To all those who cannot read these lines anymore:

Hannelore Salzwedel
Prof. Dr. Johannes Röhrich
Gabriel Terán Martinez

Data Distribution Algorithms for Storage Networks

Dissertation by Kay A. Salzwedel

Reviewer:

- Prof. Dr. math. Friedhelm Meyer auf der Heide, University of Paderborn
- Prof. Dr.-Ing. Ulrich Rückert, University of Paderborn

Contents

CHAPTER 1

Introduction

I know that it will happen
'Cause I believe in the certainty of chance

Neil Hannon, The Divine Comedy

The ability to store data persistently is one of the key elements of information technology. In modern computers this task is usually realised by magnetic hard disk drives. They form the last and, therefore, the slowest level in the memory hierarchy.

The sophisticated and complex advances in current disk drive technology have made possible an accelerated growth in disk capacity. Nowadays, a single off-the-shelf disk drive is capable of storing up to 180 gigabyte[1]. Nevertheless, this is not sufficient to satisfy the ever growing volumes of data needing to be stored. The rapid evolution in the processing power of modern processors, the global availability of information, and the increasing use of mixed-media contents, demand more flexible storage systems. The easiest way of providing almost unlimited storage capacity is the clustering of many disk drives connected in an arbitrary network called a *storage network*. This does not only provide the flexibility to adapt to changing capacity demands but also allows for the parallel use of disks and, therefore, bears the capability to achieve faster response times and reduces the likeliness of bottlenecks, like hot spots.

Unfortunately, the inherent advantages of storage networks are not easily exploited. The management of many disk drives in parallel poses the question of *data placement*. The most simple solution is to map new data to the first available free position. However, this results not only in a scattered data layout, it also makes additional functionality, such as reliability via redundancy, parallel data

[1] In the past years, the capacity of disks doubled roughly every 14 – 18 months.

access, or dynamic adaption to changed capacity demands a very complex, cumbersome, error-prone, and very expensive task. Furthermore, the maintenance and manageability of a large number of disks is much easier when they can conceptually be treated as one entity. Data recovery and periodical backups are issued to one system, hiding the physical representation from the administrator.

To further illustrate the above reasoning, consider the following situation. Suppose some data is stored on a number of disks of equal size. The data throughput is maximised by striping the data over all disks so that as many requests as possible are serviced by as many disks as possible. Now suppose that the available storage capacity is no longer sufficient due to an increased data volume. To meet this requirement, a number of newer and larger disks are introduced to the system. In order to keep the high level of performance, we must preserve the striped layout. However, this results in the redistribution of almost all of the data. During the redistribution phase, the performance of the system is reduced substantially. The more data has to be redistributed, the longer the impact on the overall performance lasts.

Clearly, there are many storage systems capable of handling a capacity growth easily, up to a limit. Most of them are monolithic and realise the redistribution via redundant data paths. However, these systems are rather expensive and the general problem remains when even these systems run out of storage space.

All of the above reasoning substantiates the need for more general management techniques of an assemblage of disk drives, i.e. a *storage network*. Such a system is a collection of disk drives which are connected by a linking network. Each of the disks is characterised by its capacity and its bandwidth (see Figure 1.1). The underlying topology or architecture of the network is unrestricted. Hence, we are not concerned with network properties like congestion or latency. The topology may vary from disks attached to a SCSI bus up to very complex networks like peer-to-peer (P2P) networks where the disk drives are associated with server nodes. In such an overlay network, a number of servers work together to solve special tasks like searching. There is no central authority to control the behaviour, and the servers might enter or leave the network at any time.

Having a closer look at the abstraction mentioned above, two crucial observations become apparent. First, larger demands for capacity are easily solved by attaching more disks to the network. However, the data needs to be distributed in an intelligent way to provide an even utilisation of the participating disks. Only a good data distribution can ensure the scalability in terms of the number of served data requests when the number of disk drives is increased.

Second, a storage network offers the possibility of parallel data accesses provided that the data resides on different disk drives. This improves not only the access time but also the disk utilisation because requests for larger data chunks are served by more then one disk and, hence, data requests are more balanced.

Yet using storage networks also imposes an inherent limitation – availability. Every disk has an estimated operation time during which an error is rather unlikely to occur. This entity is called the mean time to/between failure (MTTF or MTBF, respectively), and for modern drives it is approximately 1,000,000 hours. For a whole storage network, the MTBF decreases significantly. This is due to the fact that it depends on the failure rate λ_{sys} [SS92, Gib91] which is simply the sum of the failure rates λ_i of each participating disk[2]. Hence, we need to incorporate some form of redundancy into the data placement to compensate for the loss of disk drives.

The observations made above can be formalised as the *data distribution problem*. The question to be answered is: Where should the data be stored such that any data request can be answered as fast as possible? This is the essence of all the above storage considerations. Even the issue of availability can be reduced to that of fast data access if one has to regenerate data (from a saved checkpoint) that is not accessible at the moment. We identify a number of properties that must be addressed by a good data distribution in storage networks.

First of all, the data distribution has to ensure *scalability*. The more disks there are in a storage network, the more data requests there are to be answered and, hence, the faster the data should be delivered. This can only be achieved if the data is evenly distributed over all disks. Hence, a good distribution strategy must *balance* the data evenly over all disks. The importance of this balance is further emphasised by the accesses to the data. They are dependent on the data distribution, and, in general, there is little knowledge about the underlying access pattern. Therefore, the best a data distribution can do is to be efficient for all possible access patterns.

In recent years, modern storage systems have moved towards distributed solutions. Such systems usually consist of common components connected by standardised technology like Gigabit Ethernet or Fibre Channel. Hence, it is rather unlikely that all components have the same capacity or performance, especially, if the systems must possess the ability to adapt over time. Furthermore, the sheer increase of stored data volume leads to a constant growth of systems. However, it is rather unlikely that one always favours a new system instead of extending the old one. This leads to the *heterogeneity* problem. Not exploiting the different properties of each component may result in a waste of capacity and performance.

Closely related to the heterogeneity problem is the problem of *adaptivity*. If we would like to extend an existing system, we want to do so as fast as possible. Due to the fact that the new disks are only of value if we use them as soon as their are available, we want to keep the resulting redistribution phase, in which the overall performance is reduced, as short as possible. This leads to a new challenge for storage networks – they should easily *adapt* to changing capacity/bandwidth

[2]This is only true under the assumption that the failure rate is constant.

demands. The addition of disks is an easy solution to the storage problem if the space/access balance requirement is ignored. One simply extends the system with new disks and maps new data to recently added disks. However, this results in a distribution that cannot increase the access performance for all data, even if newer and faster disks are added. The only way to achieve both goals, space balance and the addition of disks, is to redistribute some data. This property is measured by the *adaptivity*. Naturally, the less data is redistributed the more adaptive is the underlying strategy.

The above discussion can be summarised by defining a number of requirements that a storage network has to meet.

1. **Space and Access Balance:** To ensure good disk utilisation and provide scalability the data should be evenly distributed over all disk drives. The data requests are usually generated by a file system above the storage network. Hence, the storage network is unlikely to have any knowledge about the access pattern and must handle any possible request distribution.

2. **Availability:** Storage networks need to implement some form of redundancy and suitable recovery mechanisms to tolerate the loss of failed disk drives, because of the increased sensitivity to disk failures.

3. **Access Efficiency:** Stored data that cannot be accessed fast enough is lost data. Access efficiency is defined by the *time* and *space* requirements needed to answer a request for an arbitrary data element. This property is of crucial importance when large networks are concerned. Furthermore, it assists the scalability. The resources available to any storage network are limited. If the data can only be accessed by consuming large portions of these valuable resources, the maximally manageable size of a storage network is very limited.

4. **Heterogeneity:** Heterogeneity describes the capability to handle disks of different properties like capacity or bandwidth.

5. **Adaptivity:** The data volume is constantly growing and even generously planned systems can quickly run out of storage space. This property measures the ability to adapt quickly to a changed situation such as the addition of a new disk drive.

Although it seems impossible to meet all of the above requirements simultaneously, especially as heterogeneity and adaptivity are challenging if space/access balance and access efficiency must be guaranteed. However, randomisation can ease the task significantly. Applied to the used data blocks, it ensures an even distribution over a specialised interval, and, hence, the distribution algorithm only has to concern itself with this interval.

This thesis focuses on heterogeneous, adaptive data distribution strategies that are capable of balancing the data requests as well as the data itself. We do not investigate fault tolerant schemes because they could be easily applied on top of a distribution strategy. The thesis is divided into two main parts. In the first part, we build the theoretical background, then describe and analyse the strategies. Their simple structure together with their adaptivity make them very interesting for storage virtualisation. In the second part of the thesis, we introduce a storage virtualisation based on the Share strategy, and we present results concerning its induced overhead and quality.

In the next chapter, we give an overview of previous results and then lay the theoretical groundwork for homogeneous distribution strategies. They are substantially easier than their heterogeneous counterparts because all disks get the same amount of data. Hence, if the number of disks changes, the amount of redistributed data is equal for all disks. We begin with the Nearest Neighbour strategy proposed by Karger et al [KLL+97]. This strategy deserves attention because it is applied later as a subroutine in more general algorithms. We analyse its behaviour according to our measures.

Our first strategy is the Cut-and-Paste strategy introduced in [BSS00]. The given analysis proves not only that it achieves a much better space balance compared to the Nearest Neighbour strategy, but also that it adapts to changes by redistributing a data volume that is close to optimal.

The first heterogeneous data distribution is the Level strategy proposed in [BSS00]. As the name suggests, we introduce levels within which the participating disks are treated uniformly. Thus we can apply the Cut-and-Paste strategy to solve the data distribution problem in each level. Unfortunately, the adaptivity of this strategy as well as its rather complex structure render it unsuitable for practical purposes.

This is one of the reasons that we devised the Share and the Sieve strategies [BSS02]. Both are space balanced and achieve a close to optimal adaptivity. While the Share strategy resolves heterogeneity in one phase and uses a faithful uniform scheme as a subroutine, the Sieve strategy relies completely on pseudo-random hash functions.

The ability to handle disks with different characteristics, together with the efficient adaptation to changes in the storage network, give rise to a new way of managing storage virtualisation. Therefore, we used the Share strategy as the core of our virtualisation approach [BMS+03]. We achieved feasibility by coarsening the data access; instead of keeping information for every data block, we introduced extents - a collection of consecutive data blocks. Together with a database for ensuring the consistency of the needed information, the implemented device driver is capable of realising storage virtualisation efficiently.

How reliable is our randomised approach in practice? Surely, this is an important question because the quality of the Share strategy depends not only on

hash functions but also on the number of disks in the network, the number of used extents, and the value for a quality parameter called stretch factor. The last chapter is dedicated to answer this question. First, investigate the distribution quality of our implementation. This is done by exploring different aspects such as the theoretically best possible distribution and the number of accessed elements needed to achieve such a distribution.

A second set of experiments was conducted to estimate the induced overhead of the whole storage virtualisation [BHM+04]. The key issue is the influence of each component (driver, data transfer and performance of the database) on the data throughput compared to the performance of a pure disk. We show that our approach loses roughly 10% to 15% of the maximal achievable throughput. However, we gain a considerable speedup when the extents are distributed over more than one disk. In particular, the number of random data accesses, the setting that resembles real applications best, scales almost optimally.

The quintessence of the contribution of this thesis can be formulated as follows: We liberate the use of storage networks by introducing dynamic behaviour. The devised strategies are not only proven to be theoretically sound concerning their quality but are also evaluated using real-life applications.

1.1 Preliminaries

In this section, we describe the abstractions made to tackle the given problem theoretically. The starting point is the physical representation of data on persistent storage devices, e.g. magnetic disk drive. The distinctiveness of data accesses to these devices leads to a special handling inside most operating systems. Exemplarily, we give an overview of disk accesses under Unix, particularly under Linux. A more detailed description can be found in Section 4.3.

The hardware fundamentals and basic access strategies for disk drives form the foundation of the abstractions made by our model. More specifically, we use the block device interface as our level of abstraction for data access, i.e. data is organised as an array of constant sized blocks. However, the presented strategies are more general and not limited to that level. The description of our model is concluded by the formal definition of the data placement problem for which we will devise algorithms later on.

1.1.1 Magnetic Disk Drives

Nowadays, persistent storage is realised by magnetic disk drives. These devices consist, in principle, of a number of rotating platters with a sensitive surface and a number of mechanical read/write heads. The surface of each platter is divided into concentric rings called tracks. This enhances performance because accessing

a whole track does not involve the comparatively slow movement of the head. Each track is further divided into sectors, the smallest accessible unit on a disk. Clearly, there are plenty of optimisations that may be applied to the sector layout, but for our purpose this can be neglected (see [RW94] for more detail).

The access to an arbitrary data item involves the following steps. First, the head has to be moved to the right track. The time to do so consists not only of the time to seek the right track (t_{seek}) but also of the time to settle the head over it (t_{settle}). To access the correct sector, it might be necessary to wait until it rotates under the head (t_{rotate}). The actual access to the data item is exactly the time it takes the sector to pass under the read/write head (t_{rw}). This leads to the following formula for accessing an arbitrary data item.

$$t_{access} = (t_{seek} + t_{settle}) + t_{rotate} + t_{rw} \tag{1.1}$$

The above equation is dominated by the positioning time of the read/write head. Due to the fact that the time to read an actual data item is negligibly small, reading an entire sector instead of only one item increases performance significantly.

In an operating system, this fact is modeled by the representation of a disk drive; it is a continuous set of addressable sectors, called blocks. Let us denote the fixed size of these blocks by B. All data accesses are translated into block accesses. The internal size of a block may be arbitrary, but it must be a multiple of the sector size.

This interface, the access of data blocks on a disk drive, forms the foundation for our model, i.e. we assume that the data is given as a consecutive set of addressable data blocks. Hence, the distribution problem is essentially an issue of the mapping of any number of given data blocks to a set of disk drives.

1.1.2 The Storage Network Model

We now define our model in more detail.

Definition 1 (Storage Network) *A **storage network** $S(n, C[n])$ is a collection of $n \in \mathbb{N}$ disks labeled $D_1, \ldots, D_n$ where each disk is characterised by a capacity $C_i \in \mathbb{N}$ describing the number of blocks of size B it can store.*

The disks in a storage network may have further characteristics, like the bandwidth B_i stating the average data transfer time when accessing one block. Such properties can be incorporated into our model by deriving a parameter dependent on capacity and bandwidth, e.g. the ratio of bandwidth and capacity. For the purpose of illustration, we will concentrate throughout this thesis on capacity because disk utilisation is more self-evident than throughput utilisation. The results can be straightforwardly generalised to the latter case.

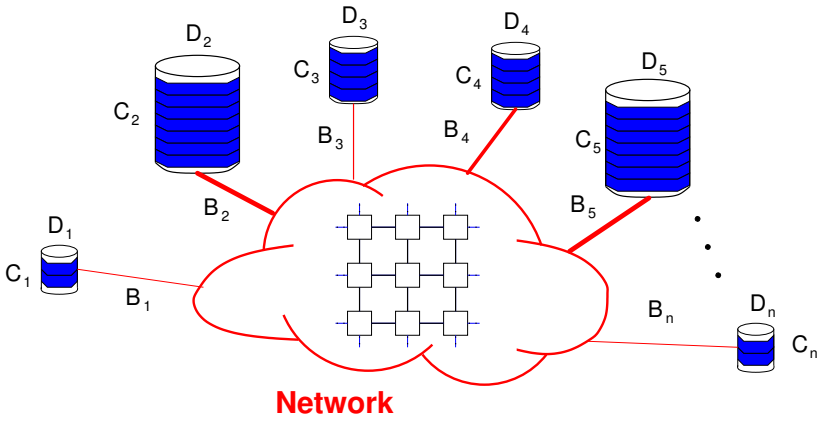

Figure 1.1: An arbitrary storage network consists of a number of disks $D_1, \ldots, D_n$ and a connecting network. Each of the disks D_i is characterised by its capacity C_i and its bandwidth B_i.

For further simplification, let us assume that the maximal number of disks is bounded by N. The capacity of the whole network can then be expressed by $C_{total} = \sum_{1 \le i \le n} C_i$.

The underlying communication network may be arbitrary as long as all disks are connected. Network specific performance measures, like congestion or latency, are not considered here. Note that the influence of the underlying network is of minor significance. The reason is the distinctive access behaviour of disk drives. It is slow compared to network performance and, therefore, forms the bottleneck in a storage network. The only way of modelling the network behaviour is by adapting the bandwidth B_i of each disk appropriately.

A storage network is called *homogeneous* (or *uniform*) if all disks have the same characteristics, i.e. the same capacity and the same bandwidth. Otherwise, the system is *heterogeneous* (or *nonuniform*).

To model the accessed data, we are given an ordered universe $U = \{1, \ldots, M\}$ of *data blocks* of size B. Let us denote the position of a data block in the universe by the *address*. The size M of the universe may be arbitrarily large but fixed. The data to be stored is given by the set $P = \{1, \ldots, p\}$ of data blocks drawn from the universe U. In general, we call all operations that read or write data blocks *data accesses* or *data requests*.

Any of the p data blocks may be completely or partially replicated. Let $r \in \mathbb{R}$ denote the *replication factor* by which all data blocks are extended, e.g. a factor of 1 indicates that every data block has another copy.

The parallel access to data blocks is modeled by the stripe concept. A *stripe* is a collection of l, not necessarily consecutive (consecutive in the defined order of data blocks), *data units*. Each data unit consists of one or more data blocks. For reasons of simplicity, we assume that each unit has exactly the size of one data block. If the data blocks are partially replicated, the stripe contains redundant blocks, e.g. parity information. The access to a stripe is fully parallel if all its units reside on different disks.

The number of data and replication blocks to be placed in the system is defined by $m = \sum_{1 \leq i \leq p} i + r \cdot i$.

Definition 2 (Data Distribution Strategy) *Given a storage network $S(n, C[n])$ a **data distribution strategy** defines a mapping of any given number of blocks $p \leq C_{total}$ drawn from the universe U to the disks such that no disk D_i gets more than C_i blocks.*

Given a replication strategy, the data distribution must also map the replication blocks to the n disks. This includes the task of mapping copies, or redundancy blocks, and data blocks to different disks. Compliance with this restriction is only an issue in randomised distribution strategies because any deterministic strategy maps full stripes, including the redundancy information.

So far, we have allowed *any* data distribution strategy to be a valid strategy. When is such a strategy a 'good' strategy? In the introduction, we have defined some requirements which we now formalise. Before we state some characteristics of good distribution strategies, we introduce some auxiliary definitions.

Definition 3 (Share) *Given a storage network $S(n, C[n])$, the **share** of a disk D_i is defined by*

$$c_i = \frac{C_i}{C_{total}} \ .$$

In other words, the share of a disk is a normalised measure of the heterogeneity in a storage network. The bigger the share of a disk, the more data blocks should be placed on that disk to ensure an even distribution.

Given the share of a disk, we are able to define the space balance property. The number of data blocks stored on disk D_i is denoted by the *load* d_i.

Definition 4 (Space Balance) *A data distribution strategy for a storage network $S(n, C[n])$ is **space balanced** if, for any given set $P \subseteq U$ of size p, every disk D_i gets at most*

$$d_i = \lceil c_i \cdot p \rceil$$

blocks. For randomised algorithms, we require that we come close to the optimal, i.e. the expected value of the number of blocks a disk must store should not exceed

$$d_i = (1 + \epsilon) \cdot c_i \cdot p \ .$$

Optimally, this property should hold for any $\epsilon > 0$.

Note that the space balance property ensures the utilisation of the storage network. It can be loaded up to almost 100% because the number of blocks any disk has to store is according to its capacity.

Fault tolerance is not one of the main issues of this thesis. Generally, an appropriate redundancy strategy (like parity) is applied and the distribution scheme ensures that the disks storing the data blocks differ from the disks hosting the redundant blocks. This could easily be implemented on top of any of the distribution strategies that we will consider.

Access efficiency measures the time and space consumption needed to access any arbitrary data block. Clearly, both should be as small as possible for good data distributions. We use common complexity measures to determine these properties. For the time complexity we count the number of computational steps needed to find the disk storing a given block b. The space complexity counts the memory resources required to do so.

So far, we are able to characterise static storage networks that do not change over time. The only dynamic aspect arises when the number of stored blocks p changes. In this thesis, we introduce a new concept for storage networks: the adaptivity to changing space requirements. We enable the storage network to increase or decrease the number of participating disks n. This imposes an algorithmic challenge if we require that the distribution strategy has to ensure the space balance property too. A natural measure for the efficiency of a distribution strategy in terms of adaptivity is the number of blocks that have to be distributed to achieve the space balance property. This determines how long the storage network must work in a degraded mode. We call all operations that result in a replacement of blocks *change operations*.

To model the dynamics in storage networks, we apply competitive analysis [BEY98]. The increasing and decreasing of disks can be interpreted as an online problem, because we do not know when a change will happen. Competitive analysis compares an online algorithm with the best possible offline strategy. In our case, the best offline strategy is a distribution algorithm that ensures space balance after each change. To do so, it has to replace at least $c_{n+1} \cdot p$ blocks. We call a distribution strategy *c-competitive* if it needs at most c times the number of replacements performed by the optimal offline strategy for any sequence of change operations.

Now we are ready to define an adaptive distribution strategy.

Definition 5 (Adaptive Distribution Strategy) *A distribution strategy is **adaptive** if it is capable of space balancing any given storage network $S(n, C[n])$ after a change in the number of disks n. It is c-competitive if the number of blocks that must be replaced to achieve space balance is only c times the number of blocks an optimal strategy has to replace. For randomised algorithms, this should hold at least for the expectation value.*

There are many different ways to analyse adaptivity. Using the share c_i of a disk is a very elegant one. Let the capacity of the storage network be given by a share vector $c = (c_1, c_2, \ldots, c_N)$ for some $N > n$. All disks that are not currently in the network have a share of $c_i = 0$. Hence, an increase in the number of disks can be modeled up to the upper bound N. Given two share vectors $(c_1, \cdots, c_N)$ and $(c'_1, \cdots, c'_N)$, the number of blocks that any space balancing strategy has to replace is given by

$$\sum_{i:c_i>c'_i} (c_i - c'_i) \cdot p \quad .$$

Note that this is equal to the number of blocks an optimal space balancing strategy has to consider.

Taking the above definitions into account, the *data distribution problem* examined in this thesis can be stated as follows: Is it possible to find an adaptive, space balancing data distribution strategy that guarantees an efficient data access and uses as few resources as possible?

1.2 Tools

The proposed strategies are randomised algorithms. For the analysis of these algorithms the following theorem is of crucial importance. It bounds the deviation from the expectation value for a number of independent random variables. In other words, it gives the tail estimation based on expectation for those independent random variables [Che52].

Theorem 6 (Chernoff bounds[Sch00a]) *Consider any set of n independent random variables $X_1, \ldots, X_n$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then it holds that for all $\delta \geq 0$,*

$$Pr[X \geq (1+\delta) \cdot \mu] \leq e^{-\min\{\delta^2, \delta\} \cdot \mu} \quad .$$

Furthermore, it holds for all $0 \leq \delta \leq 1$ that

$$Pr[X \leq (1-\delta) \cdot \mu] \leq e^{-\delta^2 \cdot \mu/2} \quad .$$

In the analysis of the strategies we use probability theory. The following definition captures the essence of high probability.

Definition 7 (High Probability) *An event X occurs with high probability (w.h.p.) if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which X occurs with probability at least*

$$Pr[X] \geq 1 - \mathcal{O}\left(\frac{1}{n^\alpha}\right) \quad .$$

CHAPTER 2

Previous Results

In this chapter, we give an overview of different approaches to tackle the data distribution problem in storage networks. Most of the previous works concentrate only on some of the necessary properties of distribution strategies. Therefore, we partition the chapter according to each of the aforementioned properties, namely space balance, availability, heterogeneity, and adaptivity. They are very general and provide the building blocks for more complex distribution schemes.

In the first section, we introduce different space balancing techniques. The main distinction is between deterministic and randomised approaches. All distribution schemes employ some of these techniques to some extent to distribute the data evenly.

The next section is dedicated to fault tolerant approaches. Because fault refractiveness is implemented by some form of redundancy (like parity information), it imposes a new challenge to the underlying space balance mechanism. In case of a disk failure, the redundant information needs to be accessed to rebuild the original content. This should be done as fast as possible because during the rebuilding time a new failure is very likely to be fatal. Hence, we introduce some redundancy balancing techniques which are tightly coupled to the fault tolerating scheme. Most of these techniques could be easily combined with existing distribution schemes to form robust algorithms that can resist one or more disk losses.

After that we consider heterogeneity, i.e. the handling of disks with different properties like capacity. There are only a few approaches that handle heterogeneity in conjunction with distribution schemes. In this section we introduce, among others, an extension to RAID level 0, which is the simple striping of data blocks over all disks, and RAID level 5, which combines simple striping with fault tolerance, that is capable of handling disks of different capacity.

