocated to it will be distributed over the remaining disks. This is safe as long as $\min_d r(d)$ exceeds N/D'. When a new disk d' is added, volumes from the disks with smallest $r(d)$ are moved to the new disk until $r(d')$ would become minimal. In order to move or reconstruct volumes, only the data in the affected volumes needs to be touched whereas all the remaining volumes remain untouched.

It remains to define a random mapping of blocks to volumes. We present a pragmatic solution that outperforms a true random mapping in certain aspects but where an accurate analysis of the scheduling algorithms remains an open question. We achieve a perfectly balanced allocation of blocks to volumes, by *striping* blocks over the volumes, i.e., blocks $iD'..(i+1)D'-1$ are mapped in such a way that each volume receives one block. To achieve randomness, block $iD'+j$, $0 \le j < D$, is mapped via a (pseudo)random permutation π_i to volume $\pi_i(j)$. Figure 8 summarizes the translation of logical addresses into block offsets, disk IDs, and positions on the disk.

In order to find out which blocks need to be moved or reconstructed when a disk is added or replaced, we would like to have permutations that are easy to invert. *Feistel* permutations [25] are one way to achieve that: Assume for now that $\sqrt{D'}$ is an integer and represent j as $j = j_a + j_b\sqrt{D'}$. Now consider the mapping

$$\pi_{i,1}((j_a, j_b)) = (j_b, j_a + f_{i,1}(j_b) \bmod \sqrt{D'})$$

where $f_{i,1}$ is some (pseudo)random function. If we iterate such mappings two to four times using pseudo-random functions $f_{i,1}, \ldots, f_{i,4}$ we get something "pretty random". Indeed, such permutations can be shown to be random in some precise sense that is useful for cryptology [25]. A Feistel permutation is easy to invert.

$$\pi_{i,k}^{-1}((a,b)) = (b - f_{i,k}(a) \bmod \sqrt{D'}, a)$$

We assume that the functions $f_{i,k}$ are represented in some compact way, e.g., using any kind of ordinary pseudo-random hash function h that maps triples

[5] If disk bandwidth is more of an issue than disk capacity, we can also balance according to the data rate a disk can support. But even without that, the lazy scheduling algorithms from Sect. 7 will automatically direct some traffic away from the overloaded disks.

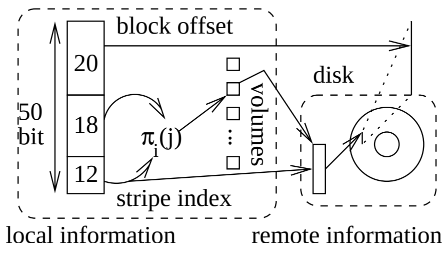

Fig. 8. How a logical address is mapped to a physical block. The numbers give an example with one Petabyte of address space, $B = 2^{20}$, and $D' = 2^{18}$ that would currently require about 10 000 disks of 100 GByte each.

(i, j, k) to values in $0..D' - 1$. In order to find out to which disk a block is mapped, the only additional data structure we need is a lookup table of size D'. This data structure is easy to replicate to the local memory of all processors. For example, even in a large system with $D = 10000$, with disk capacities varying by a factor of four, $D' = 2^{18}$ would already achieve quite good load balance. To achieve fault tolerance, this lookup table and the parameters of the hash function h should be stored redundantly at a predefined place. But even if the table gets lost, it can be reconstructed as long as the capacity of the disks and the order in which they were added or removed is known — we only need to make sure that the algorithms for mapping volumes to disks are deterministic.

Redundant Allocation

In order to use the above scheme in the context of duplicate allocation, we partition the storage server into two partitions whose total storage capacity is about equal. The volumes are mapped to *both* partitions and we have two sets of random permutations — one for each partition. More generally, if we use a coding scheme that writes w physical blocks for each logical block, we need w partitions. To achieve good fault tolerance, components in different partitions should share as few common points of failure as possible (controllers, processors, power supplies, ...). Therefore, the disks will not be assigned to the partitions one by one but in coarse grained units like controllers or even entire machines.