The newest and, therefore, the least known field is that of adaptive distribution schemes. Even if some of the space balancing techniques are concerned with

reconstruction time, there are very few approaches that consider the dynamics of extending or reducing the number of disks in a storage network.

2.1 Space Balance

One of the major goals of storage networks, apart from providing large capacities and ensuring availability, is to decrease the overall access time. In general, this time is strongly correlated to disk utilisation if the network behaviour, as in our model, is neglected. When a disk gets too many requests, the access time (most notably the track search time) is increased significantly resulting in read/write head thrashing in the worst case. Thus, the data blocks, and with them the the data accesses, must be balanced over all participating disks. Note that this property also ensures a complete saturation of the system's capacity as long as the system is homogeneous. Furthermore, it must guarantee that data units in a stripe are hosted by different disks to ensure parallel access and fault tolerance. The access balance property describes the capability of distributing the data accesses evenly over the disk drives. It ensures an even disk utilisation if all data blocks are equally likely to be accessed.

Obviously, space and access balance are dependent. As long as we have only one copy of each data block, all requests are answered by the disk storing that copy and, hence, the space balance determines the access balance.

In general, it is easy to find worst case examples for all deterministic schemes if they are completely known. The only protection against that lies in the use of randomisation or the fine-grained distribution of data. While the first solution decreases the likelihood of hot spots, the latter fails due to the special access to magnetic disk drives. As noted before, the access to any data item smaller than a block is not efficient.

Considering our abstraction model, we are only concerned with data block accesses. In operating systems an application generates those accesses through the underlying file system. These patterns can only be predicted if there exists knowledge about the file system itself and the access pattern to each data file. Because this knowledge is hard to predict and may be subject to rapid changes, a data distribution cannot take this into account. For these reasons, we assume that all data blocks are equally likely, or we simply have no knowledge about the access patterns and, hence, try to distribute the blocks as evenly as possible.

The techniques to achieve space balance are quite simple. They can be grouped into deterministic and randomised approaches. The main deterministic technique is striping [Joh84, PGK88, SGM86, JH00]. Here, the ith data block is mapped to the disk with the label $i \pmod n$ at position $i \operatorname{div} n$. Remember that the addresses define an order on the data blocks. Given this simple scheme, full consecutive stripes (one data unit from each disk) can always be accessed in par-

allel. Furthermore, complete saturation can be achieved as long as all disks have the same capacity. There are a number of variations of disk striping. Especially in multimedia systems, it might be of interest to adjust the stripe length according to a predefined accumulated bandwidth (e.g. the streaming rate of a media object). The natural extension of simple striping is the grouping of disks into clusters of stripe length l and using simple striping within the cluster.

How can the optimal length of a stripe and the size of a data unit be determined for a given demand? In [CP90] the authors synthesise rules for deriving these parameters. They show that the size of a data unit only depends on the number of outstanding data requests and the average performance of a disk, i.e. positioning time and transfer rate.

There are a number of variations of simple striping which try to break the very regular structure of this approach. One generalisation is staggered striping [BMGJ94] that uses a logical clustering of disk drives. Instead of arranging the data units into clusters of disk drives according to a needed bandwidth, they are logically distributed over the whole network. Staggered striping defines a stride g by which consecutive data units are separated, e.g. $g = 1$ defines simple striping. This gains disk utilisation if data blocks of different bandwidth requirements are accessed.

A different approach is followed by randomised distribution schemes [AT97, Bir01, SMRN00]. In general, a data block b is placed on a randomly chosen disk $D_{h(b)}$ at a randomly chosen position. In practice, randomisation is implemented by pseudo-random hash functions or precomputed random distributions [SM98a]. This makes not only randomisation feasible but also gives an efficient access scheme; a number of lookups is sufficient. A bound for the load d_i can then be determined by the balls-into-bins model (e.g. see [RS98] for an overview). The data blocks are represented by balls and the disks are represented by bins. For the case that the number of balls p is greater than $n \log n$, the load of each bin has the order of the mean value (with high probability). In particular, there exists a bin that receives

$$\frac{p}{n} + \Theta\left(\sqrt{\frac{p \ln n}{n}}\right)$$

balls with high probability (w.h.p.). Surprisingly, the multiple-choice case, each ball has the random choice of $d \geq 2$ bins, behaves differently [BCSV00]. The authors have shown by using coupling Markov Chains that the load is independent of the number of balls p. Their simple algorithm works as follows. A number of p balls are sequentially processed. Each ball has the choice of $d \geq 2$ different bins and picks the one with the smallest load.

A slightly different approach is followed by Sanders et al [SEK00]. Their goal is to access any batch of t data blocks as fast as possible, i.e. with only t/n

parallel access operations. Given an arbitrary small $\epsilon > 0$ and a buffer space of size $\mathcal{O}(n/\epsilon)$ (up to $(1-\epsilon)\cdot n$ requests have to be handled in every access step) which hosts n writing queues, and given that every data block is replicated and put on a randomly chosen disk, any t data blocks can be processed with only $\lceil t/n \rceil + 1$ access operations. This can be achieved by the combination of a randomised distribution scheme, some replication, and an optimal scheduling of data requests. They consider a replication factor of 2 but this could be reduced to $1/r$ for $r < 10$. An optimal scheduling of n blocks is found via some maximum flow computations. Beginning with a feasible schedule, the flow network is composed of overloaded and underloaded nodes representing disks. Then, this schedule is successively improved by flipping edges.

It can be said that the simple striping techniques are sufficient to achieve an optimal space and access balance as long as the system is static (the number of disks does not change over time) and the access pattern is balanced (no hot spots). Randomised approaches possess advantages when it comes to irregular or unpredictable access patterns. Unfortunately, most of the randomised solutions work with ideal randomisation which is impossible to implement. So, there is a large demand for practical evaluations of randomised approaches. Due to their simple structure, deterministic solutions are currently favoured in practice.

2.2 Availability

Availability describes the ability to retrieve data blocks despite failed disk drives and, hence, lost data blocks. It is a crucial feature of storage networks consisting of many disks because the probability of a failure in a collection of disks increases with its size [PGK88].

The problem of availability can be solved by using redundancy. The easiest way is to store r copies[1] of all data blocks like in RAID level 1 [PGK88] or the PAST storage server [RD01, DR01]. Naturally, the distribution scheme has to ensure that the copies are hosted by different disks. The system can tolerate up to $\lfloor r \rfloor$ failures and is still operational. The faulty disks may be replaced and their contents can be rebuilt from the remaining copies.

Because a full replication scheme results not only in the necessity to update all copies in order to keep the scheme consistent but also in a large waste of resources, more space efficient methods are sought. One of the common approaches is the use of parity information like in RAID level 4/5 [PGK88, CLG+94], Random RAID [Bir01], Swarm [MH00, HMS99], and many video servers [BMGJ94, SM98a, TPBG93, VRG95, AT97]. The key idea is to use redundant information to secure more than just one data unit. This can be done by deriving the bit-wise exclusive-

[1] In the storage community, $r = 1$ is also called mirroring.

or information of a whole stripe and storing it in an extra parity unit. Assuming the units reside on different disks, one disk failure can be tolerated. When a disk failure occurs the faulty disk has to be replaced, and the unavailable data blocks must be rebuilt by accessing all remaining blocks in its stripe. During this reconstruction phase, the next failure would be hazardous. Obviously, this reconstruction should be done as fast as possible aiming to minimise the maximum amount of data one has to read during a reconstruction [HG92]. This can be done by choosing a stripe length l less than n. The resulting reconstruction load for each disk is given by the declustering ratio $\alpha = \frac{l-1}{n-1}$. If $\alpha = 1$, as in the RAID level 5 layout, each of the surviving disks participate in the reconstruction.

Using redundancy imposes a new challenge on the data distribution strategy, because it has not only to ensure an even distribution of data blocks but also an even distribution of parity information. The reason for this lies in the different access pattern for data blocks and parity units. First of all, we have to distinguish between read and write accesses. A full read, a read that requests each unit in a stripe, can be done by getting any l blocks of a stripe. In contrast, a small write, the access to some arbitrary block in the stripe, has not only to access the written data block but also the parity unit because the parity information after a write may have changed. It follows that the access to the parity unit is more frequent for writes. There are a number of different approaches to handle this situation. In RAID level 4 [PGK88] the parity units are all mapped onto the same disk. As long as the whole stripe is written, this does not impose any problems. Unfortunately, this is not always the case. Small writes put a lot of stress on the disk storing the parity information. A solution to this is provided by RAID level 5. Here, the parity units are spread over all participating disks by permuting the position of the parity unit inside each stripe. In the ith stripe the parity unit is at position $i \pmod{l}$ of that stripe.

As long as the number of disks is equal to the stripe length l this approach distributes the parity units evenly. The more general case, allowing the length to differ from n, calls for more complex distribution schemes such as parity declustering [HG92]. The authors propose the use of complete and incomplete block designs to distribute data blocks. A block design is the arrangement of v distinct objects into d blocks or tuples, each containing k elements, such that each object appears in exactly t blocks, and each pair of objects appears in exactly λ_p blocks. For a complete block design one includes all combinations of exactly k distinct elements from a set of v objects. Note that there exist $\binom{v}{k}$ possible combinations. Furthermore, only three variables are free because $d \cdot k = v \cdot t$ and $t(k-1) = \lambda_p(v-1)$ are always true.

If we now associate the blocks with the disk stripes and the objects with disks, the resulting layout, also called block design table, distributes the stripes evenly over the n disks (i.e. v equals the length of a stripe l and k equals the number

of disks n). However, such a block design table gives no information about the placement of the redundant information. To balance the parity over the whole array we build l different block design tables putting the parity unit in each of them at a different position in the stripe. The full or complete block design is derived by fusing these block design tables together. Obviously, this approach is unacceptable if the number of disks becomes large relative to the stripe length l, e.g. a full 41 disk array with stripe length 5 has 3,750,000 blocks. Hence, we have to look for small block designs of n objects with a block size of l, called balanced incomplete block designs (BIBDs), which do not need full tables to balance the parity units. There is no known way to derive such designs algorithmically, and for all possible combinations of parameters a BIBD might not be known. Hall [Hal67] presents a large number of them and some theoretical techniques to find them. In the case that there exists no BIBD for a set of parameters, the use of complete designs or the use of the next possible combination is recommended.

The approach of using parity information can be extended to tolerate more than one disk failure. As an example, the EVENODD layout [BBBM94, BBB+01] can survive two disk failures. This is accomplished by taking the regular RAID level 5 layout and adding parity information for each diagonal on a separate disk. Such an approach can be even extended to tolerate t arbitrary simultaneous disk failures. Gibson et al [HGK+94] showed that in this case $t \cdot l^{1-1/t}$ disks worth of parity information are needed. Taking fast reconstruction into account, it becomes obvious that these schemes are only of theoretical interest.

The techniques mentioned so far solve only one of the crucial problems, like availability or space/access balance. Now, we have a look at distribution schemes that take both properties into account. In [SS99, SS96] the EVENODD layout is used together with the parity declustering technique. First, an appropriate BIBD is chosen. The data units in each stripe in this layout are then replaced by a complete EVENODD layout. To do so, the data units of the BIBD are partitioned such that they host a full column of an EVENODD layout. Such a data distribution can tolerate two disk failures and allows faster reconstruction compared to RAID level 5 layouts. Furthermore, the authors reduce the problem of finding a data distribution scheme to the partitioning of the storage network into parity stripes (defining the length l and the replication factor r). This is accomplished by using a network flow based method. The idea is further extended in [SSH96] by defining approximately balanced distribution schemes. The major gain lies in the construction of such layouts. Three different construction methods are proposed: randomisation, simulated annealing, and perturbation of exact layouts. The authors implemented a simulation environment and test the behaviour in degraded mode, i.e. when one disk is faulty, and in reconstruction mode. As a result, the performance of approximately balanced layouts is very close to the performance of BIBDs based layouts.

The redundant information used to protect the storage network against disk

losses can also be exploited to decrease the data access time significantly. In the case of $r \geq 1$, a simple protocol might work as follows. Suppose there exists information about the queue length of the disk drives, e.g. gathered by probing. Because each data block has at least two copies, we always access the disk with the smallest queue length. Using the same arguments as in [RS98] it can be shown that accesses are very evenly balanced.

In [Bir01] the same technique is applied to parity extended random distribution schemes. As noted above, any $l - 1$ units suffice to reconstruct the complete stripe. Hence, a data access might choose those data units which reside on the least loaded disks. By similar arguments as in [RS98], this should improve the load substantially. Note, that some further processing is needed if the parity information is accessed. This approach is especially interesting for random distributions because it clips the tail of the disk access queue-length distribution. The author has implemented some simulations and gives evidence that such a strategy improves the response time significantly.

2.3 Heterogeneity

Heterogeneity describes the ability to handle disks with different characteristics efficiently, i.e. use the disk according to its capacity or its bandwidth. This problem is easy to solve if all other properties are neglected, but it is challenging if space balance must also be preserved.

Until recently, little attention was paid to the heterogeneity problem, e.g. software implementations of the RAID distribution schemes can handle disks with different capacities, but they simply treat each disk as if it has the capacity of the smallest disk, resulting in a waste of storage resources.

Given a number of different disk drives, an easy and quick solution to the problem is the clustering of disks according to their characteristics [GS93]. Especially in multimedia systems, this is a common technique due to varying bandwidth requirements in such systems. Within a cluster, the placement of data units can be done optimally with the aforementioned techniques (namely striping). The big advantage of this approach is its simple extendability. When new disks with new characteristics enter the system they simply form a new cluster. However, the clustering approach has a main disadvantage. Adding new (and usually faster) disks does not improve the response time of all data blocks. Only the access to data blocks hosted by newer disks are served significantly faster. This also implies that the response time depends on the cluster a data block is stored in. Assuming that the access pattern is known in advance, this could be used to avoid hot spots by putting popular data elements on faster disks. Unfortunately, the access pattern is usually unknown and may change significantly over time.

Another approach to handle heterogeneous disks was proposed by Sanders [San01]. Here, the aim is to design optimal scheduling algorithms for arbitrary data requests. Each data block is replicated and randomly mapped to different disks. Then the scheduling problem that arises is transformed into a network flow problem. The requests are on the left hand side (source) and have an edge (with weight ∞) to all disks which store a copy of the requested data block. On the right hand side of the flow network (sink), the disks might be served by disk controllers and I/O-busses each of which is modelled by a node and an edge with an appropriate weight to the disks/controllers it is connected to. Now, the flow problem is solved resulting in a scheduling algorithm for the data requests. Heterogeneity is introduced by adjusting the edge weights of the flow network according to the given capacities. The disadvantage of this approach lies in its batch-like behaviour. For any collection of new requests, a new flow problem must be solved. Especially for large systems, the number of needed requests to achieve good results is considerable.

AdaptRaid [CL00, CL01] is a distribution scheme that extends the general RAID layout so that heterogeneity, in terms of different capacities, can be handled. The basic idea is very simple. Larger disks are usually newer ones and, thus, should get more data blocks. Nevertheless, the placement of whole stripes should be kept as long as possible to gain from parallelism. So far, there are extensions to handle the case without data replication (RAID level 0) and with fully distributed parity information (RAID level 5).

The initial idea of AdaptRaid level 0 [CL00] is to put as many stripes of length $l = n$ on all disks as possible. As soon as some k disks cannot store any more data units, stripes of length $n - k$ are mapped onto the remaining $n - k$ disks. This process is repeated until all disks are saturated.

This approach has the major drawback that the access to data stored in the upper part of the disk is faster than the access to data in the lower part of the disk drives because of the different stripe length. Furthermore, the assumption that newer disks are faster is not generally true. The following generalisation copes with both problems. First, we define the utilisation factor $UF \in [0, 1]$ as the ratio between the capacity of a disk and its bandwidth. This factor is determined on a per disk basis, where the largest disk always has $UF_i = 1$, and all other disks get a UF according to their capacity relative to the largest one. These values should be set by the system administrator. To overcome the access problem, data patterns are defined which capture the overall capacity configuration. These patterns have a similar structure to the patterns of the initial idea, but they are kept smaller such that many of them can fit on a single disk. The size of these patterns is measured by the second parameter, namely lines in a pattern (LIP). In general, this parameter is an indicator of the distribution quality of the layout and is measured by the number of units in the pattern that are mapped to the largest disk. The overall data layout is defined by repeating these data patterns until the disks are saturated.

The access to data blocks is twofold. In the first phase, the correct pattern has to be found. This can easily be done because the size of a pattern, i.e. the number of units it contains, is known. Then, the requested data unit is found by accessing the pattern itself. The space resources needed to access any data element is proportional to the size of the pattern.

The performance of AdapRaid0 is tested using a disk array simulator. The tests are conducted with varying ratios of fast and slow disks[2]. Two different scenarios are tested. First, the performance is compared to a RAID level 0 layout, which treats all disk equally and using only small portions of the faster ones. Naturally, AdaptRaid0 can shift the load from slower disks to the faster ones resulting in a performance gain of 8% up to 20% for read accesses and 15% up to 35% for write accesses. The second scenario compares AdaptRaid0 to a configuration that contains only fast disks. This is reasonable because it gives an evaluation of the usefulness of keeping older disks in the system and, therefore, measures the impact of parallel disk accesses. The experiments show that the performance gain is more pronounced in read accesses, e.g. in an array with 2 fast and 6 slow disks the read accesses can be answered 10% faster than in a fast disk only configuration.

It can be concluded that the exploitation of heterogeneity gains a significant performance speedup.

AdaptRaid level 5 [CL01] is the heterogeneous variant of RAID level 5. It uses the same simple idea as AdaptRaid0, mainly to put more data units on larger, usually faster, disks. The distribution problem is more complex, because to guarantee a fast reconstruction time, the parity information must be evenly distributed as well.

Following the same approach as in AdaptRaid0 and keeping the parity distribution of RAID level 5 results in a small write problem. Such a layout has stripes of different lengths. From the distribution point of view, this is not a problem, but when it comes to performance the situation is different. Note that the parity unit needs to be updated every time a data unit is written. Hence, file systems should group the data requests such that full stripes could be written as often as possible. Such a technique avoids many small writes and works well when the stripe size is fixed. Unfortunately, this is not the case here and, therefore, such optimisations would be useless for the proposed data layout.

This problem can be avoided by restricting the stripe length of each stripe to integer divisors of the length of the largest stripe. This leads to a heterogeneous layout which may waste some capacity. However, a careful distribution of the free units significantly improves this scheme. First, the layout should be independent of the number of disks still participating. This can be achieved by letting each

[2]The faster disks possess roughly double the performance parameters, such as average seek time, short seek time, and local cache.

stripe start on a different disk. The welcome effect of this transformation is the even distribution of the wasted free space over all disks, instead of concentrating them on a few disk drives. Now we can get rid of the wasted capacity by using a 'Tetris'-like algorithm. The holes of free units are filled with subsequent data units from stripes further below; data units from these stripes 'move' upwards. Because the free spaces are distributed over all disks, whole stripes can be added at the end of the pattern. The layout is still regular, and only the size of one pattern is need to locate any data unit. As in the AdaptRaid0, we generalise the solution by defining two parameters. As before, *UIF* is the utilisation factor describing the load of a disk related to the largest disk. Similar to *LIP*, the parameter *SIP*, stripes in pattern, controls the distribution of patterns over the whole array.

For the practical evaluation, the same simulator as for AdaptRaid0 is used. The new approach is tested against a RAID level 5 and a homogeneous configuration of only fast disks[3]. Obviously, both approaches waste some capacity in one way or another. The performance is measured for reads, small writes and full writes. It could be observed [CL01], that AdaptRaid5 is almost always the best choice and scales gradually with an increasing number of fast disks. Only when large chunks of data are read, the method is outperformed by the homogeneous configuration of only fast disks. This is due to the fact that the slower disks cannot deliver the data units appropriately. When full stripes are read, they form a bottleneck in the system. Nevertheless, when simulations of real workloads are used, the performance gain of the new approach is around 30% if only half of the disks are fast disks.

A different kind of heterogeneity is used in the disk merging approach [ZG00]. Here, the aim is not only to provide heterogeneous bandwidth distributions but also to increase the reliability of the system.

The disk merging approach constructs a homogeneous, logical view out of a physical collection of disks. Each physical disk D_i is partitioned into $k_i \in \mathbb{R}$ equal sized *extents*. These extents are arranged into G parity groups, the logical view of the system. The parity groups themselves are homogeneous and consist of l extents each. It is possible that an extent gets capacity and bandwidth from different physical disks. The data distribution scheme simply stripes the data units over the parity groups.

How many parity groups are possible? In order to ensure the fault tolerance of the system enforced by the striping strategy used, each extent in every parity group has to come from a different physical disk. Let D^t denote the number of logical extents in the system. Then, the number of parity groups has to fulfil the following inequality:

$$G \leq \left\lfloor i \frac{D^t}{k_i} \right\rfloor \qquad 0 \leq i \leq n$$

[3] In the experiments, the number of fast disks varies from 2 to 9 disks.

To show the reliability of the system, the behaviour is modelled by different processes like the failure process and the reconstruction process. Because the mean time to failure of any disk can be estimated, the time of these processes can be estimated too. Using a Markov model [Gib91], the mean time to service loss (MTTSL) is derived for such a heterogeneous system. It is shown that the reliability differs from the best possible configuration, namely the clustering of identical disks, by a factor of about 10.

Nevertheless, this approach still needs the experienced hand of an administrator and the performance strongly relies on his decisions.

Randomisation alone can also be used to handle heterogeneity. The initial idea is to put more data units on larger disks. This is done as follows. Recall that $c_j = C_j/C_{total}$ is the share of disk D_j. Note that $\sum_{1\le j\le n} c_j = 1$. Now, c_j is used as the probability of putting a data unit on disk D_j. In the long run, this would roughly saturate the disks according to their capacity. Nevertheless, there might be significant short time imbalances.

The RIO media server [SM98b, SM98a] uses the idea mentioned above. It is a generic multimedia storage system capable of efficient, concurrent retrieval of different types of media objects. Furthermore, it supports real-time data delivery with statistic response time guarantees. The randomised distribution scheme that is used is very simple. The data units are placed on a randomly chosen disk at a randomly chosen position[4]. Heterogeneity may occur in different disk capacities and varying bandwidth properties. As noted before, randomisation ensures a good balance in the long run, especially when it is used in conjunction with replication. In addition, RIO also exploits redundancy to improve the short time balance by carefully scheduling the data requests.

Is it possible to optimise both capacity and bandwidth simultaneously? The simplest way is a coupling of bandwidth and capacity by defining the bandwidth to space ratio $BSR = b_i/C_i$. The higher the BSP the faster the disk can deliver all its data units, hence, the more data units should be mapped to these disks. Obviously, the disks with low BSR present the bottleneck in the system. The main idea for a better balance is to shift some of the load from disks with low BSR towards disks with high BSR. This can be done using replication and putting some copies on disks with high BSR. However, how much redundancy is needed to sustain a given load? This question can be answered by using the following approach. First, the disks are grouped into t clusters according to their BSR and sorted in ascending order. Let N_i be the number of disks in cluster i. Further let $S_i = \sum_{1\le j\le N_i} C_i$ and $B_i = \sum_{1\le j\le N_i} b_j$ be the accumulated capacity and bandwidth in cluster i, respectively. Let S and B define the total capacity and bandwidth in the storage network and let $S'_i = S_i/S$ and $B'_i = B_i/B$ be the relative capacity and

[4]The latter is only important for the theoretical analysis. A practical implementation can only ensure pseudo-random behaviour.

relative bandwidth of cluster i, respectively. The relative bandwidth space ratio can then be defined as $BS_i = B'_i/S'_i$. For a homogeneous system, $BS_i = 1$. The cluster with the lowest BS_i is the most stressed cluster, not only because its disks have the slowest bandwidth but also because they host a large proportion of the data units. The number of data units that can be stored in the system is defined by $U = S/(1+r)$ where r denotes the replication rate. In formal terms, we want to use replication to shift the stress from clusters with $BS_i < 1$ to clusters with $BS_i > 1$.

How much replication is needed to sustain a certain maximal load $\lambda_{max} \leq B$? To answer this question, we have to distinguish between two different cases. Let $\rho = \lambda_{max}/B$ be the average disk utilisation. If $\rho = 1$ we have full utilisation and if $\rho < 1$ the disks are only partially utilised. Let's consider full utilisation first. It can only be achieved, if the load directed to a cluster i is of the order of its bandwidth B_i. The maximum load that can be directed to any cluster i is reached when all data units stored in this cluster are distinct and all accesses to these data units are sent to i. Hence, the maximal load can be computed by $\lambda_i = \lambda_{max} \cdot \frac{S_i}{U} = \rho \cdot B \cdot (1+r) \cdot S'_i$. To use all the available disk bandwidth on the system, we have to fulfil $\lambda_{i_{max}} \geq B_i$. This is satisfied if $BS_i \leq (1+r)$ for all $1 \leq i \leq m$. It follows that the replication rate can be expressed by $r \geq \max_i\{BS_i\} - 1 = BS_m - 1$.

In a similar way, one can determine the minimum replication rate to ensure partial bandwidth utilisation. Furthermore, it can be shown that this replication rate is sufficient to achieve the desired utilisation by defining a distribution scheme mapping replicated data units to clusters according to their BSR.

2.4 Adaptivity

Adaptivity describes the ability to react to changes in the configuration of a storage network efficiently. By changes we mean all modification induced by the addition or removal of disks or any other change in the capacity distribution. We use the number of replaced data blocks needed to ensure space balance as a measure for the efficiency of adaptivity. Note that adaptivity without other restrictions does not make sense.

Adaptive distribution schemes are relatively new. A first but very limited approach is introduced in software RAID systems [Cor99]. Here, spare disks are used to counteract the higher risk of disk failures in storage networks. The storage network is extended by a number of unused spare disks which take over when a disk fails. Furthermore, the distribution scheme has to possess some redundancy because the contents of the failed disk has to be reconstructed on the spare disk. Such an approach has two main drawbacks. First, there is some capacity in the system that is not used, namely the spare disks. Second, it does not ensure the expandability of the system due to growing storage demands.

A different approach is proposed by Leighton et al [KLL+97, Lew98]. This randomised method maps every block into a real $[0, 1)$ interval using a random hash function h. Furthermore, for each disk D_i, k real number, called disk points, are mapped into the same interval by the random hash functions $g_1, \ldots, g_k$. A block b is assigned to disk D_i if its hash value $h(b)$ is closest to any of the hashed disk points $g_j(i)$ for $1 \leq j \leq k$. It is described in more detail in Section 3.1.1 because we need this strategy later in Section 3.2.2 Despite its generality, this approach is not able to handle heterogeneity efficiently. The obvious extension would be to change the number of disk points k such that the number of disk points a disk gets is according to its capacity. However, this would require an increase in the number of points to

$$\Omega\left(\min\left\{\frac{C_{max}}{C_{min}}, p\right\}\right)$$

where C_{min} is the capacity of the smallest disk in the system and C_{max} is the disk with the largest capacity, in order to ensure that after every change operation the space balance property is still met. Thus, in the worst case, the number of disk points would be proportional to the number of blocks in the system. On the other hand, if we restrict the number of disk points to be significantly below p the strategy cannot guarantee the space balance property for all possible capacity distributions. To see this, just consider two disks D_1 and D_2 with $C_1 = c/p$ and $C_2 = (p - c)/p$ for some constant $c > 0$. In that configuration we need $\Theta(p)$ disk points to guarantee space balance.

Sanders [San01] follows a different approach. Consider a system of n parallel disks and a continuous stream of data requests for data blocks stored on the disks. Furthermore, the data blocks are replicated in some form. For the mapping of data blocks and their copies, a randomised approach is used, i.e. each data block chooses a disk uniformly at random, such that neither two copies nor two blocks in a parity stripe reside on the same disk. The question to be answered is the following: How can a number of N arbitrary reading requests be scheduled such that the number of parallel I/O operations[5] is minimised? In an earlier work [SEK00], it is shown that it is possible to find a schedule in almost every round for $(1 - \epsilon) \cdot n$ new data blocks if we allow n write queues, one for each disk, with an overall consumption of $\mathcal{O}\left(\frac{n}{\epsilon}\right)$. The authors use a flow network approach. In this network, the source is connected to all nodes representing a data block with an edge of weight 1. All request nodes have an edge to a disk node if the requested data block is stored on this disk. Note that due to the used replication each request has more than one outgoing edge. To find an optimal schedule, a sequence of maximum flow computations need to be solved.

[5] Note that we can read one data block from each disk in parallel with only one parallel I/O operation.

In [San01], this model is extended to handle more complex scenarios, which include heterogeneity in form of physical disk access structures, disk failure, and dynamic introduction of spare disks.

Heterogeneity is dealt with in two different ways. First, the randomised data placement is extended such that the probability of a data block being placed on disk D_i is biased with the capacity of that disk. As long as the number of data blocks is large enough, each disk stores a share of the blocks according to its capacity. The second kind of heterogeneity is introduced in the flow network determining the schedule of requests. Instead of connecting every disk directly to the target node, an arbitrary hierarchy of access stages, like I/O controllers, I/O busses and the like, are introduced. Hence, not only the congestion at the disks but also possible communication bottlenecks in the physical access path could be modelled.

The approach is also capable to adapt to changes. The system is able to handle spare disks and so-called virtual spares. It is shown that the restricted data placement, where different copies reside on different disks, decreases the quality of the scheduling in such a way as if it would consist of $n - 1$ instead of n disks w.h.p. When a disk fails, a spare disk can take over storing all the reconstructed information from the failed disk. Unfortunately, this method does not take into account the capacity of the new disk. For example, consider the following scenario. Disk D_f is failing and has a capacity of C_f. It would be replaced with space disk D_s with capacity C_s. As long as the capacities are different, the resulting distribution is skewed because D_s does not get a share according to its capacity C_s. Hence, the space balance property is violated, and an almost complete utilisation of the system can not be achieved.

Should one not be willing to waste the capacity of spare disks, it is possible to redistribute the data blocks from the failed disk D_i over all remaining disks using so-called *forward hash functions* f_i. They distribute the blocks of a failed disk evenly over all remaining disks. However, this method has a main drawback. The more disks fail, the more complex the scheduling gets. This is due to the fact that each time a disk fails the flow network needs to be adjusted to the changed positions of the data blocks.

To summarise, this approach has limited capabilities to handle heterogeneity and adaptivity. It is able to use randomisation to distribute data blocks over different sized disks. Furthermore, it can model bottlenecks other than disks only. With respect to adaptivity, only the failure of disks is supported. It is not possible to adapt the disk storage to changing demands. Furthermore, to schedule a batch of data requests, one has to build a specific flow network and solve a number of maximum flow problems. This does have implications on practical implementations. Taking standard state-of-the-art Storage Area Networks (SAN), a data block can be accessed within milliseconds. This would set a very demanding time window for the computation of the schedule.

2.5 Complexity Issues

To finish the previous results, we ask a question that should be the starting point of every serious research. What is the complexity of the data placement problem in general? The answer to this question highly depends on the optimisation goal behind the placement. The previous sections defined some of the goals we want to achieve. However, we like to go back to very basic properties, like disk utilisation and load utilisation. Furthermore, we simplify the problem significantly by assuming that all accesses are known in advance. Surely, we are not able to find such information on the level of data blocks. This model is more interesting when larger multi-media objects, like complete films, are concerned because this information is related to the number of clients who want to access a data object.

One should note that the space utilisation alone is easy because taking any free position on any disk suffices. The task evolves into a problem as soon as we demand another property, e.g. load restrictions for each disk. This problem is investigated as the *data placement problem* in [ST01, ST02, GKK+00]. You are given N disks with a storage capacity C_i denoting the number of objects it could store and a load capacity L_i denoting the number of accesses it could sustain, for all $1 \leq i \leq N$. Furthermore, all accesses to the objects are given in advance. The data placement problem is defined by maximising the number of accesses without violating the capacity and load restrictions for each disk. Note that a disk can store a data object only once but is able to sustain up to L_i accesses[6]. Shachnai and Tamir show that this problem is NP-hard and proposed a simple algorithm, called sliding window, that is a factor of $\frac{c}{c+1}$ away from the optimal, where c denotes the number of objects stored on each disk ($C_i = c$ for all i in the uniform case). Golubchik et al [GKK+00] make this bound tight and show that only a fraction of $\frac{1}{(1+\sqrt{c})^2}$ is not placed using the sliding window algorithm. Furthermore, they find a polynomial reduction to the 3-PARTITION problem [GJ79] and conclude that the data placement problem is NP-complete for $c = 2$ and NP-hard for $c \geq 3$.

The mapping of data objects to disks is not the full monty. The access pattern may change due to changed preferences, disks may fail, or new disks may be added. All these reason lead to a new mapping and the question of transferring one configuration into the other. Obviously, the number of migrations, the movement of a data object from one disk to another, should be minimised. However, not only the migration volume is important. A very interesting question in that context is the associated migration plan or schedule. In other words, which disk sends/receives which data object to/from which disk in which round of the migration process. Hall et al [HHK+01] considered the problem of finding an optimal movement of data objects. They model the problem with a demand graph G in

[6] This is only true if the load capacity L_i is chosen according to the sustained access rates and not according to the maximal (sequential) access performance.

which each node corresponds to a disk and two vertices u and v are connected by a directed edge (u, v) if u has a data object that v needs. At any given time, each disk can only send or receive one data object. Furthermore, all disks are connected and all data objects have the same fixed size. Now, one could easily divide any schedule into stages where each of them consists of a number of compatible sends. Thus, the data migration problem is exactly the multigraph edge colouring problem and, hence, NP-complete [HHK+01]. Nevertheless, there are good approximations.

Due to the send/receive rule, the maximum degree Δ over all nodes gives a lower bound on the number of stages needed. Three different scenarios are investigated. The first must send the data objects directly to its targets while the second scenario may send it to some intermediate nodes, called bypass node, before forwarding the data object to its target. Clearly, the number of bypass nodes should be small. In the strongest version, the free memory of each node is restricted. In other words, it is not possible that the number of incident in-edges minus the number of incident out-edges exceed the free space of the device. They found migration plans that need $4 \cdot \lceil \Delta/4 \rceil$ stages and $n/3$ bypass nodes for the space restricted case. In the case that no bypass nodes are allowed, their best algorithm needs $6 \cdot \lceil \Delta/4 \rceil$ stages.

Khuller at al [KKW03] take the data migration one step further by incorporating the possibility of generating new copies. That cloning can help the migration is very easy to see. Consider the lower bound of Δ, i.e. the largest degree of a node in the demand graph. If one allows the copying of data objects this bound does not hold anymore. To see this, consider a node v with out-degree Δ. In the former model, this node has to send a data object to each of the adjacent nodes $u_1, \ldots, u_\Delta$. Let v send the data objects in this order. At any time $t \geq 2$, node v is not obligated to send to all adjacent nodes anymore because all nodes u_j, for which $j < t$, have the objects and can copy and send them too. Note that this is essentially different from bypassing nodes.

With the cloning property the migration problem is related to broadcasting and gossiping problems. In a broadcast, one node has an object that it wants to communicate to all other nodes. The more general form of a broadcast is the gossiping problem where each node has an object that it wants to convey to all other nodes. There are two main differences from the migration problem. First, each node has to communicate an object only to a subset of all nodes. Second, each node is in the possession of more than one object. Of course, the migration problem with cloning is NP-complete. This can be shown by a reduction to the edge colouring problem [KKW03].

The authors design (a rather complicated) approximation algorithm that achieves a 9.5-approximation for the data migration problem with cloning. Their algorithm solves a number of flow problems. Let β be the maximal number of objects a disk gets and let α be the number of objects a node is source of as well as destination to. Then a unique source is chosen for each object such that α is

minimised. Using this source, they define transfer graphs distinguishing between objects needed by more than or equal and less than β disks. In a final step, the edges of this graph are coloured. This can be done because the out-degree of each node is bounded.

There are some practical works concerning migration [AHH+01, CGL99]. In [KKW03] the authors describe some preliminary performance results. They found that their algorithm is inferior to simple matching heuristics if the sources and destination nodes are distributed according to a Zipf distribution. In [CGL99] the authors discuss some interesting aspects of migration such as tradeoffs between using resources for replication and using resources for servicing requests. They raise some interesting questions but do not give sufficient material to propose a solution. The most comprehensive practical evaluation is done in [AHH+01]. The authors compare the performance of five different migration algorithms, two new ones (Greedy-Matching and Max-Degree-Matching), and three proposed in [HHK+01] (2-factoring, 4-factoring direct, and 4-factoring indirect). Both of the new algorithms find their solution via a number of maximum matchings. While the latter finds an optimal migration plan, nothing is known about the worst-case bounds of the Greedy-Matching algorithm. All algorithms are tested with the same test graphs that come from 4 different classes. First, they considered load-balancing graphs which are generated using real world applications (data warehousing application). The second class of graphs consisted of randomly chosen general graphs, i.e. the graphs are generated by taking n nodes and picking m edges from all possible edges uniformly at random. This class is used to study the varying density of demand graphs. The third class are regular graphs, i.e. every node has degree d. To generate these graphs, $d/2$ randomly chosen 2-regular graphs over n vertices are edge joined. The last class consists of Zipf graphs. Any graph in this class has minimum degree d_{min} and the node degrees follow the Zipf distribution, i.e. the number of nodes with degree d is proportional to $1/d$.

For load-balanced graphs, they discover that the number of bypass nodes is always far less than the theoretical bound even for large graphs consisting of up to 300 nodes. The newly proposed Max-Degree-Matching needs almost no bypass nodes at all. Furthermore, the theoretically unbounded Greedy-Matching approach performs better than, in terms of the number of stages, the 4-factoring algorithm. For the random graphs, they test against varying the number of nodes and edges. In all experiments the number of stages needed increases very slowly with increasing density, number of nodes, and degree. Nevertheless, they are always much lower than the theoretical bounds suggested. In particular, Max-Degree-Matching uses almost always less bypass nodes than 2-factoring and the Greedy-Matching always takes less stages than 4-factoring. For the number of bypass nodes it is interesting to note that their number is 10 times smaller than the theoretical bounds suggested.

All the research mentioned above is related to our approach insofar, as we

face similar obstacles. The requests to *any* arbitrary number of data blocks has to be evenly distributed over all participating disks in the network. We have to take this more conservative approach because we have no knowledge about the access pattern generated by higher applications. However, as we have seen, even this additional information leads to NP-hard problems if we are interested in the optimal solution. Hence, the use of randomisation is more than justified.

The same is true for adaptivity. The addition and removal of disks results in a change of access pattern. In the migration problem, objects need to be replaced to match changed demands, whereas we try to keep an even distribution to enhance as many parallel accesses as possible. Further, the migration research suggested that it is a difficult problem to find an optimal migration plan supporting the use of randomisation even further.

CHAPTER 3

Adaptive Data Distribution Strategies

The field of adaptive data distribution strategies is rather new. As noted in the last chapter, there are only a few algorithms which can handle adaptivity in a uniform setting well (see also [Sal03]). In this chapter, we introduce and analyse our new approaches. We start with data distribution strategies for uniform storage networks, because homogeneity is much easier to handle. The first strategy is the Nearest Neighbour strategy [KLL+97], also known as consistent hashing. Because it is used as a subroutine in a later algorithm, we state its main properties and give sketches of their proofs concerning our desired properties. We finish the first part by presenting the Cut-and-Paste strategy [BSS00]. This simple algorithm is able to achieve a distribution that is superior to that of the Nearest Neighbour strategy. We show that the quality of the distribution only depends on the hash function used. Furthermore, the number of computations needed to access an arbitrary is always logarithmic in the number of disks.

The second part of this chapter introduces adaptive distribution algorithms for heterogeneous storage networks. We begin with the natural extension of the Cut-and-Paste, namely the Level strategy introduced in [BSS00]. It is capable of handling heterogeneity, but the analysis shows that there are some effects that prevent a good competitive ratio.

This was good reason to propose two more strategies, namely Share and Sieve [BSS02], that overcome the deficiencies of the Level strategy and define the first practical approach to solve the adaptive data distribution problem for heterogeneous storage networks. Share works in two phases. The first phase transforms the problem into a number of homogeneous ones. In the second phase, any uniform strategy with certain properties can be used to solve the generated uniform subproblems. The Share strategy shows not only good theoretical behaviour but

is also easy to implement. Hence, we chose it for our implementation given in the next chapter.

In contrast to Share, the Sieve strategy relies heavily on random hash functions. As the name suggests, it works like a multistage filter. Each stage needs a new hash function and maps a number of blocks to the disks. The number of stages is logarithmic but it must be extended additionally by f the quality parameter to guarantee to adaptivity. We show that its adaption and distribution quality are very good, but the extensive use of randomness reduces its practical value.

3.1 Strategies for Homogeneous Networks

We begin the discussion of adaptive distribution strategies by analysing the uniform case. It is significantly easier because we know in advance which fraction of used blocks a new disk gets and how many each of the disks has to supply to achieve good adaptivity. In this setting, all disks of the storage network have the same properties, i.e. $C_i = C_j$ for all i and j. Note that we restrict the heterogeneity to capacity only.

In the following, we introduce two uniform strategies: the Nearest Neighbour strategy and the Cut-and-Paste strategy and analyse them regarding adaptivity and space balance.

3.1.1 Nearest Neighbour Strategy

The Nearest Neighbour strategy was introduced by Karger et al [KLL+97] as consistent hashing. It is part of a global distribution scheme for web-based contents and is described in more detail in [Lew98] and applied to Web caching in [KSB+99].

Like most of the adaptive distribution strategies, the Nearest Neighbour strategy relies heavily on random hash functions. To simplify the description, we define by N the maximum number of disks in the system. At any given point in time, there are n disks in the network. Let M be the size of the universe U from which the blocks are drawn.

The strategy works as follows. Suppose we are given a random hash function f_B and a set of independent, random hash functions $g_1, \ldots, g_k$, where k may depend on n. We use the first function $f_B : \{1, \ldots, M\} \to [0, 1)$ to map the blocks uniformly at random to real numbers in a $[0, 1)$ interval, and each of the k functions $g_j : \{1, \ldots, N\} \to [0, 1)$ assigns a real number from the same $[0, 1)$ interval to each of the disks. The hash value $f_B(b)$ of block b is called *block point*, while the hash value $g_k(i)$ of disk D_i is termed *disk point*.

For the assignment of blocks to disks, we view the $[0, 1)$ interval as a ring. A disk D_i stores all blocks b that have a hash value $f_B(b)$ that is closest to any of the

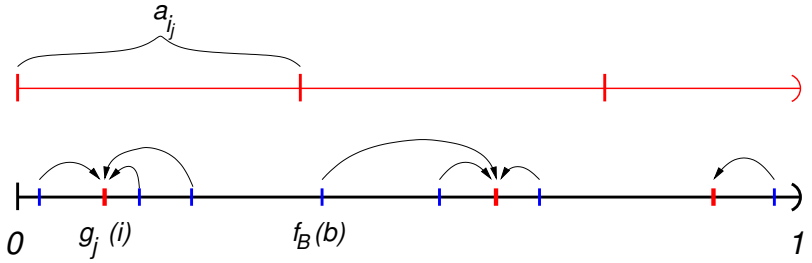

Figure 3.1: The general approach of the Nearest Neighbour strategy.

disk points of D_i, i.e. it stores all blocks b for which

$$\min_j \min\left\{|f_B(b) - g_j(b)|,\ 1 - |f_B(b) - g_j(b)|\right\} \tag{3.1}$$

is minimised. In other words, each disk point $g_j(i)$ defines an arc a_{i_j} for which the disk D_i is responsible, i.e. it hosts all the blocks falling into that arc. The mapping of blocks to disks is depicted in Figure 3.1.

Given the above strategy, we can compute the position of an arbitrary block b very efficiently as stated by the next theorem.

Theorem 8 *[KLL+97] The Nearest Neighbour strategy can access an arbitrary block in expected constant time if each disk uses $k = \Theta(\log n)$ disk points.*

Proof (sketch). We can show this theorem by constructing a data structure that achieves this bound. To simplify the description, we start with a structure that needs $\mathcal{O}(\log n)$ computations to get the position of b. Note that we have $c \log n$ arcs per disk where each arc represents a subinterval of $[0, 1)$. So we can employ a balanced tree with $c \cdot n \log n$ leaves storing the arcs. Hence, the height of the tree is bounded by $\mathcal{O}(\log n)$. To access some block b, we only need to find the corresponding interval in the tree.

To get an expected constant time, we divide the $[0, 1)$ interval into roughly $c \cdot n \log n$ segments of equal size. For each segment we use a separate search tree to handle all arcs that fall within it. To access the position of b, we have to find the right segment that the appropriate block point falls into (can be done in time $\mathcal{O}(1)$) and then query the attached search tree for the correct interval. The latter takes only $\mathcal{O}(1)$ computations because the number of intervals in each segment is expected to be constant. The problem of updating the search trees can be reduced by using only arcs of size $\frac{1}{2^x}$ for some $x \in \mathbb{N}$. During the process of adding new disks, arcs are bisected, amortising the induced overhead gradually over all operations. ■

Theorem 9 *[KLL+97] The Nearest Neighbour strategy is space balanced for $k = \Omega(\log n)$ for any p blocks drawn from the universe U. The strategy can guarantee that every disk gets $\mathcal{O}\left(\frac{p}{n}\right)$ blocks w.h.p.*

Proof (sketch). We sketch only the idea of the proof here, because it follows straightforwardly from the proofs given in [KLL+97].

As can be seen in Figure 3.1, each of the disk points $g_j(i)$ are responsible for an arc a_{i_j}. The number of blocks mapped to disk D_i is the sum of all blocks that fall into one of the k arcs of disk D_i. Now it is easy to compute the probability that an arc has a certain length, with regard to the hash functions g_i. To show that the Nearest Neighbour strategy is space balanced, we simply derive the number of blocks that fall into all the arcs of disk D_i. It can be shown that $k = \Omega(\log n)$ suffices to achieve high probability. Obviously, the space balance is better the larger one chooses k. ∎

Before we analyse the adaptivity, we have a closer look at the dynamic behaviour of the Nearest Neighbour strategy. When the number of disks changes from n to $n+1$, k new disk points are mapped into the $[0,1)$ interval decreasing the size of at most $2k$ arcs. All blocks for which Property (3.1) changes are mapped to the new disk D_{n+1}. We have to distinguish two situations in which blocks are replaced. The first, as noted above, arises when the number of disks changes from n to $n+1$. However, we also have to ensure that the number of disk points is always at least $\Omega(\log n)$. The latter problem could be tackled by using $c \cdot \log N$ points in the beginning, where N should be the expected size of the network. Then, as long as $n \leq N^c$, we do not need to increase the number of disk points.

$$\begin{aligned} (N)^c &\geq n \\ 2^{c \cdot \log N} &\geq 2^{\log n} \\ c \cdot \log N &\geq \log n \end{aligned}$$

The above reasoning results in the following lemma.

Theorem 10 *[KLL+97] The Nearest Neighbour strategy is 1-competitive in the expected case for $k = \Omega(\log N)$.*

Proof (sketch). We can apply the same arguments as in Theorem 9. Whenever we map new disk points into the $[0,1)$ interval, the hash functions ensure that the sum of the arcs are roughly the same. To prove the theorem, we need to ensure that blocks are only mapped to newly inserted disk points. Obviously, this is the case since the only blocks that are replaced are those for which Property (3.1) changes, and that happens only if a new disk point, which is closer to to some old disk point, is inserted. ∎

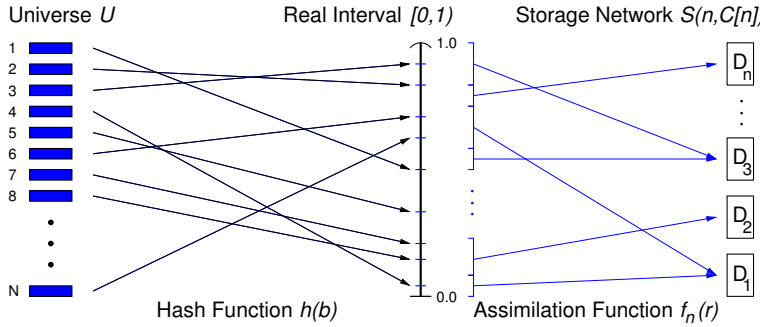

Figure 3.2: The general approach of the Cut-and-Paste strategy.

Looking closer at the Nearest Neighbour strategy, some properties become apparent. First of all, we need a rather large number of independent hash functions to achieve a good space balance. The parameter k can quickly grow up to a thousand, increasing the overall resource consumption significantly. Furthermore, the size of the arcs associated with any disk D_i is not equal for all disks, i.e. the distribution quality does not only depend on the quality of the hash functions g_i but also on the parameter k. Due to that fact, the strategy can only guarantee a space balance of $\mathcal{O}\left(\frac{p}{n}\right)$. As we will see in a later chapter, this is an issue in practice.

3.1.2 Cut-and-Paste Strategy

The Cut-and-Paste strategy was introduced in [BSS00] and aims to improve on the distribution quality compared to the Nearest Neighbour strategy. It works in two phases. In the first phase, the data blocks, drawn from the universe U, are hashed into the real interval $[0,1)$ using the pseudo-random hash function $h(b) : U \to [0,1)$. Whether or not the blocks are evenly scattered over the whole interval depends only on the quality of this function. In the second phase, an assimilation function f_n cuts the interval into ranges $r = [lb, ub)$ and assigns them to the n disks of the storage network. All blocks b whose hash value $h(b)$ falls into range r are mapped to the disk r is assigned to (see Figure 3.2).