Although this partitioning problem is NP-hard, there are good approximation algorithms [3]. In particular, since we are dealing with a small constant number of partitions, fully polynomial time approximation schemes can be developed using standard techniques [44].

Maintaining reasonably balanced partitions while components enter (new hardware) or leave (failures) the system in an online fashion is a more complicated problem. In general, we will have to move components but these changes in configuration should only affect a small number of components with total capacity proportional to added or removed capacity. At least it is easy to maintain the invariant that the difference between the capacities of the smallest and largest partition is bounded by the maximum component capacity.

11 Discussion

We have introduced some of the algorithmic backbone of scalable storage servers. We have neglected many important aspects because we believe that they are orthogonal to the concepts introduced here, i.e., their implementation does not much affect the decisions for the aspects discussed here: We need an infrastructure that allows reliable high bandwidth communication between arbitrary processors in the network. Although random allocation helps by automatically avoiding hot spots, good routing strategies can be challenging in inhomogeneous dynamically changing networks.

Caching can make actual disk accesses superfluous. This is a well understood topic for centralized memory [17, 26] but distributed caching faces interesting tradeoffs between communication overhead and cache hit rate.

There are many more important issues with a different flavor such as locking mechanisms to coordinate concurrent accesses, file systems, real time issues, ...

In addition, there are interesting aspects that are less well understood yet and pose interesting questions for future work. For example, we have treated all data equal. But in reality, some data is accessed more frequently than other data. Besides the short term measure of caching, this leads to the question of *data migration* (e.g. [37]). Important data should be spread evenly over the disks, it should be allocated to the fastest zones of the disks, and it could be stored with higher redundancy. The bulk of the data that is accessed rarely, could be stored on cheaper disks or even on disks that are powered down for saving energy. Such **M**assive **A**rrays of **I**dle **D**isks [13] are a candidate for replacing tape libraries and could scales to 10s of thousands of disks.

Acknowledgements

In the last years I had many interesting discussions about algorithms for storage servers. Here is a partial list of people who have helped me to form a coherent picture of this complex issue between algorithmics, probability, coding, operating systems, data bases, computer architecture, market mechanisms, patents, ... : Joep Aerts, Eitan Bachmat, Petra Berenbrink, Andre Brinkmann, Sebastian Egner, Harald Hüls, David Irwin, Mahesh Kallahalla, Jan Korst, Jan Marien, Kurt Mehlhorn, Kay Salzwedel, Christian Scheideler, Martin Skutella, Peter Varman, Jeff Vitter, Gerhard Weikum, Winfried Wilcke, Gerhard Woeginger.

References

1. J. Aerts, J. Korst, and W. Verhaegh. Complexity of retrieval problems. Technical Report NL-MS-20.899, Philips Research Laboratories, 2000.
2. R. K. Ahuja, R. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, 1993.
3. N. Alon, Y. Azar, G. J. Woeginger, and T. Yadid. Approximation schemes for scheduling. In *SODA*, pages 493–500, 1997.
4. 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)*, volume 25,2 of *Computer Architecture News*, pages 62–72, New York, June 2–4 1997. ACM Press.