The assimilation function f_n has to ensure that all disks get the same share of the $[0,1)$ interval. Furthermore, the mapping establishes that only the optimal number of blocks are replaced when a change in the number of disks occurs. To simplify the description, let's denote by $r_i = [lb, ub)$ a range mapped to disk D_i. When more than one range is assigned to a disk, they are fused together to form the range $r_i = [0, 1/n)$ of size $1/n$. Note that we are in the uniform setting and, hence, each disk should get an equal share of ranges.

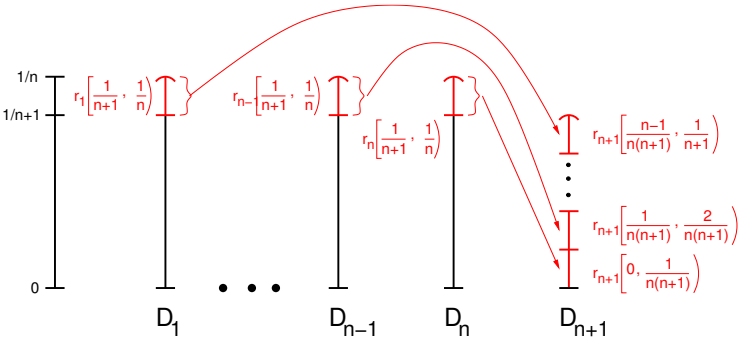

Figure 3.3: The transition from n to $n+1$ disks. The uppermost part of all disks is cut and concatenated to the new local range of disk D_{n+1}.

We describe the assimilation function from the adaptive viewpoint, i.e. characterise its behaviour in the case of a change in the number of disks. Suppose, the number of disks changes from n to $n+1$. Then, f_n performs the following transition. It cuts off the highest part of each local range, namely $r_i = \left[\frac{1}{n+1}, \frac{1}{n}\right)$ $(i \leq n)$ and concatenates them to the new local range $r_{n+1} = \left[0, \frac{1}{n+1}\right)$ for disk D_{n+1} such that $r_0 = \left[\frac{1}{n+1}, \frac{1}{n}\right)$ comes topmost. More formally, we map $r_i = \left[\frac{1}{n+1}, \frac{1}{n}\right)$ to $r_{n+1} = \left[\frac{n-i}{n\cdot(n+1)}, \frac{n-i+1}{n\cdot(n+1)}\right)$. This process is illustrated in Figure 3.3.

The assimilation function defines not only the mapping of ranges to disks but is also the foundation for the access of data blocks. Given a block b, let $h_i(b) \in [0, 1/i)$ be the *height* of that block in the presence of i disks. Initially, $h_1(b)$ equals $h(b)$. Whenever a range containing $h_i(b)$ is moved to a new disk, the height is adjusted according to the new local range[1]. In other words, moving a range r from disk D_i to D_j induces the replacement of all data blocks which had fallen into that range.

To calculate the disk that stores block b, we 'simulate' the replacements b would have performed if the storage network would have been built up from one to the current number of n disks. By simulate we mean the identification of such transitions that coursed the adjustment of the height $h_i(b)$ of block b. Because of the adjustments, the identification of a replacement situation is simple, it happens if the height of blocks b is less than $1/j$ for some $j \leq n$. Therefore, we could derive the next disk D_j by

$$j = \left\lfloor \frac{1}{h_i(b)} \right\rfloor + 1 \ . \tag{3.2}$$

[1]This is done in the same way that the boundaries of cut ranges are adjusted to fit inside the newly fused local range.

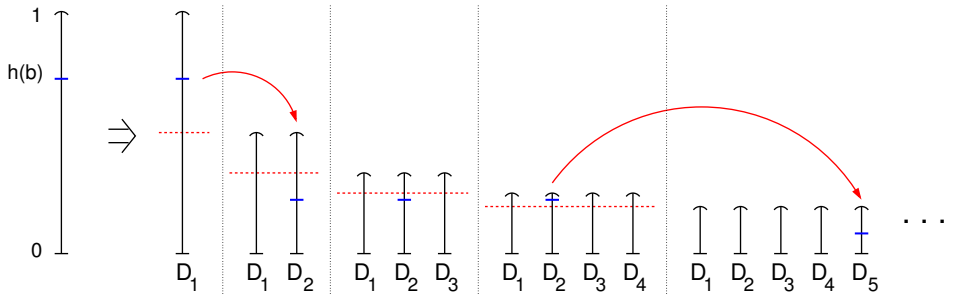

Figure 3.4: The stepwise assimilation in the uniform case. We simulate the growth of the storage network from 1 to n disks and track the height of the block whenever it is replaced.

We could stop the calculation as soon as $j \geq n$. The access of an arbitrary block b is depicted in Figure 3.4.

Analysis of the Cut-and-Paste Strategy

In this section, we prove the quality of the Cut-and-Paste strategy. We start by showing the space balance property. Obviously, the assimilation function ensures that each disk in the storage network gets the same share of the $[0,1)$ interval. If we add up the size of the ranges mapped to D_{n+1} when increasing the number of disks by one, we get

$$n \cdot \frac{1}{n(n+1)} = \frac{1}{n+1} \quad .$$

The cutting process ensures that all other disks also have a range of size $\frac{1}{n+1}$. Hence, all disks get the same share of the interval and the space balance depends only on the pseudo-random hash function $h(b)$.

We start the analysis with two auxiliary lemmata deriving the number of blocks that fall into any range in the expected case and with high probability. This enables us to bridge the first phase of the Cut-and-Paste strategy such that we have to concern ourselves only with the size of the ranges.

Lemma 11 *Let p be the number of blocks hashed into a $[0,1)$ interval by a random hash function $h(b)$. Then, any range of size $l \in \mathbb{R}$ of the $[0,1)$ interval is expected to contain $p \cdot l$ blocks.*

Proof. Consider any of the p blocks. Any value in the $[0,1)$ interval is equally likely to be the hash value $h(b)$ because of the randomness of h. Note, that the

size of the whole interval is 1. Therefore, the probability of b falling into a range $r_i = [lb, ub)$ of size $l = ub - lb$ is equal to l,

$$Pr[h(b) \text{ falls into range of size } l] = l \ .$$

Furthermore, the hash function h ensures that each of the p blocks is independent of each other. Hence, the expected number of blocks falling into a range of size l is equal to $p \cdot l$. ∎

Lemma 12 *Let p be the number of blocks hashed into a $[0,1)$ interval by a random hash function $h(b)$. The number of blocks in any range $r_i = [lb, ub)$ of size $l = ub - lb$ is at most*

$$\left(1 + \mathcal{O}\left(\sqrt{\frac{\log n}{l \cdot p}}\right)\right) \cdot p \cdot l \quad w.h.p.$$

if $p \cdot l > \log n$.

Proof. From Lemma 11 we already know that the expected number of blocks in a range of size l is $p \cdot l$. Because the random hash function ensures the independence of the hash values of each of the p block, we can apply the Chernoff bounds (Lemma 6),

$$Pr[X \geq (1+\delta) \cdot \mu] \leq e^{-\min\{\delta^2, \delta\} \cdot \mu} \ .$$

To prove the lemma, we need to derive the right δ and show the high probability property. If we set δ to

$$\delta = \sqrt{\frac{3 \cdot \log n}{p \cdot l}} \ , \tag{3.3}$$

the Chernoff bounds give us

$$\begin{aligned} Pr\left[X \geq \left(1 + \sqrt{\frac{\log n}{p \cdot l}}\right) p \cdot l\right] &\leq e^{-\left(\sqrt{\frac{\log n}{p \cdot l}}\right)^2 \cdot p \cdot l} \\ &= e^{-\log n} \\ &= \frac{1}{n} \ . \end{aligned} \tag{3.4}$$

Note that we use the assumption of $p \cdot l < \log n$ in order to bound δ such that it is always below 1. Hence, we can use the first case in the minimisation in the exponent of the right hand side of the Chernoff bounds.

If we now increase δ by a constant, the probability decreases exponentially in n. This concludes the proof. ∎

Now, we use the lemma above to establish the space balance property for the Cut-and-Paste strategy.

Theorem 13 *Given a uniform storage network $S(n, C[n])$, p arbitrary blocks drawn from the universe U, and a random hash function $h(b)$ the Cut-and-Paste strategy applied to $S(n, C[n])$ guarantees that every disk D_i gets at most*

$$\left(1 + \mathcal{O}\left(\sqrt{\frac{n \log n}{p}}\right)\right) \cdot \frac{p}{n}$$

blocks with high probability if $n \log n > p$.

Proof. The assimilation function guarantees that the local range on disk D_i has size $\frac{1}{n}$. Note that the assumption of $n \log n > p$ is needed to bound δ in the Chernoff bounds. Applying Lemma 12 proves the theorem. ∎

Corollary 14 *Let p blocks be drawn from the universe U. The Cut-and-Paste strategy is space balanced and, w.h.p., no disk gets more blocks than*

$$\left(1 + \mathcal{O}\left(\sqrt{\frac{n \log n}{p}}\right)\right) \cdot \frac{p}{n} \ .$$

Let us now consider the access to an arbitrary block b. As noted before, we only need to take the 'simulated' replacements of block b into account. Clearly, the access efficiency depends on that number.

Theorem 15 *The maximum number of times an arbitrary block b is replaced when simulating the assimilation function from 1 to n disks is $\lfloor \log n \rfloor + 2$.*

Proof. Consider any block b on disk D_{n_0}. This block needs to be moved to disk D_{n_1} when $\frac{1}{n_1} \leq h_{n_0}(b)$ holds for the first time. Hence, $n_1 = \lceil \frac{1}{h_{n_0}(b)} \rceil$. Furthermore, we know that the height of block b on disk D_{n_1} equals $h_{n_1} \leq \frac{(n_1 - n_0)}{n_1 \cdot (n_1 - 1)}$ because all cut ranges of size $\frac{1}{n_1 \cdot (n_1 - 1)}$ from all disks between n_0 and n_1 are below the cut range from disk D_{n_0}. Now we can derive the height of block b when it will be moved the next time. It holds that

$$n_2 \geq \left\lceil \frac{n_1 \cdot (n_1 - 1)}{n_1 - n_0} \right\rceil \tag{3.5}$$

when b will be put on disk D_{n_2}. Note that $n_1 > n_0$. It is not difficult to show that $\frac{n_1 \cdot (n_1 - 1)}{n_0 \cdot (n_1 - n_0)}$ is minimised if $n_1 = 2n_0 - 1$ or $n_1 = 2n_0$ for that restriction. To see this, consider the following function

$$f(n) = \frac{n \cdot (n - 1)}{n - n_0} \ . \tag{3.6}$$

To find the extrema of $f(n)$, we have to look at the first and second order derivatives $f'(n)$ and $f''(n)$, respectively:

$$\begin{aligned} f'(n) &= \frac{2n-1}{n-n_0} - \frac{n^2-n_1}{(n-n_0)^2} \\ &= \frac{n^2-(2n\cdot n_0)+n_0}{(n-n_0)^2} \end{aligned}$$

$$\begin{aligned} f''(n) &= \frac{1}{n-n_0} - \frac{4n-2}{(n-n_0)^2} + \frac{2n(n-1)}{(n-n_0)^3} \\ &= \frac{2n_0(n_0-1)}{(n-n_0)^3} \end{aligned}$$

Setting $f'(n) = 0$ gives us the extrema of $f(n)$, namely n_2 and n_3.

$$\begin{aligned} n_2 &= n_0 + \sqrt{n^2-n_0} \\ n_3 &= n_0 - \sqrt{n^2-n_0} \end{aligned}$$

The restriction of $n_3 > n_0$ cancels out the latter. To distinguish a minimum from a maximum, we have to look at the value of $f''(n)$ for the extremum n_2,

$$\begin{aligned} f''(n_2) &= \frac{2n_0(n_0-1)}{(n_2-n_0)^3} \\ &= \frac{2n_0(n_0-1)}{\left(n_0+\sqrt{n^2-n_0}-n_0\right)^3} \\ &= \frac{2n_0(n_0-1)}{(n^2-n_0)^{\frac{3}{2}}} \\ &> 0 \ . \end{aligned}$$

Hence, it follows that $f(n)$ has a minimum at n_2. Furthermore, we know that n_1 has to be a positive integer. That implies that $f(n)$ is minimal for

$$n_1 = 2n_0 - 1 \qquad \text{and} \qquad n_1 = 2n_0 \ .$$

Substituting these values into Inequality (3.5) leads to the following facts;

$$\begin{aligned} n_2 &\geq \left\lceil \frac{n_1\cdot(n_1-1)}{n_1-n_0} \right\rceil \\ &\geq \frac{2(2n_0-1)(n_0-1)}{n_0-1} \\ &= 4n_0-2 \end{aligned}$$

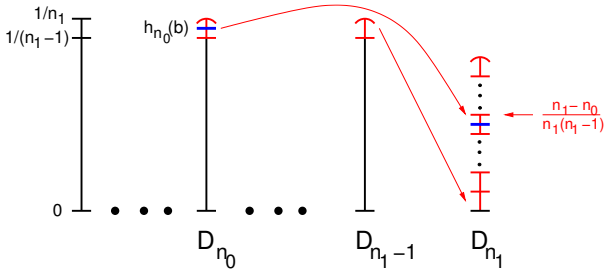

Figure 3.5: Illustration for the proof of Theorem 15.

$$\begin{aligned} n_2 &\geq \left\lceil \frac{n_1 \cdot (n_1 - 1)}{n_1 - n_0} \right\rceil \\ &\geq \frac{2n_0(2n_0 - 1)}{n_0} \\ &= 4n_0 - 2 \end{aligned}$$

for $n_1 = 2n_0 - 1$ and $n_1 = 2n_0$, respectively. Therefore, for any choice of n_0 and $h_{n_0}(b)$ it holds that

$$n_2 > 4 \cdot n_0 - 2 \quad . \tag{3.7}$$

In other words, every $(2i)$th replacement of any block b is at a disk with a label at least 4^{i-1}. This fact proves the theorem. ■

It turns out that our assimilation function is very close to the optimal as the next lemma states.

Lemma 16 *For any uniform, space balanced distribution strategy, the maximum number of replacements a block performs when simulating the situation from* 1 *to* n *disks is at least* $\ln n - 1$.

Proof. Consider p blocks. The space balance property implies that p/i blocks need to be replaced when evolving from $i - 1$ to i disks for $i \geq 2$. Therefore, the total number of replacements is at least

$$\sum_{i=2}^{n} \frac{p}{i} \geq p \cdot (\ln n - 1) \quad .$$

It follows that the average (and therefore also the worst) number of replacements a block performs when evolving from 1 to n disks is at least $\ln n - 1$, proving the lemma. ■

Next we consider the adaptivity of the Cut-and-Paste strategy. Suppose we add a disk to the existing n ones of storage network $S(n, C[n])$. In this case, the assimilation function tell us which cut ranges from the n disks are placed on the new disk D_{n+1}. When we decrease the number of disks, a straightforward application of the assimilation function is not possible. Instead, we roll back to a former configuration, namely the distribution when the network consisted of $n-1$ disks. The following two theorems manifest the adaptivity of the Cut-and-Paste strategy.

Theorem 17 *The Cut-and-Paste strategy is* 1*-competitive in the expected case (or* $(1+\epsilon)$*-competitive w.h.p. with* $\epsilon < 1$ *if* $n \log n < p$*) when adding a new disk to a storage network* $S(n, C[n])$*.*

Proof. Let p blocks be stored on the n disks of the storage network. Consider the situation when changing from n to $n+1$ disks. An optimal offline algorithm for the problem has to move at least $\left\lfloor p \cdot \frac{1}{n+1} \right\rfloor$ blocks due to the space balance requirement. The Cut-and-Paste strategy applies the assimilation function and cuts ranges from all n disks and concatenates them to the new local range of D_{n+1}. As noted before, the size of the cut ranges is given by

$$n \cdot \frac{1}{n(n+1)} = \frac{1}{n+1} \ .$$

The expectation value for the number of redistributed blocks follows from Lemma 11 that relates the number of blocks in any range to its size. The high probability case is derived directly from Lemma 12. Choosing δ appropriately leads to the needed result and proves the theorem. ■

Theorem 18 *The Cut-and-Paste strategy is* 2*-competitive in the expected case (or* $(2+\epsilon)$*-competitive w.h.p. with* $\epsilon < 1$ *if* $n \log n < p$*) when removing a disk from a storage network* $S(n, C[n])$*.*

Proof. Let p blocks be stored on the n disks of the storage network. The results from Lemma 11 and Lemma 12 imply that we only need to derive the size of all replaced ranges to prove the theorem. Assume that we want to remove disk D_j. Then, an optimal offline strategy must replace at least $p \cdot \frac{1}{n}$ blocks to evict all blocks from the disk D_j. Because the assimilation function relies on the order of the participating disks, it cannot be directly applied when a disk is removed. Hence, we do the following trick. We undo the last assimilation step to free the last disk D_n. Now we can move all the blocks from disk D_j to D_n and D_n takes the place of D_j. Both operation clearly move all the blocks associated with a range of size $1/n$. From Lemma 11 and Lemma 12 the theorem follows. ■

Note that the quality of the Cut-and-Paste strategy depends roughly on the ratio between the number of disks and the number of blocks stored on those disks. Furthermore, the quality of the distribution depends only on one random hash function. This fact, together with the equal size of the ranges mapped to disks, ensures a very even distribution of blocks.

Furthermore, we need only the number of disks n and the hash function $h(b)$ to access any given block b. Together with the logarithmic number of computations, this ensures resource efficiency.

3.2 Strategies for Heterogeneous Networks

So far, we have seen only data distribution strategies that are applicable for uniform storage networks. It turns out that the introduction of heterogeneity makes the data distribution problem much more challenging. This is due to the fact that the share of data blocks a disk D_i has to store depends on its capacity and cannot be derived from the number of disks itself. Hence, we need to find an efficient way to represent the capacity distribution. Because the number of data blocks that must be replaced in case of a change depends also on the capacity distribution such a representation is crucial for the adaptivity.

3.2.1 The Level Strategy

The Level strategy is the first nonuniform strategy. Even if its quality is not very good when it comes to adaptivity, it enables some valuable insights into the problems introduced by heterogeneity. First, we introduce some useful definitions and give a description of the strategy before we analyse it.

How can we tackle heterogeneity? We know from the last section that there are algorithms for uniform networks. Therefore, the most natural solution which suggests itself lies in the reduction of heterogeneity to homogeneity.

Call a disk *saturated* if it has no capacity left to store any more blocks. We reduce the heterogeneity problem to a number of uniform ones by introducing *levels* in which only non-saturated disks participate. Whenever we are unable to place a block in level L_j we try it in the next higher level L_{j+1}. Each level L_j is 'separated' form the one below by a random hash function $h_j(b)$ which maps the blocks, reaching that level, anew into a $[0, 1)$ interval. To place the block inside a level L_j, we apply the Cut-and-Paste strategy. This is possible because the levels are not correlated, and the disks inside a level are uniform. Note that the order of disks inside a level may vary between levels. In other words, the separation by the random hash functions makes each level independent of each other. To access block b, we apply h_0 to it and derive the disk storing it in level L_0 via the Cut-and-

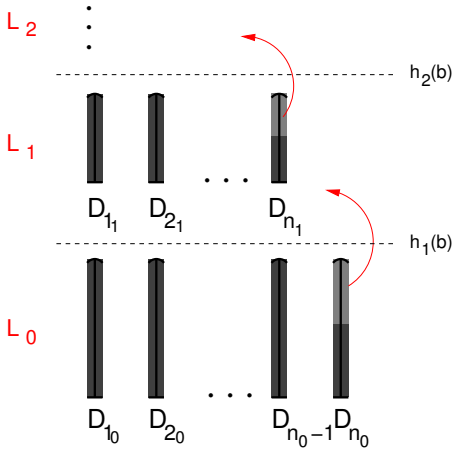

Figure 3.6: The shaded areas in this figure symbolise the number of blocks associated with the ranges. The darker regions are the blocks stored on a disk in that level while the lighter ones correspond to all the blocks that cannot be stored on a disk due to saturation. Whenever a block b cannot be placed in level L_j it is hashed into the $[0, 1)$ interval for the next level L_{j+1} with hash function h_{j+1}. Inside each level we apply the Cut-and-Paste strategy to find the correct disk.

Paste strategy. If this disk is saturated, we apply the next hash function to b and try to find it in the next higher level. The general approach is illustrated in Figure 3.6.

Let the number of disks in each level L_j be denoted by n_j with $n_0 = n$. Furthermore, let B_j be the number of blocks each disk is supposed to store in level j. Then it holds that $B_0 = \frac{1}{n_0} \cdot p$ and $B_j = \frac{n_j - n_{j+1}}{n_{j+1}} \cdot B_{j-1}$. Let ϵ_j be the rate by which the number of disks in level L_j decreases. Using this parameter we get $n_j = \epsilon_j \cdot n_{j-1}$ and $B_j = \left(\frac{1-\epsilon_j}{\epsilon_j}\right) \cdot B_{j-1}$ for the number of disks per level and the number of blocks stored in it, respectively.

So, how do we detect that a block b is stored in level L_j? We do know the number of blocks a disk should store in level L_j, i.e. B_j. Hence, we redefine the local range of disk D_i to

$$r_i = \left[0, \min\left\{\frac{1}{n_j}, \frac{C_i - B_j}{B_j \cdot n_j}\right\}\right) .$$

All blocks falling into r_i are placed on disk D_i and all blocks falling into $\left[\min\left\{\frac{1}{n_j}, \frac{C_i - B_j}{B_j \cdot n_j}\right\}, \frac{1}{n_j}\right)$ are moved to the next level L_{j+1}.

Analysis of the Level Strategy

In this section we analyse the Level strategy. This analysis consists of the verification of space balance properties, examination of access behaviour, and adaptivity. We begin with some consideration concerning space balance.

Heterogeneity opens up different approaches to space balance. Due to the different disk capacities, some disks are unable to store the full amount of blocks and, hence, saturate. The important question is how a distribution scheme copes with that situation. If it is space balanced, the disks should saturate roughly at the same time, i.e. when p is close to C_{total}, the total capacity of the storage network. In contrast, the distribution strategy might try to use the disks as long as possible, enhancing the parallel access to blocks. All disks participate in the answering of block requests until they are saturated. The Level strategy aims to distribute the blocks as evenly over all disks as possible following the latter approach.

Theorem 19 *Given a storage network $S(n, C[n])$ and p blocks, the Level strategy is not space balanced until p is close to C_{total}.*

Proof. The space balance property requires that for any p, each disk D_i gets $\frac{C_i}{C_{total}} \cdot p$ blocks on expectation. To prove the first part, consider any $p \leq C_j \cdot n$ where C_j is the disk with the smallest capacity. Hence, we have only one level and, therefore, each disk gets the same amount of blocks, depending on the quality of the uniform strategy used, violating the space balance property.

For the second part of the theorem, consider p close to C_{total}. Because only non-saturated disks participate in a new level, almost all of the n disks are saturated. It follows that each disk D_i gets an expected number of $\frac{C_i}{C_{total}} \cdot p$ blocks, proving the theorem. ■

Even if the Level strategy is not space balanced, the blocks are evenly distributed over the n disks. It turns out that space balance and maximal parallel access cannot be achieved simultaneously in the heterogeneity setting, stating the first important lesson learnt from this strategy.

To access any block b, we need to find it in each level L_j. Hence, the number of operations is bounded by $L_{max} \cdot UNIFORM(n_j)$, where $UNIFORM(n_j)$ denotes the complexity to access a block in a uniform setting consisting of n_j disks. Obviously, a trivial upper bound on the number of levels is given by n because each disk can only introduce one new level. We begin the analysis by showing a lower and an upper bound for the number of levels, starting with the lower bound.

Theorem 20 *For any n and $p = n^q$ with $q > 2$ there is a collection of disks for which our approach requires at least $\frac{\log p}{2 \cdot q}$ levels for placing p data blocks.*

Proof. We prove the theorem by constructing a counterexample. Let's assume that ϵ_j is equal to some fixed ϵ for all levels L_j. Using the above notation, it follows for level L_j that $n_j = \epsilon^j \cdot n_0$ and $B_j = \left(\frac{1-\epsilon}{\epsilon}\right)^j \cdot B_0$. Surely, it has to hold that for all levels L_j, we have $n_j \geq 1$ and $B_j \geq 1$. To prove the theorem, we show that for $\epsilon = 2^{-\frac{1}{k}}$ with $k = \frac{q}{2 \log q}$ these inequalities hold for at least $k \log n$ levels.

The fixing of ϵ leads to $n_j = 2^{-\frac{j}{k}} \cdot n_0$. Since $n_0 = n$, it follows that n_j is at least 1 for $j \leq k \cdot \log n$,

$$\begin{aligned} n_j &= \frac{n_0}{2^{\frac{j}{k}}} \\ &\geq \frac{n_0}{2^{\frac{k \log n}{k}}} = \frac{n_0}{n} = 1 \ . \end{aligned}$$

It remains to show that $B_j \geq 1$ for $j \leq k \cdot \log n$. We know that

$$\frac{1 - 2^{-\frac{1}{k}}}{2^{-\frac{1}{k}}} = 2^{\frac{1}{k}} \left(1 - 2^{-\frac{1}{k}}\right) = 2^{\frac{1}{k}} - 1$$

and

$$2^{\frac{1}{k}} - 1 \geq \frac{1}{2k} \quad \Leftrightarrow \quad 4 \geq \left(1 + \frac{1}{2k}\right)^{2k} \ .$$

This is true for all $k \geq 1$. Because we require $p = n^q$, it holds that

$$\begin{aligned} \left(\frac{1}{2k}\right)^{k \log n} &= \frac{1}{2^{k \log(2k) \log n}} \\ &\geq \frac{1}{2^{\frac{\log p}{2}}} \\ &= \frac{1}{\sqrt{p}} \end{aligned}$$

It follows that

$$B_{k \cdot \log n} = \left(\frac{1-\epsilon}{\epsilon}\right)^{k \cdot \log n} \cdot \frac{p}{n} \geq \frac{\sqrt{p}}{n} \geq 1$$

for all $q \geq 2$, which completes the proof. ∎

Using a very similar argument, we are able to prove a worst case bound on the number of levels (L_{max}) induced by our strategy.

Theorem 21 *For any n and $p = n^q$ with $q \geq 1$, the maximum number of levels L_{max} induced by the Level strategy applied to a storage network $S(n, C[n])$ is*

$$\frac{3 \cdot (q+1)}{\log(q+1)} \cdot \log n \ .$$

Proof. We construct the worst case by deriving two inequalities which must be fulfilled for as long as possible. As before, the first restricts the number of disks and the second the number of blocks in level L_j. Using our notation we get

$$n_j = \left(\prod_{i=0}^{j-1} \epsilon_i\right) \cdot n$$

and

$$B_j \cdot n_j \leq \left(\prod_{i=0}^{j-1} (1-\epsilon_i)\right) \cdot p \ .$$

These two inequalities imply the following two conditions which need to be met for any level L_j,

$$\prod_{i=0}^{j-1} \epsilon_i \geq \frac{1}{n} \quad \text{and} \quad \prod_{i=0}^{j-1} (1-\epsilon_i) \geq \frac{1}{p} \ . \tag{3.8}$$

Now, we show that Inequality (3.8) can be fulfilled for at most $\frac{3(q+1)}{\log(q+1)}$ levels.

Suppose that the conditions can be satisfied for $l \geq 3k \log n$ levels, where $k = \frac{q+1}{\log(q+1)}$. Then, there must be at least $k \log n$ levels L_j with $\epsilon_j < 2^{-\frac{1}{k}}$, or at least $2k \log n$ levels L_j with $\epsilon_j \geq 2^{-\frac{1}{k}}$. The first situation implies that after $k \log n$ levels,

$$\prod_{i=0}^{k \log n} \epsilon_i < 2^{-\log n} \leq \frac{1}{n}$$

which violates the first condition. In the other case, we have

$$\prod_{i=0}^{2k \log n} (1-\epsilon_i) \leq \left(1 - 2^{-\frac{1}{k}}\right)^{2k \log n} \ .$$

Because it holds for all $k \geq 1$ that $\left(1 - \frac{1}{k}\right)^k < \frac{1}{2}$, the above inequality becomes

$$\left(1 - 2^{-\frac{1}{k}}\right)^{2k \log n} < \left(\frac{1}{k}\right)^{2k \log n} \leq 2^{-q \log n} = \frac{1}{p} \ ,$$

violating the second condition of Inequality (3.8). This completes the proof. ∎

If we use the Cut-and-Paste strategy for the uniform placement inside each level, the above theorem and Theorem 15 lead to the following corollary.

Corollary 22 *Given a storage network $S(n, C[n])$ storing p blocks and using the Level strategy as the distribution scheme, the number of computations needed to locate any block b is bounded by $\mathcal{O}\left(\log n \cdot \frac{\log p}{\log q}\right)$ with $p = n^q$.*

The above theorem shows that our approach is very close to the optimal, but the number of levels and, hence, the efficiency to access a block, depends on the number of blocks in the system. This is a big disadvantage if the storage network is large and has to store a vast number of blocks.

Before we are able to analyse the adaptivity, we need to explain what we do when the number of disk drives changes. The insertion of new disks is straightforward, but the removal needs some further consideration. Our simple strategy that we applied in the uniform case, free the last disk and exchange it with the failed one, is of no use since the disks in each level are uniform, but the disks themselves may store a very different amount of blocks. Nevertheless, we may exploit the levels themselves. Whenever a disk D_f fails we set its size to zero and let the next level handle its blocks. This has the advantage that the contents of D_f is evenly spread over all remaining disks. Unfortunately, the number of disks in each level increases by one and, hence, the number of computations for the uniform placement is greater. However, the dependence is logarithmic, resulting in one more step whenever the number of disks is doubled. This seems acceptable since such faults do not occur very often. Furthermore, when adding new disks these holes are filled.

If we use the Level strategy with the simple assimilation function within each level, we are not able to bound the competitive ratio tightly. This is due to the fact that we cannot guarantee that blocks move only from 'old' to 'new' disks. Consider the case when a new disk D_{n_j+1} enters level L_j but it has only one free block left. We need to apply the assimilation function to balance the load evenly among the disks in that level. Assume that t blocks need to be put on disk D_{n_j+1} but it is only capable of storing just one more. Hence, the remaining $t-1$ blocks are moved to the next level, returning to the same group of disks they are coming from, as depict in Figure 3.7. This process can continue in higher levels since the one block moved to D_{n_j+1} may come from a saturated disk which now has to participate in the next level. The number of blocks t can be arbitrarily large which results in a unbounded competitive ratio.

To overcome this behaviour, we refine the Level strategy. Instead of fusing all cut ranges together, we use a new hash function that provides new heights in $\left[0, \frac{1}{n+1}\right)$ for the blocks in the corresponding ranges. This function ensures that for any local range r_{i+1} of disk D_{i+1}, every disk D_j with $j \leq i$ has the same number of blocks with new height at most the upper bound of r_{i+1}. Thus, we replace only those blocks that are below the new upper bound of r_{i+1} and leave all other blocks unchanged. However, we treat the untouched blocks for the next levels as if they

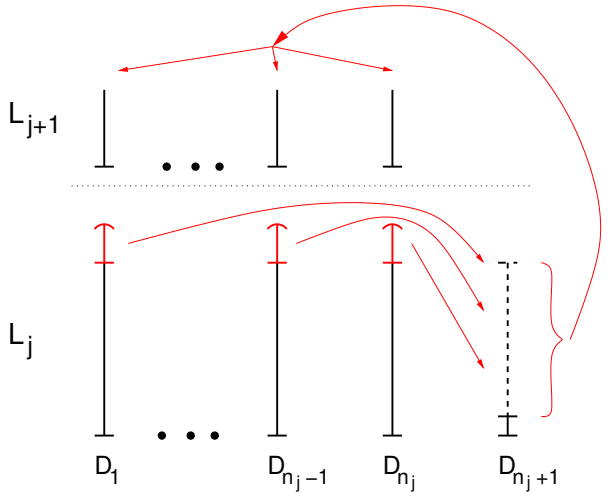

Figure 3.7: When a disk D_{n_j+1} with only one free block enters a level L_j, the uniform assimilation function cuts the appropriate ranges from all other disks in this level and puts all blocks inside those ranges back (via the next level) to a subgroup of those disks.

would participate there (as they would in the simple strategy). This leads to a distribution which is very close to optimal, and it can be shown that every block is replaced at most $\mathcal{O}(\log n)$ times in each level, w.h.p. This ensures that all bounds shown for the simple Level strategy still hold.

The main drawback of the refined assimilation function lies in the introduced ambiguity of the location of a block. At some level it might have been kept back because some disk was not able to store it, but it could also have been moved one level up due to the insertion of new blocks. However, there is only one possible position in each level for storing a block, namely the last disk that was able to hold it. This implies that we need at most l requests to get any block where l is the number of levels induced by the new strategy. Only one of these requests involves a disk access because there is only one location for any block and, therefore, data is only once be transported over communication links.

In the uniform case, the addition and removal of blocks had no effect on the adaptivity of the distribution strategy since the disks could always store the blocks unless the whole system was saturated. In a heterogeneous setting, this is different. We have to preserve the internal structure of the Level strategy, and this introduces an unavoidable overhead.

Theorem 23 *Given a storage network $S(n, C[n])$ and p blocks, the refined Level strategy applied to $S(n, C[n])$ is l-competitive for the removal and $(l+1)$-competitive for the addition*

of a block $b \in U$ where l is the number of levels introduced by the refined Level strategy for the placement of p blocks.

Proof. As noted above, the internal structure of the Level strategy has to be preserved. When a new block b is accessed any optimal strategy has to place at least this block. The worst case for the Level strategy occurs when b must be placed on a saturated disk D_s and its height is less than the height of D_s. Because we have to keep the range on D_s compact, the block with the maximal height is evicted and placed in the next level. In the last level it might happen that the new block introduces a new level. This cannot occur when a block is removed, proving the theorem. ■

The addition of new disks is straightforward. We simply apply the uniform scheme in every level in which the new disk participates. This leads to the following theorem. Note that there are two cases to be considered. In the first case, the newly added disk can store the complete $1/(n+1)$ fraction of the p blocks. In the second case, the capacity of disk D_{n+1} is less than the $1/(n+1)$ share of the blocks and only C_{n+1} blocks are put on that disk.

Theorem 24 *Given a storage network $S(n, C[n])$ and p blocks, the refined Level strategy is l-competitive for the addition of a new disk, where l is the number of levels induced by the refined Level strategy.*

Proof. The above reasoning implies that an optimal offline algorithm has to replace at least $t = \min\left\{C_{n+1}, \frac{1}{n+1} \cdot p\right\}$ blocks. Because the new disk participates in as many level as it has capacity left in, it follows that the refined Level strategy puts t blocks on D_{n+1}. Hence, it evicts at most t blocks from the other disks. This situation is the same as removing t blocks from the system. Applying Theorem 23 leads to the theorem. ■

For the removal of disks, we apply the zero-height strategy, i.e. failed disks leave the storage network by setting their capacity to zero. As before, an optimal offline strategy has to move at least C_f blocks from a failed disk D_f. However, the Level strategy has also to preserve its internal structure and, therefore, replaces more blocks.

Theorem 25 *Given a storage network $S(n, C[n])$ and p blocks, the zero-height strategy is $(l+1)$-competitive, where l is the number of levels induced by the refined Level strategy, when a disk is removed.*

Proof. An optimal offline strategy has to move at least C_f blocks when disk D_f is removed. The zero-height strategy introduces the C_f blocks to the next level. Applying Theorem 23 concludes the proof. ■

The refined Level strategy is a first step towards a heterogeneous, adaptive distribution strategy. It is capable of handling different sized disks in a storage network. Nevertheless, it has disadvantages that make a practical application doubtful. First, the number of levels and, hence, the access to any block depends on the number of blocks currently in the system. Second, the possible oscillation of disks between levels induces a more complicated assimilation function resulting in a rather complicated distribution scheme. Most of the problems can be traced back to the heterogeneity of the disks. So it seems natural to investigate the impact of a 'smoother' heterogeneity. What do we mean by 'smooth'? Obviously, there is a dependence between the number of levels and the number of different sized disks. So we get some flexibility if we allow the disks to waste storage space because, then, the disks do not need to participate in the next level even if they have some space left.

Exploiting Slackness

By slackness we mean the freedom of disks not to enter new levels even if they have some capacity left. We achieve this flexibility by allowing some disk capacity to be wasted. Clearly, we cannot guarantee a completely even distribution anymore. Instead, we ensure that the blocks are evenly distributed over all non-wasted parts of the disks.

Under that assumption, we can pursue two different approaches:

1. **Local Slackness Strategy**
   We allow that up to a fixed ratio $w \in (0,1)$ of the capacity of *each* disk may be left free. This implies that we waste at most w of the total capacity of the storage network.

2. **Global Slackness Strategy**
   Up to a fixed ratio $w \in (0,1)$ of the *total* capacity may be wasted.

Clearly, the global slackness strategy bears more potential because the capacity constraints can be handled with more flexibility.

The Local Slackness Strategy The local slackness strategy has the capability to mute the oscillation of disks between two consecutive levels L_j and L_{j+1}. This ability arises from the property that not all of the capacity of a disk has to be used, i.e. by wasting up to a ratio $w \in (0,1)$ of its capacity. We get the freedom to define a threshold below which a disk may stay in a certain level L_j even if it still has free capacity. Note that this is very similar to the trick applied in the refined Level strategy. Instead of keeping blocks back, this situation is completely avoided.

The local slackness strategy works as follows. Let the *threshold* x be defined by $x = w/(1-w)$. Clearly, all disks participate in level L_0. However, for all $j > 0$,

we allow a disk only to participate in level L_j if its remaining capacity is at least x times its capacity consumed in all previous levels. This constraint ensures that at most a fraction w of its capacity is dissipated. Evidently, level L_j is the last level if

$$B_j \leq x \cdot \sum_{i=0}^{j-1} B_i \ . \tag{3.9}$$

For all disks participating in a level, the refined level strategy is used.

The main consequence of the above rule is that as soon as the free capacity of a disk drops below the participation threshold, it does not participate in higher levels. We show, that this significantly reduces the number of levels and the amount of blocks that have to be moved due to oscillation.

Clearly, as long as the number of blocks $p \leq (1-w) \cdot C_{total}$, our placement strategy is always able to map all p blocks onto the n disks. Furthermore, the maximum relative deviation from an even distribution can be at most w.

In the following, we analyse the adaptivity of the local slackness strategy. More specifically, we give an upper and a lower bound for the number of levels induced by that strategy, proving that it is close to optimal.

Theorem 26 *For any w and n, the maximum number of levels caused by the local slackness strategy is at most $\log_{1+x} n + 1$, where $x = w/(1-w)$.*

Proof. Let $\delta_j = 1/\epsilon_j$ be defined for every level L_j. The key to the proof lies in bounding the number of blocks stored in level L_{j+1}. A disk in level L_j has to store the maximal amount of blocks if all disks that are saturated in level L_j just store the minimal number of blocks that enables them to participate in level L_j. Hence, we are able to bound B_{j+1} as follows,

$$\begin{aligned} B_{j+1} &\leq \frac{n_j - n_{j+1}}{n_{j+1}} \cdot \left(B_j - x \cdot \sum_{i=0}^{j-1} B_i \right) \\ &= \left(\frac{1}{\epsilon_j} - 1 \right) \cdot \left(B_j - x \cdot \sum_{i=0}^{j-1} B_i \right) \\ &= (\delta_j - 1) \cdot \left(B_j - x \cdot \sum_{i=0}^{j-1} B_i \right) \ . \end{aligned}$$

Now, we can estimate δ_j by using Inequality (3.9). Assuming that L_{j+1} is not the

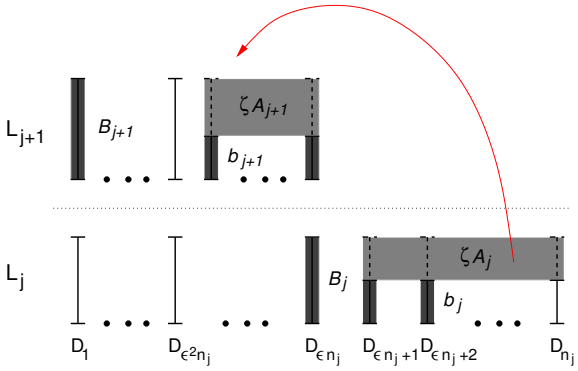

Figure 3.8: The figure illustrates the definitions of Theorem 27.The shaded areas represent the number of blocks associated with ranges and not the ranges themselves. The fact that we are allowed to waste some storage space on each disk results in a minimal size of all b_j.

last level, we get the following inequality,

$$\begin{aligned}(\delta_j - 1) \cdot \left(B_j - x \cdot \sum_{i=0}^{j-1} B_i \right) &\geq x \cdot \sum_{i=0}^{j} B_i \\ (\delta_j - 1) \cdot B_j &\geq x \cdot B_j \\ \delta_j &\geq 1 + x \ . \end{aligned} \tag{3.10}$$

Due to the correlation of δ_j and ϵ_j, the last inequality bounds the rate by which the number of disks in the levels decrease. This concludes the proof. ■

Note that the number of levels is now independent of the number of accessed blocks p. This is a great achievement because it improves all theorems for the refined level strategy significantly.

The question that remains to be answered concerns the quality of that achievement. How tight is this bound? The following lower bound on the maximum number of levels shows that it is very close to optimal.

Theorem 27 *For any w and n, the maximum possible number of levels imposed by the local slackness strategy is as most* $\log_{4(1+x)^2} n + 1$.

Proof. We prove the theorem by constructing a counterexample. Let's assume that ϵ_j is fixed to some ϵ for all levels L_j. Furthermore, we assume that a fixed ratio ξ of all the blocks stored in level L_j is transferred to level L_{j+1}. Now we know

for level L_j that $n_j = \epsilon^j \cdot n_0$ and $B_j = \left(\frac{\xi}{\epsilon}\right)^j \cdot B_0$. Let's define by b_j the least number of blocks any disk in level L_j can store. This is reasonable because we have defined a participation threshold x. Let A_j denote the number of blocks in level L_j. Hence, we have $A_j = B_j \cdot n_j$. Figure 3.8 illustrates the definitions.

From the above assumptions we derive the following inequality for b_j using an estimate on the number of blocks that must be moved to the next level,

$$\begin{aligned} \xi \cdot A_j &= (B_j - b_j) \cdot (1-\epsilon) \cdot n_j \\ \xi \cdot n_j \cdot B_j &= (B_j - b_j) \cdot (1-\epsilon) \cdot n_j \\ b_j &= \frac{1-\epsilon-\xi}{1-\epsilon} \cdot B_j \ . \end{aligned}$$

Introducing $k = \frac{1-\epsilon-\xi}{1-\epsilon}$, we get

$$b_j = k \cdot B_j \ .$$

For all levels L_j it must hold that $n_j \geq 1$ and $b_j \geq x \cdot \sum_{i=0}^{j-1} B_i$. To complete the proof, we derive the condition for which the second inequality holds,

$$\begin{aligned} b_j &\geq x \cdot \sum_{i=0}^{j-1} B_i \\ k \cdot \left(\frac{\xi}{\epsilon}\right)^j \cdot B_0 &\geq x \cdot \sum_{i=0}^{j-1} \left(\frac{\xi}{\epsilon}\right)^i \cdot B_0 \\ k \cdot \left(\frac{\xi}{\epsilon}\right)^j &\geq x \cdot \frac{\left(\frac{\xi}{\epsilon}\right)^j - 1}{\frac{\xi}{\epsilon} - 1} \\ k \cdot \left(\frac{\xi}{\epsilon}\right)^{j+1} - (k+x) \cdot \left(\frac{\xi}{\epsilon}\right)^j + x &\geq 0 \\ k \cdot \left(\frac{\xi}{\epsilon}\right)^{j+1} - (k+x) \cdot \left(\frac{\xi}{\epsilon}\right)^j &\geq 0 \\ k \cdot \frac{\xi}{\epsilon} - (k+x) &\geq 0 \ . \end{aligned} \tag{3.11}$$

Now, we set $\epsilon = \frac{1}{4(1+x)^2}$ and $\xi = \frac{1}{2(1+x)}$ such that the Inequality (3.11) holds for all $x \geq 0$ and $w \geq 0$. The theorem follows. ∎

The Global Slackness Strategy The global slackness strategy has more freedom to reduce the number of levels because it is allowed to arrange the disks globally. More specifically, we try to find g groups of equal size such that at most a fraction w of the total capacity is wasted. Obviously, if we have g such groups the

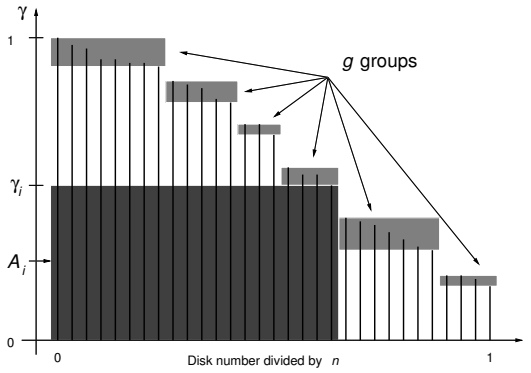

Figure 3.9: The figure illustrates the definitions of Theorem 28. Note that the disks are ordered by their capacity C_i.

number of levels would be g. Note that all disks inside a group do not need to have exactly the same capacity because we are allowed to waste some storage space.

The following notations enable us to prove an upper and a lower bound on the number of levels induced by the global slackness strategy. We assume, w.l.o.g., that the disks are ordered according to their capacities C_i, i.e. $C_i \geq C_{i+1}$ for all $i \geq 1$. The *relative height* of disks D_i is defined by $\gamma_i = C_i/C_1$. Furthermore, let $\Gamma = \frac{1}{n}\sum_{i=1}^{n} \gamma_i$ be the *area* covered by the disks. Clearly, it holds that $\frac{1}{n} \leq \Gamma \leq 1$.

We first show a lower bound on the number of groups and, hence, on the number of levels. Then we give an upper bound. Both bounds relate the 'heterogeneousness', measured by Γ, to the number of levels g. We conjecture that a storage network with almost equal sized disks behaves much better than a network where the capacity of the disks are far apart.

Theorem 28 *For any Γ and w, there exists a collection of disks such that $g \geq (1-w)\ln\left(\frac{1}{\Gamma}\right)$.*

Proof. Suppose it is possible to organise the disks into g groups of equal capacity such that at most $w \cdot \Gamma$ of the total capacity C_{total} is wasted. Then, there must exist g integers $1 \leq i_1 < i_2 \ldots < i_g \leq n$ with the following property (additionally, there are $i_0 = 0$ and i_{g+1} with the same property): If for all $k \in \{1, \ldots, g+1\}$, every disk D_l with $l \in \{i_{k-1}+1, \ldots, i_k\}$ is given a height of γ_{i_k} and a capacity of $\gamma_{i_k} \cdot C_1$, then the total capacity of the g groups is at least $(1-w) \cdot C_{total}$. Let the number of disks in group k be denoted by π_k. The situation is depict in Figure 3.9.

Now, we show up to which g this property holds, which results in a lower bound on the number of groups. As a first step, we estimate the area covered by

all groups:

$$\begin{aligned}
\sum_{k=1}^{g} \pi_k \cdot C_{i_k} &\geq (1-w) \cdot C_{total} \\
\sum_{k=1}^{g} i_k \cdot C_{i_k} &\geq (1-w) \cdot C_{total} \\
\sum_{k=1}^{g} \frac{1}{n} \cdot i_k \cdot \frac{C_{i_k}}{C_1} &\geq (1-w) \cdot \sum_{i=1}^{n} \frac{1}{n} \cdot \frac{C_i}{C_1} \\
\sum_{k=1}^{g} \frac{1}{n} \cdot i_k \cdot \gamma_{i_k} &\geq (1-w) \cdot \Gamma \ . \qquad (3.12)
\end{aligned}$$

Now, we introduce for every $i \in \{1, \ldots, n\}$ an area equivalent to $A_i = \frac{i}{n} \cdot \gamma_i$. Using Inequality (3.12), we get the following inequality

$$\sum_{k=1}^{g} A_{i_k} \geq (1-w) \cdot \Gamma \ . \qquad (3.13)$$

For Inequality (3.13) to be fulfilled there must be at least one k with $A_{i_k} > (1-w) \cdot \frac{\gamma}{g}$. We use this inequality to find the largest g for which it is not satisfied, gaining a lower bound on the number of groups. More specifically, we must find the largest g for which it is possible to set $A_k \leq (1-w) \cdot \frac{\Gamma}{g}$ for all $k \in \{1, \ldots, n\}$. From the definition of A_k we know that $\gamma_k \leq n \cdot (1-w) \cdot \frac{\Gamma}{i \cdot g}$. Furthermore, it holds that $\gamma_k \leq 1$ for all $k \in \{1, \ldots, n\}$. Using these constraints we get the following:

$$\begin{aligned}
& \frac{1}{n} \sum_{k=1}^{n} \min\left\{1, \frac{n \cdot (1-w) \cdot \Gamma}{i \cdot g}\right\} \\
\geq \ & \frac{1}{n}\left(\frac{n \cdot (1-w) \cdot \Gamma}{g} + \sum_{k=n(1-w)\Gamma/g+1}^{n} \frac{n \cdot (1-w) \cdot \Gamma}{i \cdot g}\right) \\
\geq \ & \frac{1}{n}\left(\frac{n \cdot (1-w) \cdot \Gamma}{g} + \left(\ln n - \ln\left(n \cdot (1-w) \cdot \frac{\Gamma}{g}\right)\right) \cdot \frac{n \cdot (1-w) \cdot \Gamma}{g}\right) \\
\geq \ & \frac{(1-w) \cdot \Gamma}{g} + \ln\left(\frac{g}{(1-w) \cdot \Gamma}\right) \cdot \frac{(1-w) \cdot \Gamma}{g} \ .
\end{aligned}$$

This is at least Γ under the constraint that for $\epsilon = \frac{1-w}{g}$ it holds that

$$\epsilon\left(1 + \ln\left(\frac{1}{\epsilon \cdot \Gamma}\right)\right) \geq 1 \ ,$$

which is equivalent to

$$\Gamma \leq \frac{1}{\epsilon} \cdot e^{1-\frac{1}{\epsilon}} \ .$$

This is true if $\epsilon \geq \frac{1}{\ln(\frac{1}{\Gamma})}$, i.e. $g \leq (1-w) \cdot \ln\left(\frac{1}{\Gamma}\right)$. This proves the theorem. ■

The theorem shows that the number of groups still depends on Γ and, hence, on the ratio of the disk capacities. Furthermore, the freedom of dissipating larger portions of some disks does not decrease the number of levels significantly. Like the local slackness strategy, the global variant gains more by using less heterogeneity, i.e. using disks whose capacity does not vary greatly, than by giving up some storage space.