5. M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. In IEEE, editor, *37th Annual Symposium on Foundations of Computer Science*, pages 550–559. IEEE Computer Society Press, 1996.
6. Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *26th ACM Symposium on the Theory of Computing*, pages 593–602, 1994.
7. R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.
8. 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*, pages 2–12, Paderborn, Germany, October 5 1999.
9. P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. In *32th Annual ACM Symposium on Theory of Computing*, pages 745–754, 2000.
10. 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*, pages 245–254, 1994.
11. J. Cain, P. Sanders, and N. Wormald. A random multigraph process for linear time almost optimal RDA disk scheduling. manuscript in preparation, 2003.
12. L. T. Chen and D. Rotem. Optimal response time retrieval of replicated data (extended abstract). In ACM, editor, *13th ACM Symposium on Principles of Database Systems*, volume 13, pages 36–44, New York, NY 10036, USA, 1994. ACM Press.
13. D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *SC'2002 Conference CD*, Baltimore, MD, November 2002. IEEE/ACM SIGARCH.
14. R. B. Dial. Algorithm 360: Shortest-path forest with topological ordering. *Communications of the ACM*, 12(11):632–633, 1969.
15. 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. Technical report, U. C. Berkley, 1988.
16. D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. In *9th European Symposium on Algorithms (ESA)*, number 2161 in LNCS, pages 62–73. Springer, 2001.
17. S. Irani. Competetive analysis of paging. In *Online Algorithms — The State of the Art*, volume 1442 of *LNCS*, pages 52–73. Springer, 1998.
18. 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*, pages 318–326, May 1992.
19. J. Korst. Random duplicate assignment: An alternative to striping in video servers. In *ACM Multimedia*, pages 219–226, Seattle, 1997.
20. F.J. MacWilliams and N.J.A. Sloane. *Theory of error-correcting codes.* North-Holland, 1988.
21. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, 1999.
22. U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Tutorial.* Springer, 2003.
23. E. L. Miller and R. H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23:419–446, 1997.
24. R. Muntz, J.R. Santos, and S. Berson. A parallel disk storage system for real-time multimedia applications. *International Journal of Intelligent Systems*, 13:1137–1174, 1998.

25. M. Naor and O. Reingold. On the construction of pseudorandom permutations: Luby-Rackoff revisited. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 12(1):29–66, 1999.
26. O'Neil, O'Neil, and Weikum. An optimality proof of the LRU-K page replacement algorithm. *JACM: Journal of the ACM*, 46, 1999.
27. D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of ACM SIGMOD'88*, pages 109–116, 1988.
28. B. Pittel, J. Spencer, and N. Wormald. Sudden emergence of a giant k-core in random graph. *J. Combinatorial Theory, Series B*, 67:111–151, 1996.
29. M. O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.
30. C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.
31. K. Salem and H. Garcia-Molina. Disk striping. *Proceedings of Data Engineering'86*, 1986.
32. P. Sanders. Asynchronous scheduling of redundant disk arrays. In *12th ACM Symposium on Parallel Algorithms and Architectures*, pages 89–98, 2000. extended version to appear in IEEE Transactiopns on Computers.
33. P. Sanders. Reconciling simplicity and realism in parallel disk models. *Parallel Computing*, 28(5):705–723, 2002. short version in 12th SODA, p. 67–76.
34. P. Sanders. Asynchronous scheduling of redundant disk arrays. *IEEE Transactions on Computers*, 52(9):1170–1184, 2003.
35. P. Sanders, S. Egner, and Jan Korst. Fast concurrent access to parallel disks. *Algorithmica*, 35(1):21–55, 2003. short version in 11th SODA, p. 849–858.
36. J. R. Santos, R. R. Muntz, and B. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *ACM SIGMETRICS*, pages 44–55, 2000.
37. P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *VLDB Journal: Very Large Data Bases*, 7(1):48–66, 1998.
38. L. A. M. Schoenmakers. A new algorithm for the recognition of series parallel graphs. Technical Report CS-R9504, CWI - Centrum voor Wiskunde en Informatica, January 31, 1995.
39. H. S. Stone and S. F. Fuller. On the near-optimality of the shortest-access-time-first drum scheduling discipline. *Communications of the ACM*, 16(6):352–353, June 1973. Also published in/as: Technical Note No.12, DSL.
40. W. Tetzlaff and R. Flynn. Block allocation in video servers for availability and throughput. *Proceedings Multimedia Computing and Networking*, 1996.
41. 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*, pages 144–150, 1996.
42. J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.
43. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two level memories. *Algorithmica*, 12(2/3):110–147, 1994.
44. G. J. Woeginger. When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (fptas). *INFORMS Journal on Computing*, 12:57–75, 2000.