We finish the analysis of the global slackness strategy by giving an upper bound on the number of capacity groups g.

Theorem 29 *For any Γ and w, the number of capacity groups and, hence, the number of levels, when using the global slackness strategy is bounded by $g \leq \frac{2\log^2(\frac{1}{\Gamma})}{w}$.*

Proof. We construct a worst case, consisting of g groups, by using an induction argument. To simplify the description, let the pair (x_i, y_i) characterise the ith iteration with x_i denoting the disk number divided by n and y_i the relative height. We start the induction with $(x_0, y_0) = (0, 1)$. Assume that for some $i \geq 0$ we are currently at (x_i, y_i) with $x_i \geq \sum_{j=1}^{i} 2^j \cdot \Gamma$ (which implies that $y_i \leq \frac{1}{2^i}$). For the induction to hold, we show that it takes at most $\frac{2\log(\frac{1}{\Gamma})}{w}$ groups to get to (x_{i+1}, y_{i+1}) with $x_{i+1} \geq \sum_{j=1}^{i+1} 2^j \cdot \Gamma$. If this is true, then we require at most $\frac{2\log^2(\frac{1}{\Gamma})}{w}$ groups to reach (x, y) with $x \geq 1$. This would end the induction because the last disk would have been reached.

For $i = 0$, the assumption for the induction step clearly holds. So assume in the following that it holds also for some $i \geq 0$. Suppose, as a worst case, that we start at $(x_i, y_i) = \left(\sum_{j=1}^{i} 2^j \cdot \Gamma, \frac{1}{2^i}\right)$. To simplify the description, we define the x-coordinate to be relative to x_i, i.e. we use $x' - x_i$ instead of x' itself. Now consider some fixed positive number A which bounds the waste inside a group to at most A. We aim to bound the number of groups, with waste at most A, it takes to reach (x'_{i+1}, y'_{i+1}) by $x'_{i+1} \geq 2^{i+1} \cdot \Gamma$. Clearly, such a pair is reached if we end up at (x, y) with either $x \geq 2^{i+1} \cdot \Gamma$ or $y \leq 0$. Assume that we need l groups numbered from 1 to l with loss A each for this. Suppose furthermore, that a group s represents a rectangle of height γ_s and width b_s. Then it follows that $\gamma_s \cdot b_s \geq A$. For the worst case we take $\gamma_s \cdot b_s = A$. Furthermore, we require $\sum_{s=1}^{l} \gamma_s \leq 2^{-i}$ and $\sum_{s=1}^{l} b_s \leq 2^{i+1}\Gamma$. In order to find a maximum l for which these restrictions can be fulfilled, we

assume that $\gamma_s = \epsilon \cdot A$ for all s with $\epsilon \in [0,1]$. This leads to two constraints on l,

$$\begin{aligned} l \cdot \epsilon \cdot A &\leq 2^{-i} \\ l &\leq \frac{1}{2^i \cdot \epsilon \cdot A} \end{aligned}$$

and

$$\begin{aligned} l \cdot \frac{1}{\epsilon} &\leq 2^{i+1} \cdot \Gamma \\ l &\leq 2^{i+1} \cdot \epsilon \cdot \Gamma \ . \end{aligned}$$

To maximise l, we choose an ϵ such that

$$\frac{1}{2^i \cdot \epsilon A} = 2^{i+1} \cdot \epsilon \cdot \Gamma \ ,$$

namely $\epsilon = \frac{1}{2^i \cdot \sqrt{2 \cdot A\Gamma}}$. Hence, we get $l = \sqrt{2 \cdot \frac{\Gamma}{A}}$. The lost area, W, is therefore bounded by $l \cdot A = \sqrt{2 \cdot \Gamma \cdot A}$. We need to require $W \leq \frac{w \cdot \Gamma}{\log(1/\Gamma)}$ to end up with a lost area of at least $w \cdot \Gamma$. In this case,

$$\sqrt{2 \cdot \Gamma A} \leq \frac{w \cdot \Gamma}{\ln(1\Gamma)} \qquad \Rightarrow \qquad \sqrt{A} \leq \frac{w\sqrt{\Gamma}}{\sqrt{2} \cdot \log(1/\Gamma)} \ .$$

Substituting this inequality into the equation for l we get

$$m \leq \sqrt{2 \cdot \Gamma} \cdot \frac{\sqrt{2} \cdot \log(1/\Gamma)}{w \cdot \sqrt{\Gamma}} = \frac{2\log(1/\Gamma)}{w} \ .$$

Summing up over all $\log(1/\Gamma)$ induction steps yields the theorem. ■

The Level strategy is only a first step towards adaptive, heterogeneous distribution strategies. Even when applying all the refinements it is not suitable for practical purposes. The main reason is the number of induced levels and the corresponding increase in access time. Furthermore, the refinements make the strategy too complicated to be efficient. The implementation of the levels alone is cumbersome and error-prone.

Nevertheless, the strategy gives a good insight into the nature of heterogeneity. The reduction of a heterogeneous setting to a homogeneous one looks promising not least because of the efficiency of the uniform approaches. Furthermore, we found some evidence that the heterogeneity indeed complicates adaptive approaches. The less the capacities vary, the better the Level strategy. Last but not least, it seems rather surprising that the number of levels can be significantly reduced by giving up a certain percentage of the overall capacity.

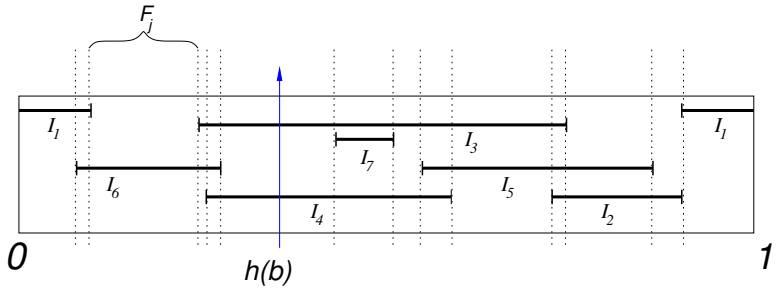

Figure 3.10: The figure illustrates the general approach of the Share strategy. In this example, $S_{h(b)}$ is equal to $\{D_3, D_4\}$ because both corresponding intervals contain $h(b)$.

3.2.2 The Share Strategy

The Share strategy [BSS02] combines many of the advantages of the previously described strategies. Instead of extending the Cut-and-Paste strategy, it is based on the principles of the Nearest Neighbour strategy. We first give an overall description before we analyse the strategy in more detail.

The Share strategy works in two phases. The first phase is a reduction phase which allows the conversion of the heterogeneity problem into a number of homogeneous ones. More specifically, given a block b, the result of the reduction phase is a set $S_{h(b)}$ of disks that are equally likely to store b. So, in the second phase, we can apply a uniform distribution strategy to get the correct disk in $S_{h(b)}$ that is responsible for b. Due to the construction of the reduction not all uniform strategies are suitable. We define some properties that must be fulfilled to achieve the desired results. To denote the call to the uniform distribution algorithm, we use $\textsc{Uniform}(b, S_{h(b)})$. The general approach of the Share strategy is illustrated in Figure 3.10 and the algorithm is given in Figure 3.11.

The reduction phase of the Share strategy is based on two hash functions. We use $h : \{1, \ldots, M\} \to [0,1)$ to map the blocks to random real numbers in the $[0,1)$ interval and $g : \{1, \cdots, N\} \to [0,1)$ to assign real numbers from $[0,1)$ to the disks. As noted above, this phase is very similar to the Nearest Neighbor strategy (see Section 3.1.1) and differs only insofar that the real numbers now represent the starting points of intervals. In addition, two fixed parameters $s \geq 1$ and $1/N \leq \delta < 1$ are used. The former is the *stretch factor* that is used to ensure high probability, and the latter is introduced to simplify the analysis. The values of both parameters will be specified later.

The reduction of the first phase works as follows. Suppose that the capacities of the disks are given by their relative capacities c_i. To simplify some of the proofs, we split some of the disks into virtual disks of size at most δ. This ensures that the

Algorithm SHARE(b)**:**

Input: block b and a data structure containing all intervals I_i
Output: disk that stores b

Phase 1: query data structure for point $h(b)$ to derive the interval set $S_{h(b)}$
Phase 2: $disk_b$ = UNIFORM(b, $S_{h(b)}$)
return $disk_b$

Figure 3.11: The Share algorithm.

disks do not have a capacity which is too high. So, for every disk D_i with $c_i > \delta$ we introduce $\lfloor \frac{c_i}{\delta} \rfloor$ virtual disks D'_i with $c_{i'} = \delta$ and, if necessary, one additional virtual disk taking the rest of D_i. The sum of the relative capacities of all virtual disks for disk D_i is therefore equal to c_i. All remaining disks stay as they are, and they are treated as one virtual disk. The following lemma states the number of virtual disks.

Lemma 30 *Splitting the relative capacities of n disks into virtual disks of size at most δ produces at most $n' \leq n + \frac{1}{\delta}$ virtual disks.*

Proof. To prove the lemma, we simply sum up over all generated virtual disks. For each split disk, we have at most one virtual disk containing the part that is smaller than δ. Together with the non-split disks, their number is at most n. Each of the split disks contributes at most $\lfloor \frac{c_i}{\delta} \rfloor$ virtual disks of size δ. Because the sum of all relative capacities is at most 1 it follows that

$$\sum_{i:c_i>\delta} \left\lfloor \frac{c_i}{\delta} \right\rfloor \leq \sum_{i=1}^{n} \left\lfloor \frac{c_i}{\delta} \right\rfloor \leq \frac{1}{\delta} \ .$$

Summing up completes the proof. ∎

Now, we assign an interval $I_{i'}$ of length $s \cdot c_{i'}$, for some fixed stretch factor s, to every virtual disk $D_{i'}$. Using the starting point computed by g we map this interval into $[0, 1)$ such that it reaches from $g(i')$ to $g(i') + s \cdot c_{i'} \pmod 1$. Note that we view the $[0, 1)$ interval as a ring and, therefore, wrap around the endpoints. We assume that $\delta \leq \frac{1}{s}$ to prevent an interval $I_{i'}$ being wrapped around several times. The only reason behind this is to keep the description and the analysis as simple as possible.

Analysis

In this section, we analyse the behaviour of the Share strategy in regard to the space balance and adaptivity.

First, we look at the quality of the reduction. The starting point is the determination of the size of the stretch factor s, the crucial parameter of the strategy, because it determines not only the distribution quality but also the required space resources.

Throughout the analysis, we treat virtual disks as real disks and, therefore, interchange i' with i. For every $x \in [0,1)$ let $S_x = \{i : x \in I_i\}$ be the set of intervals containing x and $s_x = |S_x|$ be the *contention* at point x, i.e. the number of such intervals. Each endpoint of an interval marks the beginning or ending of a *frame* F_j. Due to our restriction on the size of the virtual disks, we cut the $[0,1)$ interval into at most $2\left(n + \frac{1}{\delta}\right)$ frames. It holds that S_x is unique for every $x \in F_j$. This is important to ensure that the data structure implementing Share has a low space complexity.

Obviously, the correctness of this strategy can only be guaranteed when every point $x \in [0,1)$ is covered by at least one interval I_i w.h.p. The next lemma defines a lower bound for s for which this is fulfilled.

Lemma 31 *For every $n \leq N$ a stretch factor $s = l \cdot \ln N$ with $l \geq 3$ is sufficient to ensure that $s_x > 0$ for every $x \in [0,1)$.*

Proof. For every s and every i it holds that for every $x \in [0,1)$ the probability of x falling into I_i is equal to

$$Pr[x \in I_i] = s \cdot c_i \ .$$

Hence, we get for the expectation value of s_x,

$$E[s_x] = \sum_{i=1}^{n} s \cdot c_i = s \ .$$

Due to the fact that we use a random hash function to map the starting points of the I_i into the $[0,1)$ interval, the I_i are independent of one another. This enables us to apply the Chernoff bounds from Lemma 6,

$$\begin{aligned} Pr[s_x = 0] &= Pr\left[s_x = (1-1) \cdot E[s_x]\right] \\ &\leq e^{-\frac{s}{2}} \\ &= e^{-\frac{\ln n \cdot l}{2}} \\ &= \frac{1}{N^{\frac{l}{2}}} \ . \end{aligned}$$

Because we require $\delta \geq \frac{1}{N}$ the number of possible frames is at most $4N$. It follows that we have at most $4N$ different s_x to consider. So, the probability that there is at least one frame with a contention of 0 is at most $4/N^{\frac{l}{2}-1}$. ■

Next we have a look at the time and space complexity of Share. To measure the space complexity, we assume that one word can hold $\log(\max\{N, M\})$ bits. For the hash functions, we simply choose any hash family from the vast pool of efficient ones.

Theorem 32 *Suppose that the applied uniform distribution scheme has a space complexity of $Space_{Uniform}$ words. Furthermore, let a hash function consume h words. Then, the Share strategy can be implemented such that the position of any block b can be determined in expected time $\mathcal{O}(1)$ using $\mathcal{O}\left(2h + s \cdot \left(n + \frac{1}{\delta}\right) \cdot Space_{Uniform}\right)$ words.*

Proof. We begin by showing the time complexity. To get the expected constant time we apply the same trick as has been used for the Nearest Neighbour strategy. The main idea is to divide the $[0, 1)$ interval into *segments* of size $\min\left\{\frac{1}{n}, \delta\right\}$ and keep a separate search tree for each of the segments. Now, the time to locate the position of an arbitrary block b consists of the time to locate the right segment (which takes constant time) plus the time to find the correct interval I_i inside the segment. Since the total number of frames is at most $\mathcal{O}(n + 1/\delta)$, the expected number of frames overlapping with a segment is constant. Furthermore, the number of intervals in a segment is expected to be constant. So, the call to $\textsc{Uniform}(b, S_{h_x})$ is constant too. Summing up, it follows that the access to an arbitrary block b is implementable in a time of $\mathcal{O}(1)$.

The space complexity follows straightforwardly from the construction. As mentioned in the lemma, it holds that $E[s_x] = s$ for every $x \in [0, 1)$. Hence, for every starting point or endpoint of an interval I_i, the expected number of other intervals crossing x is at most s. Keeping in mind that no interval starts or ends inside a frame, the expected number of intervals in a frame F_j is at most $s + 1$. Since there are at most $2(n + 1/\delta)$ different frames, we need $\mathcal{O}(s \cdot (n + 1/\delta))$ words to store the intervals' information for all frames. Furthermore, $\mathcal{O}(n + 1/\delta)$ words are necessary to store a data structure handling the $\max\{n, 1/\delta\}$ segments. Summing up, the theorem follows. ■

Now we are able to analyse the space balance of the Share strategy. The main idea of the proof is to estimate the number of blocks that fall into an interval I_i depending on the stretch factor s. We do that point-wise, i.e. derive for every point x in the interval the influence it has on the contention of the disks. The following auxiliary definitions are helpful.

Let the *ration at position x* be defined as

$$q_x = \frac{1}{s_x} \ . \tag{3.14}$$

The following lemma proves some useful property of the ration that is needed to show the space balance.

Lemma 33 *For any* $0 < \epsilon < 1$, *if* $s \geq \frac{(6 \ln N)}{\sigma^2}$ *with* $\sigma = \frac{\epsilon}{1+\epsilon}$, *then w.h.p.*

$$q_x \in \left[\frac{1-\epsilon}{s}, \frac{1+\epsilon}{s}\right]$$

for all $x \in [0,1)$. *Furthermore, for any* $x \in I_i$,

$$E[q_x] \leq \frac{(1+\epsilon)}{s} \quad .$$

Proof. We already know from Lemma 31 that $E[s_x] = s$ for all $x \in [0,1)$. Furthermore, the Chernoff bounds (Lemma 6) imply that for any $0 < \epsilon < 1$

$$Pr[s_x \leq (1-\epsilon) \cdot s] \leq e^{-\epsilon^2 \cdot s/2}$$

and

$$Pr[s_x \geq (1+\epsilon) \cdot s] \leq e^{-\epsilon^2 \cdot s/3} \quad .$$

Using $\sigma = \frac{\epsilon}{1+\epsilon}$ we get

$$\begin{aligned} Pr\left[q_x \geq \frac{1+\epsilon}{s}\right] &= Pr\left[\frac{1}{s_x} \geq \frac{1}{(1-\sigma)\cdot s}\right] \\ &= Pr\left[s_x \leq (1+\sigma)\cdot s\right] \\ &\leq e^{-\sigma^2 \cdot s/2} \end{aligned}$$

and

$$\begin{aligned} Pr\left[q_x \leq \frac{1-\epsilon}{s}\right] &\leq Pr\left[\frac{1}{s_x} \leq \frac{1+\epsilon}{(1+2\epsilon)\cdot s}\right] \\ &= Pr\left[\frac{1}{s_x} \leq \frac{1}{(1+\sigma)\cdot s}\right] \\ &= Pr[s_x \geq (1+\sigma)\cdot s] \\ &\leq e^{-\sigma^2 \cdot s/3} \quad . \end{aligned}$$

From these two inequalities it follows that if $s \geq \frac{6 \ln N}{\sigma^2}$, then $q_x \in \left[\frac{1-\epsilon}{s}, \frac{1+\epsilon}{s}\right]$ with probability at least $1 - \frac{2}{N^2}$. Since we have at most $4N$ different frames this is true for all x with probability at least $1 - \frac{8}{N}$.

Now we compute $E[q_x]$ for $s \geq \frac{2 \ln N}{\sigma^2}$ in the case that $x \in I_i$ for some i, i.e. $s_x \geq 1$. It holds that

$$\begin{aligned} E[q_x] &= \sum_{c=1}^{\infty} \frac{1}{c} \cdot Pr\left[q_x = \frac{1}{c}\right] \\ &\leq 1 \cdot Pr\left[q_x \geq \frac{1+\epsilon}{s}\right] + \frac{1+\epsilon}{s} \cdot 1 \\ &\leq \frac{1}{N} + \frac{1+\epsilon}{s} \\ &\leq \frac{1+\epsilon'}{s} \end{aligned}$$

for some constant ϵ' very close to ϵ. Note that N is the maximal number of disks in the network and can be made arbitrarily large. This completes the proof. ■

We are now ready to define the *quota* of each virtual disk, i.e. the number of blocks that are mapped to it.

Definition 34 (Quota) *The quota of virtual disk D_i is defined as*

$$Q_i = \int_{x=g(i)}^{g(i)+s \cdot c_i} q_x \, dx \ .$$

In other words, the quota of disk D_i denotes the number of blocks that are mapped to it if the underlying uniform strategy can guarantee an even distribution. Now we can prove that the quota for a disk differs only by an arbitrary small factor from its share c_i. In fact, we bound the quota using the above bounds for the ration.

Lemma 35 *For all $0 < \epsilon < 1$ it holds that if $s \geq \frac{6 \ln N}{\sigma^2}$ with $\sigma = \frac{\epsilon}{1+\epsilon}$ then*

$$Q_i \in [(1-\epsilon) \cdot c_i, (1+\epsilon) \cdot c_i]$$

for all i w.h.p.

Proof. We use the property of q_i stated in Lemma 33 to prove the lemma. Given a stretch factor $s \geq \frac{6 \ln N}{\sigma^2}$ with $\sigma = \frac{\epsilon}{1+\epsilon}$, it follows from Lemma 33 with a probability at least $1 - \frac{8}{N}$ that for all i,

$$\begin{aligned} Q_i &= \int_{x=g(i)}^{g(i)+s \cdot c_i} q_x \, dx \\ &\leq \int_{x=0}^{s \cdot c_i} \frac{1+\epsilon}{s} \, dx \\ &= (1+\epsilon) \cdot c_i \ . \end{aligned} \tag{3.15}$$

Furthermore, it holds with probability at least $1 - \frac{8}{N}$ that for all i,

$$\begin{aligned} Q_i &= \int_{x=g(i)}^{g(i)+s\cdot c_i} q_x \, dx \\ &\geq \int_{x=0}^{s\cdot c_i} \frac{1-\epsilon}{s} \, dx \\ &= (1-\epsilon)\cdot c_i \ . \end{aligned} \tag{3.16}$$

From Inequality (3.16) and Inequality (3.15) the lemma follows. ■

After characterising the quota of any disk D_i, we are able to prove the space balance property for the Share strategy.

Theorem 36 *Given a storage network $S(n, C[n])$ and p blocks, the Share strategy is space balanced if $s = \Omega(\ln N)$.*

Proof. We prove the space balance property by showing an even stronger property, namely that the number of blocks on every disk differs from the optimal by not more than a factor of ϵ for any $\epsilon > 0$ with high probability.

For the underlying uniform distribution strategy we use the Nearest Neighbour strategy (see section 3.1.1) because it can be shown that for any given n disks and p blocks, the expected number of blocks hosted by disk D_i is within $(1 \pm \epsilon') \cdot c_i \cdot p$, where ϵ' can be brought arbitrarily close to 0.

Now, let the random variable B_x denote the number of blocks b with $h(b) = x$, or more precisely with $h(b) \in [x, x + dx]$ for $dx \to 0$. Similar to the definition of q_x above, let B_x^i denote the number of blocks in B_x that are hosted by disk D_i. Now we define the random variable L_i which denotes the load of disk D_i, i.e. the total number of blocks stored by D_i, by using the above notations. It holds that

$$L_i = \int_{x=0}^{s\cdot c_i} B^i_{g(i)+x} \, dx \ .$$

From the fact that the Nearest Neighbour strategy is space balanced, we can conclude for all $x \in I_i$ that

$$\begin{aligned} E[B_x^i] &\leq \sum_{b,c\geq 1} (1+\epsilon') \cdot \frac{b}{c} \cdot Pr[B_x = b] \cdot Pr\left[q_x = \frac{1}{c}\right] \\ &= (1+\epsilon') \cdot E[B_x] \cdot E[q_x] \\ &\leq (1+\epsilon') \cdot (p \cdot dx) \cdot \frac{1+\epsilon}{s} \end{aligned}$$

where ϵ and ϵ' can be made arbitrarily small. The fact that $E[q_x] \leq \frac{1+\epsilon}{s}$ follows from Lemma 33. Hence, it holds for the expectation value of the load of disk D_i

that

$$\begin{aligned} E[L_i] &= \int_{x=0}^{s\cdot c_i} E\left[B^i_{g(i)+x}\right] dx \\ &\leq \int_{x=0}^{s\cdot c_i} (1+\epsilon'') \cdot \frac{p}{s}\, dx \\ &= (1+\epsilon'') \cdot p \cdot c_i \end{aligned}$$

where ϵ'' could be brought arbitrarily close to 0. Using the same arguments, it can be shown that

$$E[L_i] \geq (1-\epsilon'') \cdot p \cdot c_i$$

where ϵ'' can be made arbitrarily small too. This proves the theorem. ■

It remains to show that the Share strategy adapts to changes in the number of disks. We use the share vectors defined in the introduction for representing heterogeneity. Let the change in the storage network be given by the share vectors $c = (c_1, c_2, \cdots, c_N)$, the capacity distribution before the change, and $c' = (c'_1, c'_2, \cdots, c'_N)$, the capacity distribution after the change, respectively. The Share strategy performs the following *lazy update* strategy.

Let $0 < \lambda < 1$ be some fixed constant. This constant is used to define the *laziness* of the Share strategy, i.e. it only changes the share for disk D_i from c_i to c'_i if

$$c'_i \geq (1+\lambda) \cdot c_i \tag{3.17}$$

or

$$c'_i \leq (1-\lambda) \cdot c_i \ . \tag{3.18}$$

Clearly, applying this strategy can cause the share of the whole system to differ from 1, but the deviation is within $1 \pm \lambda$. So, as long as λ is sufficiently small, this does not effect the results mentioned above. In other words, the Share strategy is still space balanced with respect to the correct capacity distribution.

When the capacities of disks change, a number of blocks need to be redistributed such that, afterwards, the call to $\textsc{Uniform}(b, S_{h(b)})$ results in the correct disk for block b. There are various solutions to solve this task algorithmically, i.e. identify the blocks that need to be moved. However, this is rather an implementation issue, and currently we are only interested in the volume of replaced blocks, i.e. the competitive ratio of the Share strategy.

The following theorem shows the adaptivity of Share.

Theorem 37 *Given a stretch factor of $s = \Omega(\ln N)$ the Share strategy is at most $(2+\epsilon)$-competitive for any $\epsilon > 0$ in case of any change in the characteristics of the underlying storage network.*

Proof. To prove the adaptivity we need to consider two different scenarios. First, the number of blocks p may change. Clearly, none of the disk intervals I_i changes and, hence, no block is moved. The use of random hash functions guarantees that the space balance property is preserved.

Second, the number of disks and consequently, the size of the intervals I_i may alter. Suppose that the change is given by a share vector $c' = (c'_1, \ldots, c'_N)$ and the current configuration, according to the lazy update strategy, is given by $c = (c_1, \ldots, c_N)$. Let the deviation induced by the lazy update rule be expressed by α, i.e. $\alpha \in [-\lambda, \lambda]$ is defined by $\sum_i c_i = 1 + \alpha$. From the size of the stretch factor $s = \Omega(\ln N)$ we conclude that

$$E[s_x] \in [(1-\epsilon)(1+\alpha)\cdot s,\ (1+\epsilon)(1+\alpha)\cdot s]$$

for some constant $\epsilon > 0$ that can be made arbitrarily small. From this it follows that for the expected quota of disk D_i the following property holds

$$E[Q_i] \in [(1-\epsilon)(1+\alpha)\cdot c_i,\ (1+\epsilon)(1+\alpha)\cdot c_i] \quad .$$

Consider now some fixed disk D_i. If c'_i is within $(1 \pm \lambda)c_i$ then c_i does not change according to the lazy update rule. Thus, the number of replaced blocks caused by disk D_i is zero. Otherwise, $c'_i = (1+\beta)c_i$ for some β not in $[-\lambda, \lambda]$. Following the lazy update rule, the share for that disk is set to c'_i. As we know from the introduction of share vectors, we need to consider the case that $c'_i > c_i$, in which case the expected number of replaced blocks is at most

$$\begin{aligned}
(1+\lambda)\left((1+\epsilon)\cdot c'_i \cdot p - (1-\epsilon)\cdot c_i \cdot p\right) &= (1+\lambda)\left(c'_i - c_i + \epsilon(c'_i + c_i)\right)\cdot p \\
&\leq (1+\lambda)\left(\beta\cdot c_i + \epsilon\cdot(2+\beta)\cdot c_i\right)\cdot p \\
&\leq (1+\gamma)\cdot\beta\cdot c_i\cdot p \\
&= (1+\gamma)\cdot p\cdot\left|c'_i - c_i\right|
\end{aligned}$$

for any $\gamma > 0$ if $\lambda > 0$ and $0 < \epsilon < \lambda \cdot \frac{\gamma}{2}$ are sufficiently small. This restriction holds because we know that $\alpha \leq \lambda$ and $\beta \geq \lambda$. For the case $c'_i \leq c_i$, we only need to exchange c'_i and c_i in the above calculation, and we get the same bound of $(1+\gamma)\cdot p\cdot\left|c'_i - c_i\right|$ for the expected number of replace blocks in the worst case.

Summing up, we can conclude that the Share strategy requires the movement of at most

$$(1+\gamma)\cdot\sum_i \left|c'_i - c_i\right|\cdot p$$

blocks when changing the capacity distribution from c to c'. Since the minimum amount of blocks needed to be replaced when staying space balanced is given by

$$\sum_{i:c_i > c'_i}(c_i - c'_i)\cdot p = \frac{1}{2}\cdot\sum_i\left|c'_i - c_i\right|\cdot p \quad ,$$

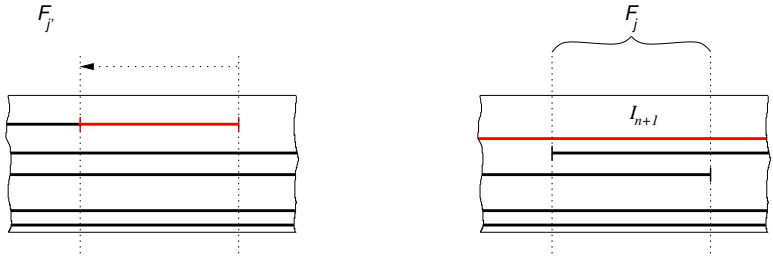

Figure 3.12: The figure depicts the two possible situations of replacement when a new disk enters the storage network. On the left hand side, the figure shows the shrinking of an existing interval. The contents in frame F_j must be moved. The right hand side depicts the addition of a new disk to an existing frame F_j.

the theorem follows. ■

As noted before, we need a uniform strategy to make Share work, because we have to solve a uniform setting in the second phase. Unfortunately, not every such strategy is applicable. Now, we discuss the properties of the underlying uniform strategy demanded by Share.

Clearly, the uniform strategy used has to meet our requirements, i.e. must be space balanced and adaptive. Otherwise, we are not able to prove Theorem 36. Furthermore, in Theorem 37 we have shown that the expected number of replaced blocks is only twice the optimal number. Clearly, the underlying uniform strategy must be able to cope with the replacements appropriately. It turns out that the requirements of adaptivity are in essence the following property.

Definition 38 (monotone) *A distribution strategy is monotone if it can ensure that in the case of a change in the number of disks, blocks are only moved towards (away from) the new (failed) disks.*

The monotonicity is the crucial property when it comes to adaptivity. If we cannot guarantee that blocks only move towards newly introduced disks, the competitive ratio for a strategy is hard to bound.

Looking at the uniform distribution algorithms of section 3.1 we get the following corollaries.

Corollary 39 *The Nearest Neighbour strategy is monotone when adding disks.*

Corollary 40 *The Cut-and-Paste strategy is monotone when adding disks.*

Let's have a closer look at the mechanisms behind the replacements. From Theorem 37 we know that the expected number of blocks that need to be replaced

is twice as high as the number any optimal strategy has to move. After a change there are two main tasks to consider. First, the existing intervals have to shrink when a new disk enters the storage network. Second, blocks have to move into the new interval associated with a new disk. The latter is easier to solve because it is just the addition of a disk in the uniform setting. Let the new disk be denoted by D_{n+1}. Then, we have to add this disk to all frames completely covered by I_{n+1} and the two frames induced by the endpoints of I_{n+1} (see the right hand side of Figure 3.12). This is easy because it corresponds to the addition of a new disk in the uniform setting.

The former is much more difficult and defines a further restriction on the underlying uniform strategy. To see this, let us illustrate the necessary steps in an example. Consider an arbitrary frame F_j. When a new disk is added to the storage network, all existing intervals I_i are shrinking. This means that the right endpoint defining F_j moves to the left and all blocks stored on D_j falling into this area must be replaced. This corresponds to the removement of that disk in the group of disks defining that frame, i.e. in the uniform setting. Hence, the underlying uniform strategy has not only to be monotone when a disk is added but also when a disk is removed. From section 3.1.2 and section 3.1.1 we can conclude the following.

Corollary 41 *The Nearest Neighbour strategy is monotone when removing a disk.*

Corollary 42 *The Cut-and-Paste strategy is not monotone when removing a disk.*

How many blocks are involved in the shrinking of intervals? We know from the analysis that the expected number of blocks moved to the new disk is $c_{n+1} \cdot p$, i.e. it is related to the capacity of D_{n+1}. Hence, the existing intervals all together shrink proportional to c_{n+1} too. So, if we could not preserve the good adaptivity for the underlying uniform strategy, we lose a factor in the adaptivity of the Share strategy.

It follows that we can only assure the adaptivity if we use the Nearest Neighbour strategy as the underlying uniform strategy.

To summaries the chapter, one can said that the Share strategy is a simple and efficient method to solve the heterogeneous distribution problem. Nevertheless, it has some drawbacks. First of all, we can only guarantee that the expected number of blocks per disk meets the heterogeneous capacity distribution. In other words, the number of blocks stored on disk D_i is not highly concentrated around the capacity of D_i. This could only be improved if the stretch factor s is very high which results in a larger space requirement for Share.

The second disadvantage concerns the space consumption. Instead of depending on n, the number of disks currently in the network, the space requirements rely on N, the maximum number of disks.

Nevertheless, in Chapter 5 we show that the quality is sufficient for practical purpose and the limiting factor is not the reduction phase but the underlying uniform strategy.

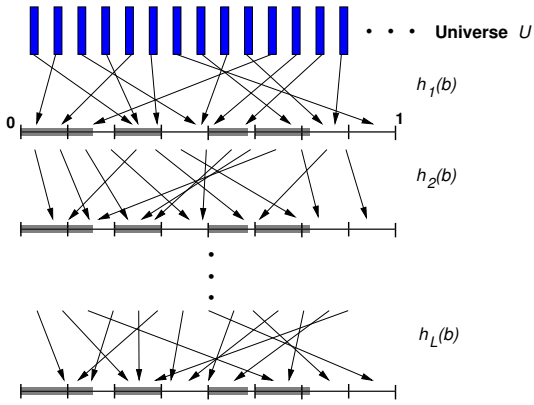

Figure 3.13: The figure illustrates the general approach of the Sieve strategy. The example shows a configuration for 4 disks. Hence, there are 8 scopes. The disks ranges are denoted by shaded areas and mapped to scopes such that at most half of the scopes are covered. A block is stored on disk D_i if its hash value $h_j(b)$ falls in the range of disk D_i for some stage j. All blocks that cannot be places in any stage are stored on the fallback disk.

3.2.3 The Sieve Strategy

The Sieve algorithm overcomes most of the drawbacks inherent to the Share strategy. Nevertheless, this involves a higher number of random hash functions. The most important advantage to the previous distribution algorithm is the absence an underlying uniform subroutine.

The Sieve strategy is organised, as the name suggests, as a multistage filter. In each stage, a certain number of blocks are assigned to disks and only the remaining blocks enter the next stage. In each stage j, we use a hash function $h_j : U \to [0,1)$ that maps the blocks uniformly at random into a $[0,1)$ interval. Suppose that the initial number of disks is equal to n. Then we cut the $[0,1)$ interval into

$$n' = 2^{\lceil \log n \rceil + 1}$$

scopes of size $\frac{1}{n'}$. In other words, we create at least twice as many scopes as we have disks. Now, whole scopes, or portions of them, are associated with *disk ranges*, the mesh of the sieve. Any scope can only be chosen by one disk. If a scope is chosen by disk D_i then any portion of the scope starting at the left boundary of it may be covered by a disk range. The size of the disk ranges will be specified later. If the hash value of a block falls into a range, the block is assigned to the disk owning that range. We call a scope *completely occupied* by disk D_i if the disk range of disk D_i covers the whole scope. A disk may own several scopes but it

Algorithm SIEVE(b)**:**

Input: block b
Output: disk number that stores b

for $i = 1$ **to** L **do**
  set $x = h_i(b)$
  if x is in some covered scope of disk D_i **then return** i
return number of fallback disk

Figure 3.14: The Sieve algorithm.

is only allowed to have one not completely covered scope. The general principle of the Sieve strategy is depicted in Figure 3.13.

The size of the disk ranges is chosen such that any disk covers not more than $\frac{c_i}{2}$ of the whole interval. In fact, the size will slightly deviate from that but at the moment it is acceptable to assume that value. The property ensures that only half of the $[0, 1)$ interval is covered by disk ranges. We could state the following lemma.

Lemma 43 *For any capacity distribution it is possible to assign scopes to the disks in a one-to-one fashion such that each disk can select scopes in* $[0, 1)$ *of total size at most* $\frac{c_i}{2}$.

Proof. Because we know that every disk has only one partly covered scope, the overall number of them is bounded by n. Hence, there remain at least n scopes with an accumulated size of at least $\frac{1}{2}$. These are sufficiently many to completely hold the remaining completely occupied scopes because their accumulated size is at most $\frac{1}{2}$. ■

So, it is possible to find correct assignments of disks to scopes. To access an arbitrary block b, we use the algorithm stated in Figure 3.14. The fallback disk is defined later. The number of stages is equal to L which we set initially to $L = \log n' + f$ for some parameter f which is specified later too.

Analysis

We start the analysis of the Sieve strategy by stating the time and space complexity to access an arbitrary block b.

Theorem 44 *The Sieve strategy can be implemented such that the disk hosting an arbitrary block b can be determined in an expected time of* $\mathcal{O}(1)$ *requiring* $\mathcal{O}(n)$ *words plus the space for L, L-way independent random hash functions.*

Proof. From the fact that the disk ranges cover only half of the $[0,1)$ interval, we conclude that the probability of assigning an arbitrary block b in stage j is $\frac{1}{2}$. Hence, the expected computation time to determine the disk hosting b is $\mathcal{O}(1)$.

The space of $\mathcal{O}(n)$ words is needed to manage the n disk ranges. This is the only information needed for the block access, apart from the L hash functions and some $\mathcal{O}(1)$ information holding constants like n' and n. ■

Now we are ready to fix the still undefined parameters and complete the description of the Sieve strategy. Recall the access of a block from Figure 3.14. We call a block *failed* if it cannot be assigned in the for-loop of the algorithm, i.e. if it cannot be mapped to a disk in any of the L stages. Clearly, the expected fraction of failed blocks is equal to $\frac{1}{2^L}$. Thus, we can expect that any disk, apart from the fallback disk, gets an allotment of

$$c_i \cdot \left(1 - \frac{1}{2^L}\right) \ . \tag{3.19}$$

Clearly, this is not the desired property because, to ensure space balance, we want to have that every disk gets $c_i \cdot p$ of all blocks. To overcome this, we carefully choose a fallback disk and adjust the shares appropriately.

Initially, the fallback disk is set to the disk with the largest capacity. During the service time of the storage network, we change the fallback disk only if its capacity is exceeded by a factor of 2 by some new disk. In this case, the new disk becomes the new fallback disk.

Now, we have to adjust the shares of the disks accordingly to ensure a balanced distribution. Let every disk, except the fallback disk D_f, choose an *adjusted share* of

$$c_i' = \frac{c_i}{1 - \frac{1}{2^L}} \ , \tag{3.20}$$

and let the fallback disk D_f choose an adjusted share of

$$c_f' = \frac{c_f - \frac{1}{2^L}}{1 - \frac{1}{2^L}} \ . \tag{3.21}$$

The summing up of the adjusted shares shows that we still have a valid share distribution;

$$\begin{aligned} \sum_i c_i' &= \frac{1 - c_i}{1 - \frac{1}{2^L}} + \frac{c_i - \frac{1}{2^L}}{1 - \frac{1}{2^L}} \\ &= 1 \ . \end{aligned}$$

Thus, we can state the following theorem.

Theorem 45 *Given a storage network $S(n, C[n])$ and any p blocks drawn from the universe U, the Sieve strategy is space balanced and can guarantee that every disk D_i gets at most*

$$d_i = c_i \cdot p + \mathcal{O}\left(\sqrt{c_i \cdot p \log n}\right)$$

blocks w.h.p. if $\sqrt{3 \cdot c \cdot \frac{\log n}{c_i \cdot p}} < 1$ *for some constant* $c > 0$.

Proof. The space balance follows from the fact that the random hash functions ensure the independent mapping of blocks into the $[0, 1)$ interval. Furthermore, the adjusted shares guarantee that the number of blocks stored on disk D_i is according to its share c_i.

Additionally, the number of blocks placed on a disk is highly concentrated around the expected value. In other words, we get a high probability that any disk contains the expected number of blocks. Because the blocks are mapped independently at random, we are allowed to apply the Chernoff bounds from Lemma 6 gaining the following bounds:

$$Pr[d_i \geq (1+\epsilon) \cdot c_i \cdot p] \leq e^{-\min\{\epsilon^2, \epsilon\} \cdot c_i \cdot \frac{p}{3}} \tag{3.22}$$

and

$$Pr[d_i \leq (1-\epsilon) \cdot c_i \cdot p] \leq e^{-\epsilon^2 \cdot c_i \cdot \frac{p}{2}} \tag{3.23}$$

for every $\epsilon > 0$.

To prove the bound on the concentration, we only need to set ϵ such that high probability is ensured. We use Inequality (3.22) to derive that ϵ for which the probability drops below $\frac{1}{n^c}$,

$$Pr[d_i \geq c_i \cdot p + \epsilon \cdot (c_i \cdot p)] \quad \leq \quad e^{-\min\{\epsilon^2, \epsilon\} \cdot c_i \cdot \frac{p}{3}} \ .$$

From the right hand side we get the following, assuming $\epsilon < 1$,

$$\begin{aligned} e^{-\epsilon^2 \cdot c_i \cdot \frac{p}{3}} &\leq n^{-c} \\ n^c &\leq e^{\epsilon^2 \cdot c_i \cdot \frac{p}{3}} \ . \end{aligned}$$

Setting

$$\epsilon = \sqrt{\epsilon' \cdot \log n} \tag{3.24}$$

leads to

$$\begin{aligned} n^c &\leq e^{\left(\sqrt{\epsilon' \cdot \log n}\right)^2 \cdot c_i \cdot \frac{p}{3}} \\ n^c &\leq e^{\log n \cdot \epsilon' \cdot c_i \cdot \frac{p}{3}} \\ n^c &\leq n^{\epsilon' \cdot c_i \cdot \frac{p}{3}} \\ \epsilon' &\geq 3 \cdot \frac{c}{c_i \cdot p} \ . \end{aligned}$$

Substituting this into Inequality (3.24) concludes the proof. ∎

It remains to analyse the adaptivity of the Sieve strategy. Here, we must not only consider a change in the capacity but also take three other alterations into account for which a replacement of blocks becomes necessary.

1. Change in the capacity (adding or deletion of disks),
2. Increase the number of n' scopes to accommodate additional disks,
3. Change of fallback disk, and
4. Increase the number of stages L to ensure that $\frac{1}{2^L}$ is below the share of the fallback disk.

Because the Sieve strategy relies on random hash functions, there is no need to consider the change in the number of blocks. It still remains space balanced.

We begin the analysis of adaptivity by considering a change in the capacity (case 1 above). Let the share vector $p = (p_1, \ldots, p_N)$ represent the capacity distribution before the change and let $q = (q_1, \ldots, q_N)$ represent the situation afterwards. The Sieve strategy responds to the change as follows: Every disk D_i with $q_i < p_i$ reduces its occupied scopes such that it has only one partly occupied scope afterwards, and every disk D_i with $q_i > p_i$ extends its range such that thereafter it has also at most one partly occupied scope. It can easily be seen that there are always enough scopes left to accommodate the ranges of all disks which increased their ranges as long as we always start with the disks decreasing their ranges.

It remains to bound the expected fraction of blocks that need to be replaced.

Theorem 46 *For any change in the capacity distribution (from one shared vector to another) that does not involve the change of the fallback disk, the Sieve strategy is 2-competitive.*

Proof. For the proof we need to consider the adjusted shares of the disks. Let p'_i and q'_i be the adjusted shares of disk D_i before and after the change, respectively. For any i with $q_i < p_i$ the fraction of the $[0,1)$ interval taken away from disk D_i is equal to $(q'_i - p'_i)/2$ due to the fact that only a portion corresponding to $c_i/2$ is occupied by D_i. For the same reason, the fraction of the $[0,1)$ interval added to any disk D_i with $q_i > p_i$ is equal to $(p'_i - q'_i)/2$. Hence, we can conclude that the expected fraction of blocks participating in the first stage of the Sieve strategy and affected by the change is equal to

$$\frac{||p' - q'||}{2}$$

with

$$||p' - q'|| = \sum_{i=1}^{n} |p'_i - q'_i| \ . \tag{3.25}$$

This holds for all stages in the sieve. Therefore, for the remaining blocks that previously participated in the next or even one of the further stages, the fraction affected by the change is also equal to $\frac{\|p'-q'\|}{2}$.

Now, for any stage $j \in \{1, \ldots, L\}$, let X_j denote the fraction of blocks previously participating in stage j, and let Y_j denote the fraction of these blocks still participating in stage j that have to be replaced. Note that we can exclude the failed blocks because any of these that still fail are placed to the same fallback disk. Using this notation, the number of blocks that need to be replaced can be expressed by

$$Y = \sum_{j=1}^{L} Y_j \ . \tag{3.26}$$

Since it holds that for a given X_j the expectation value of Y_j is defined by

$$E_{X_j}[Y_j] \leq \frac{1}{2} \cdot \|p' - q'\| \cdot X_j \ , \tag{3.27}$$

we get for any given $X_1, X_2, \ldots, X_L$ that

$$\begin{aligned} E_{X_1,\ldots,X_L}[Y] &= \sum_{j=1}^{L} E_{X_i,\ldots,X_L} Y_j \\ &\leq \frac{1}{2} \cdot \|p' - q'\| \sum_{j=1}^{L} X_j \ . \end{aligned}$$

From the fact that only half of all participating blocks in each stage are placed, we know that the expected value on the number of participating blocks in any stage j is defined by

$$E[X_j] = \frac{1}{2^{j-1}} \ .$$

Hence, it follows that the expectation value of the number of replaced blocks can be expressed by

$$\begin{aligned} E[Y] &\leq \frac{1}{2} \cdot \|p' - q'\| \sum_{j=1}^{L} \frac{1}{2^{j-1}} \\ &= \left(1 - \frac{1}{2^L}\right) \|p' - q'\| \\ &= \left(1 - \frac{1}{2^L}\right) \frac{\|p - q\|}{1 - \frac{1}{2^L}} \\ &= \|p - q\| \ . \end{aligned} \tag{3.28}$$

Since we know that any space balanced strategy has to replace at least a fraction

$$\frac{\|p-q\|}{2}$$

of all blocks the lemma follows from Inequality (3.28). ■

Next we consider the adaption (increase) of the number of scopes n' (case 2). This only happens if the restriction of

$$n' = 2^{\lceil \log n \rceil + 1}$$

is violated due to some new disk. In this case, we simply subdivide each scope into two new ones. Afterwards, every disk still has only one partly occupied scope and, hence, nothing has to be replaced.

Let's consider the change of the fallback disk (case 3). Note that this only happens if a new disk has a maximum share of at least twice the share of the current fallback disk. Denote by D_1 the current fallback disk and by D_2 the new one, w.l.o.g. Assume further that the number of disks in the network is n. Then the new fallback disk D_2 has a share of at least $\frac{1}{n}$. Furthermore, at the time D_1 was selected as a fallback disk, its share was at least as large as the share of D_2. Hence, the total amount of changes in the shares of D_1 and D_2 since then must have been at least $\frac{1}{2n}$. Changing from D_1 to D_2 involves the movement of an expected fraction

$$\left|\frac{c_1 - \frac{1}{2^L}}{1-\frac{1}{2^L}} - \frac{c_2}{1-\frac{1}{2^L}}\right| + \left|\frac{c_2}{1-\frac{1}{2^L}} - \frac{c_1 - \frac{1}{2^L}}{1-\frac{1}{2^L}}\right| \leq \frac{3}{2^L - 1}$$

of all blocks. If we choose the parameter f for the number of stages carefully (sufficiently large), we get

$$\frac{3}{2^L - 1} \ll \frac{1}{2n} \ .$$

Hence, we can 'hide' the replacement work for the change of the fallback disk inside the movements needed to adapt to the necessary capacity changes of the system.

The last change we consider is the growth of the number of stages L (case 4). This might be necessary once in a while if many new disks are introduced into the storage network and the fallback disk might become incapable or unwilling to store a fraction $\frac{1}{2^L}$ of all blocks. By pursuing the following strategy, we ensure that this does not happen.

Whenever the share of a fallback disk is less than $\frac{1}{2^{L-t}}$ for some integer t we increase the number of levels by one. Clearly, this causes the replacement of some

blocks, but we show that this overhead could be 'hidden' inside the amount of blocks needed to be replaced due to changes in the capacity distribution.

Let D_k be the fallback disk that requires an increase in the number of stages from $L-1$ to L. If no such disk exists, we choose the initial fallback disk D_j. Furthermore, let D_l be the current fallback disk that calls for an increase in the number of stages from L to $L+1$. From our assumption we know that D_k had a share of at least $1/(2^{L-(t+1)})$ when it became a fallback disk. Assume that D_l overtook the role of the fallback disk from D_k. Obviously, its share had to be twice as large as the share of D_k. Furthermore, its share was at most as large as the share of D_k when D_k became the fallback disk because otherwise we would have chosen D_l as the fallback disk. Hence, the total amount of changes in the shares of D_k and D_l must have been at least $1/(2^{L-t})$. With similar arguments it is possible to show the same for a longer history of fallback disks.

Changing the number of stages from L to $L+1$ involves the movement of an expected number of at most

$$\left(\sum_{i\neq k}\left|\frac{c_i}{1-\frac{1}{2^L}}-\frac{c_i}{1-\frac{1}{2^L}}\right|\right)+\left|\frac{c_k-\frac{1}{2^L}}{1-\frac{1}{2^L}}-\frac{c_k-\frac{1}{2^L}}{1-\frac{1}{2^L}}\right|\leq\frac{2}{2^L-3} \tag{3.29}$$

blocks. Again, if we choose t and $f\geq t$ to be sufficiently large, we get

$$\frac{2}{2^L-3}\ll\frac{1}{2^L}\ . \tag{3.30}$$

It follows that, again, we can 'hide' the effort to change the number of stages inside the replacements necessary to accommodate changes in the distribution of shares.

This leads to the following theorem.

Theorem 47 *Given a storage network $S(n,C[n])$ and p blocks, the Sieve strategy is $(2+\epsilon)$-competitive where $\epsilon>0$ can be made arbitrarily small.*

Proof. The theorem follows from the above reasoning. ∎

The Sieve strategy is not only space balanced and adaptive but could also be applied in a distributed way. If every disk has complete knowledge about the capacity distribution, every disk could consistently and independently react to changes. It only has to derive the correct changes in the assignment. Then, every disk can independently check if there is any block falling into one of its former scopes that calls for replacement. Similar techniques could be applied to any of the situations that change the configuration.

CHAPTER 4

Application Storage Virtualisation

In the last chapters we gave the theoretical background for the efficient use of storage networks. We devised two algorithms that are capable of handling heterogeneous storage networks in terms of space and access balance, efficient data access, and adaptivity. Now we look at some application of our strategies. This chapter closes the gab between theory and practice. More specifically, after abstracting the theoretical model from practical considerations such as disk access and network properties, we come back to practice and give details of an actual implementation.

The primary reason for the introduction of storage networks, from the practical point of view, is the centralisation of storage resources, i.e. the concentration of all storage devices into one single network. This has a number of advantages, like low maintenance costs, easier and simpler administration, higher flexibility, and faster response to changes. However, these advantages only come to pass if a number of properties are fulfilled. In general, we must be able to separate the physical representation from the logical view of the storage resources. Why is that so? Centralising all storage resources means to manage the storage as one large entity. This suggests the flexibility to use the storage space as it is needed by applications until there is non left. Furthermore, new applications may be introduced with new storage needs. However, this can only be achieved if the applications have an abstract view of the storage resources which changes only according to their demands. Such an abstraction of storage is called *storage virtualisation*.

There are three main tasks to be solved to implement storage virtualisation.

1. Associate a group of storage devices to each logical view.
2. Distribute data elements over these groups.
3. Define mechanisms to expand and shrink these groups.

Obviously, any strategy implementing all the above points defines an abstraction of physical storage resources. Note that the last property is needed to achieve flexibility.

In the business world storage virtualisation is closely related to *storage area networks* (SAN). SANs, like storage networks, consist of storage devices and/or storage subsystems connected by a high-speed special-purpose network. Data servers, which handle the access to storage, are connected to the network. Different to storage networks, the software and hardware basis to handle data retrieval, data migration, sharing of data, etc. are part of the storage area network. Due to that fact, SANs are mostly monolithic systems. Usually, their compatibility is restricted to vendor dependent products.

The virtualisation conditions noted above are the foundations for the application of adaptive distribution schemes in that field; these strategies are capable of mapping any number of data blocks to any number of storage devices. Furthermore, the adaptivity ensures that only a minimal number of blocks has to be migrated to transform the storage network from one configuration into another. In other words, applying for instance the Share strategy to a changing group of storage devices enables storage virtualisation.

In this chapter, we describe the implementation of our storage virtualisation approach. We begin with some arguments concerning the general way of realising virtualisation. There are a number of different approaches which differ in the place where the data mapping is implemented. This discussion leads to a general description of our virtualisation solution. A number of issues have to be addressed before the Share strategy can be used, like the size of blocks, or how to preserve consistency. Then we give an introduction of its main components. One part is directly implemented into the kernel of the Linux operating system. To get an idea about the major coherences, we introduce the mechanisms of data access to block devices under Linux including the general file system structure.

4.1 Storage Virtualisation

Storage virtualisation is an abstract concept separating the physical from the logical view of storage resources and it is implemented on top of a storage network. In general, the virtualisation corresponds to a special mapping of data to disks. Inside the storage network, such a mapping could be integrated into several different components. The only restriction is that it has to happen somewhere along the access path from the connecting server to the storage devices. There are five major variants.

1. **Symmetric** or **In Band**
2. **Asymmetric** or **Out of Band**

3. **Server-based**
4. **Switch-based**
5. **DASD-based (Direct Attached Storage Device)**

The first two solutions make use of so-called *appliances* which implement the virtualisation strategy. In the symmetric case, all data paths to the attached storage devices pass through one of the appliances. This eases the implementation because one has direct access to all data requests. However, the scalability of the system is costly due to the fact that more appliances are needed.

In the asymmetric solution, the appliances are outside the direct data path. They keep track of the attached devices and give their information back to the accessing servers. The third approach is very similar to the asymmetric case. Instead of an appliance, the server themselves have to keep the virtualisation information.

The switch-based solution is, from our point of view, the most promising approach. Here, the virtualisation is incorporated into the network hardware itself. It is part of the storage area network without limiting its scalability. Unfortunately, the implementation can be done by major vendors only.

The last approach uses only subsystems that are directly attached to a host, like SCSI devices attached to a bus. It is the most simple solution and has the major drawback of lack of scalability. Nevertheless, it is the most widespread solution because of its simplicity. In this work, it is only noted for historical reasons.

For our implementation we choose the asymmetric approach. The major reason for that is the possibility to implement it entirely in software. That would not be possible for the symmetric and switch-based solution. Furthermore, we are interested in the scalability. In the asymmetric solution the limiting factor is the performance of the appliance. However, we have designed the adaptive distribution algorithms such that the space resources as well as the needed computational power are low. Hence, the effect on the appliance should be, at least in theory, minimised. The next section gives the details of how Share can be used to implement a storage virtualisation.

4.2 Virtualisation with Share

The key to storage virtualisation lies in a flexible data distribution strategy. We chose the Share strategy (defined in Section 3.2.2) for that purpose. It is not only adaptive and space balanced but also easy to implement [BMS+03].

Nevertheless, the conversion of the randomised distribution strategy into a working storage virtualisation is not straightforward. There are various problems to be solved.

The first, and most important, concerns the granulation of data elements. The smallest accessible units are data blocks[1]. The block interface is capable of accessing any arbitrary data block on a block device, e.g. magnetic disk drive. Under Linux the size of a block is always a multiple of 512 byte. Above the block interface, in terms of data granulation and abstraction, is the file system interface. A file, in the broader sense, is an arbitrary collection of data blocks. Hence, the file system is responsible for managing and organising files which includes the handling of structure information (directories) and the management of free and used data blocks. File systems can be fused together by the process of mounting, i.e. a complete file system is integrated into an existing one at some special point as a directory.

At which abstraction level do we apply the Share strategy? The block interface has the advantage of being clearly defined, in essence it is the reading and writing of arbitrary blocks to devices. Using this interface for our purpose entails the definition of a new device and the implementation of a new device driver. Any valid file system could then be built on that device. However, there are two disadvantages to that approach. First, the Share strategy, or at least the procedures preserving consistency, have to be implemented inside the kernel. Second, we have to keep track of all accessed blocks. This is necessary due to the underlying randomisation. While the distribution strategy gives us the correct disk to store the data block we have no information about the position of that block on the disk[2]. Therefore, we have to keep track of the physical positions of each accessed block. Considering the resources of SANs (tera byte to peta byte), this results in a tremendous waste of resources.

On the other hand, we could move the abstraction one level up and use the file system level as an implementation platform. That does not mean that a distributed data element is a file but that a new file system must be developed. The drawbacks of the implementation of a new file system are incompatibility and complexity. The restriction to one operating system can be avoided only, if the file system is implemented on each operating system separately. Furthermore, the file system interface to applications is far more complex than the block interface. Above all, the mentioned granulation problem still remains.

To keep a higher compatibility, we choose the block interface. However, instead of using data blocks, we coarsen the accessible data units by introducing the concept of *extents*.

Definition 48 (Extent) *An extent is a consecutive collection of data blocks on a disk. The size of an extent is fixed initially but may change over time.*

So, the distribution strategy takes extents as inputs and distribute them over participating disks. The gain in space resources is acquired by a loss in the space

[1] In general, it is possible to use smaller units but it is highly inefficient on disks (see Section 1.1.1).
[2] Any possible criteria to define some pattern is destroyed by the randomisation.

balance. The bounds from Section 3.2.2 have to be scaled to extents rather than pure data blocks. Nevertheless, we believe that an extent size of about 1024 kilobytes is sufficient to achieve good results. The actual size of the extents should depend on the size of the smallest disk and the size of the whole SAN. Furthermore, it should be possible to adapt the coarseness if the system grows.

Having fixed the building blocks for the Share strategy, we are now able to substantiate the access point to the storage network.

Definition 49 (Virtual Device) *A virtual device is a block device that consists of a collection of extents.*

In contrast to the theoretical view of storage networks, like taking the whole network a one entity, we like to divide the network into storage pools. That might appear inconsequential but it is very reasonable. As noted in the introduction, fault tolerance plays an important role in the design of distribution schemes. There are a number of different levels of reliability that demand different safeguarding against failure. To see this, consider two virtual devices where the first uses mirroring to protect its extents and the other uses no fault tolerant scheme. If a disk D_f fails, the likelihood that it is hosting extents from one of the devices is the same for both of them. However, that is not consistent with the decision of the administrator to use a fault tolerant distribution scheme for that virtual device. For that reason, we associate *storage pools* to virtual devices.

Definition 50 (Storage Pool) *A storage pool is a collection of disks and/or storage subsystems. It is unique such that a device can only participate in one pool.*

With the above definitions, we fix the external view of the storage network. Any application using the virtualisation is provided with new virtual devices that consist of a collection of extents drawn, according to the Share strategy, from a number of disks arranged in a storage pool.

Another important problem to solve is consistency. On the one hand, we need a global view of the storage area network and the used extents. This part of the virtualisation is put into the appliance. On the other hand, the order of data accesses to data blocks must be preserved. This is essential for data consistency. However, during the redistribution of extents, as a result of a changed system, the access to some extents is restricted. This task falls to the servers in general and to the device driver in particular.

Now we are ready to characterise the solution in a bit more detail. The virtualisation consists of four main components (see Figure 4.1):

1. **Graphical User Interface (GUI)**
   This module is the administration tool for the SAN.

2. **Device Driver**
   The driver manages all data accesses on the server side.

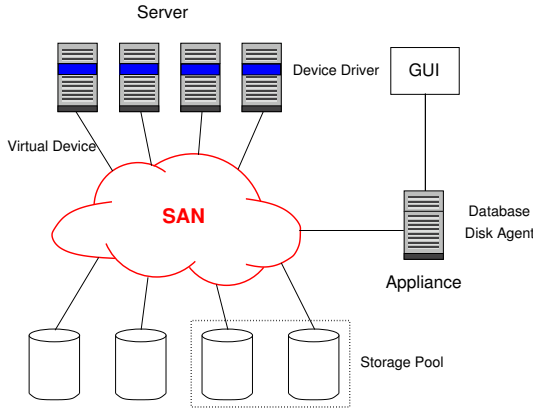

Figure 4.1: Overview of the virtualisation approach. The server and the GUI access the SAN from the outside (LAN). The database and the disk tools are placed at the appliance and there is communication with the servers. The servers themselves access the SAN via virtual devices. To do so, the driver module is linked to the Linux kernel.

3. **Database**
   In the database we store all the relevant information about the storage network itself. This is our consistent view of the storage network.

4. **Disk Tools**
   The disk tools are a bunch of service tools responsible for replacement of extents (disk agent), detection of new/changed devices (device detect), as well as communication between server and database.

The GUI is Java-based and can be installed on any machine having access to the SAN. Its main task is the administration of the storage network, i.e. creation of storage pools, assignment of disk to pools, creation of virtual disks, etc. All information from the GUI is sent to the appliance and entered into the database.

The appliance is hosting the SQL-based database and the disk tools, i.e. it stores all metadata needed for the virtualisation. In the case of a change in the configuration, the disk agent gets informed and starts the replacement of extents.

As noted before, the data access is established via the device driver in each server. When they are linked to the kernel they contact the database and request the current configuration. To handle the data accesses they maintain a data structure in which all extents for each device are stored. Whenever the driver gets an address for which it has no extent, it requests it from the database and insert it into its local structure.

In the next section, we give a general overview of device drivers under Linux. This does not only clarify the access to the storage network but also defines the basics for the experiments in the next chapter.

4.3 Device Driver under Linux

As the implementation platform for the accessing server, we choose the Linux operating system [RC01]. The reason is the open source concept of Linux which enables us to integrate a new device driver into the kernel. As noted before, the interface to access the storage network is a new device. In this section, we give a short overview of Linux device drivers. This is not only necessary to illustrate the implementation of the driver module but also essential to understand the experiments given in the next chapter.

A device under Linux is a unit that is capable of managing data. There are three kinds of essentially different devices:

1. **Character Devices**, which support the access in a character-based manner. An examples of a character device is the printer interface *lp*.

2. **Block Devices** access a device, like magnetic disk drives, in blocks of fixed size.

3. **Network Devices** communicate through an interface with other hosts. Instead of reading and writing procedures the device communicates with routines for receiving and sending data packets.

The device system relies heavily on the file system concept. This means that (almost) every device is represented as a file[3]. The access to a file is characterised by three main operations: *read*, *write*, and *seek*. This is essentially the same functionality that a device needs. Hence, any device driver has to support these operations.

Access to devices is granted by so-called device files which can be found in the special directory */dev* in the main file system. In contrast to all other files in the system, they possess two more parameters, namely the *major* number and the *minor* number. These numbers tell the kernel which device driver it has to call when such a file is accessed. While the major number characterises the driver, the minor number differentiates the devices for the same driver. The tuple $(major, minor)$ is unique within the system.

[3] In general, directories are nothing more than files.

4.3.1 The Linux File System

All data accesses on a Linux system have to pass the file system because every access to a device is made via the device files. This calls for a standardised interface for all devices. Therefore, the file system is divided into a general interface (virtual filesystem switch – VFS) and concrete implementations. From the application's point of view, any file is only a stream of data. It is the task of the file system to handle this stream efficiently.

A file is strictly divided into metadata and the 'plain' data itself. The most important data structure is the *inode* structure. It contains all the relevant management information, like owner, access rights, size, etc. Furthermore, it consists of pointer structures to data blocks (simple, double and triple indexed). The access to a file is handled via the inode of a file.

Any implementation of a file system, like EXT2, EXT3, Reiser [vH02], lies on top of the virtual file system. This means they all have to realise the functionality of the VFS. Apart from this, the main task of the proper file systems is the efficient resource administration. This includes the allocation of free blocks, handling of metadata, and keeping the consistency of the file system. To get the data from the appropriate device the block interface is used (see Section 4.3.2).

Like the VFS itself, any implementation of a file system has a very similar structure. Exemplarily, we give an overview of the EXT2 layout in ascending order of the data blocks. The zeroth block is reserved for the *boot block*, i.e. the data block containing all the relevant code to boot the operating system. Next is the *superblock* which keeps all metadata. The superblock is crucial for the integrity of the file system. Therefore, copies of it are placed at different positions of the underlying device. To have a faster access to the file structure, the inodes representing the file system structure are stored in a cluster after the superblock. The rest of the device consists of data blocks including index blocks.

File systems could be fused together at any directory, called mount point, using the *mounting* operation. The internal representation of the directory, namely its inode structure, is overlaid by the uppermost level (root directory) of the fused file system. Using that operation, very large file systems stored on different devices can be built, i.e. building separate file systems on each of the devices represented by the device file and fusing them together with mount operations.

This technique is used for our virtualisation too. As noted before, our interface is the block interface. Hence, we introduce a suitable driver together with a new device file in */dev*. Using this driver, we are able to build an arbitrary file system onto that device and mount it to any mounting point in the file system tree.

4.3.2 Access to Block Devices

The device realising the access to the storage network is a block device because, essentially, it has to behave like a disk drive and it delivers data in blocks from the participating disks.

Block devices access the data in chunks for fixed size. This is crucial for all devices whose access time is almost independent from the size of the accessed data. However, the access to data on block devices might be too slow. The Linux operating system tries to close this gap by using parts of the main memory to cache accesses to block devices. Hence, all read and write operations do not go directly to the appropriate devices but are buffered in a part of the main memory, called the *buffer cache*.

The buffer cache is organised by the buffer head structure. The buffer heads consist of some fields needed to access the device, like block number, size, device representation, etc. and a pointer to the data itself. They are organised into a number of lists, like free heads, heads with changed data fields, or locked buffer heads. There is only an access to the corresponding device if the buffers are not yet read or its data field differs from the analogous data block stored on the device[4]. The caching effect is achieved by writing a certain percentage of all dirty buffers periodically.

All requests for data from a block device pass through the buffer interface. They reach the kernel, more specifically, the device driver responsible for the device hosting the corresponding file system, in the form of read/write requests for buffer heads. This is the place where our driver intercepts. It maps the given block number to an offset inside an extent and changes the device of the buffer head to the device on which the extent is stored. To do so, the driver has to get and preserve information about used extents. The mapping itself is realised by the Share strategy. Instead of applying it inside the kernel, the driver communicates with the database to get the relevant information. The reason is consistency. If some information is lost, the driver can always contact the database to get the relevant information.

The buffer cache technique improves the device access significantly because a high percentage of requests are answered from main memory. However, it also makes the virtualisation inside the kernel quite complex. As long as there is no change in the storage pool, the mapping is easy. During a replacement phase, we have to ensure the correct order of accesses to the device. This is accomplished by using a locking mechanism. Whenever an extent is moved, we contact all accessing server and demand authorisation. Inside the driver, requests to this extent are blocked. After completion of the replacement, the driver gets informed and unlocks the extent and all blocked requests are passed on to underlying

[4] The buffer head is called *dirty* in that case.

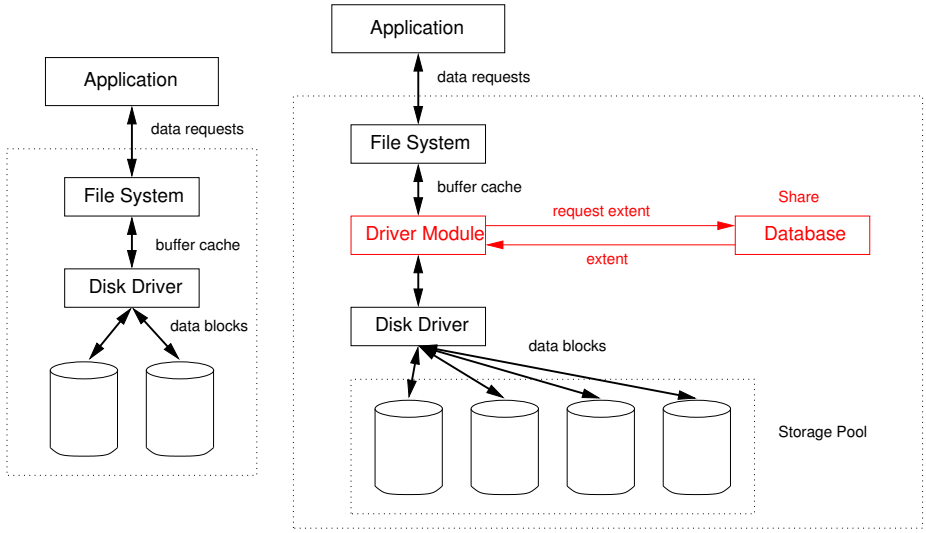

Figure 4.2: The figure shows the data access from the applications point of view in principle. The left hand side illustrates the standard access in a Linux system. When an application reads/writes a file, the underlying file system uses the buffer cache to get the appropriate data. A buffer head structure is filled by using the appropriate disk driver. The right hand side shows the access for our virtualisation approach. Here, we have to communicate with the database to access a data block. The Share strategy maps the extent to a disk. The allocation of space is done by using the first free space of that device.

device drivers (to perform the actual disk access). Notice that this part of the virtualisation is very sensitive to errors because failures in the direct data access usually result in a deadlock inside the kernel, rendering the machine unusable.

For an application using the storage network, it is only interesting to minimise the delay of data accesses. Hence, the device driver plays an important role in the performance of the data delivery. How much overhead compared to normal device accesses do we have to sacrifice for the virtualisation? The principle data access is shown in Figure 4.2. The left hand side shows the general data access under Linux. When a file is read of written, the application uses file system functions. This results in data requests to the buffer cache. These requests are answered by underlying device drivers (like SCSI layer) accessing the disk drives.

The right hand side of Figure 4.2 illustrates the data access in our virtualisation approach. The beginning and the end of the access chain are equivalent in both approaches. The main difference lies in the fact that the driver does not know a priori on which disk a data block can be found. Therefore, the driver module

intercepts the data path after the file system and handles the mapping of block addresses. First, it finds the extent containing the accessed data block. If this extent was accessed before, the necessary fields in the buffer head structure are changed and the request is passed on to the disk driver. If this was not the case, the extent is requested from the database. The driver sends a message via the communication network to the appliance hosting the database. In case the extent exists, the database sends back the necessary extent information. A new extent is found by applying the Share strategy on the extent address and allocating the first free space on that device.

It follows that either the driver itself, the performance of the database, or the bandwidth of the communication link could present a performance bottleneck.

4.4 Replacement

The replacement phase is realised by a cooperation of all main parts of the virtualisation, namely device driver, database, and disk agent. The actual replacement of an extent is implemented inside the disk agent. The full replacement algorithm is given in Figure 4.3. It is not the most efficient implementation, yet, but it tries to increase robustness.

The replacement algorithm has a number of performance relevant steps. Our goal is to minimise the time an extent is locked inside the driver because all requests for that extent are blocked and, hence, the application accessing it. Nevertheless, we need to inform the servers about the change otherwise the accesses might get lost. For that reason, we decided to replace each extent separately. This increases the communication volume but decrease the delay of blocked accesses.

The overhead for the device driver during a replacement phase depends on the number of replaced extents. For each of these, the driver has to find the correct entry in its internal structures and set its status to replacement mode. Furthermore, it has to unlock the extent when the replacement is finished, which takes the same number of operations.

The communication with the disk agent is handled outside the kernel to improve performance[5]. Information is transferred via the *ioctl*-interface (I/O control). Using that interface, it is possible to hand over arbitrary data structures to a device driver. The communication task is realised by one of the disk tools running in user mode. It accepts all communications to the driver module.

[5]The kernel is the performance bottleneck because any application accesses the device driver.

Algorithm REPLACEMENT**(***vddev***):**

Input: virtual device *vddev* that is subject to changes
Output: replacement for all extents that changed their location

```
get a list of used extents for vddev from database
for all extents in list do
   derive location of extent using Share
   if location of extent changed then
      reserve space on new device (database)
      inform accessing server (driver) about replacement
      wait for authentication from server
      replace extent block-wise
      inform server about new location
      update database for new extent
   end if
end for
```

Figure 4.3: The general (simplified) replacement algorithm. The algorithm checks for every used extent if its location has changed and if that is the case copies it to the new location.

CHAPTER 5

Performance Results

This chapter details the experiments conducted to test the performance of our storage virtualisation approach. Some of the results presented here can be found in [BMS+03].

Why are the theoretical results not sufficient to guarantee a good practical behaviour? There are two main reasons for that. First, we use a randomised algorithm for the core of our virtualisation. While we prove some of the theoretical results with high probability, some issues hold only for the expectation value. Therefore, the distribution can vary significantly. Furthermore, the quality depends of parameters, like the stretch factor. Hence, the feasibility of the strategy, regarding the size of the parameters versus distribution quality, can only be evaluated practically.

Second, a virtualisation is only practical if its induced overhead is acceptable. This validates not only our solution but also provides insight into further improvements. If we are able to characterise the overhead as detailed as possible, we can look for replacements of the performance-killing components and, hence, improve the solution significantly. Both arguments give sufficient justification for a detailed examination of our implementation.

We start this chapter with some tests concerning the quality of the Share strategy for which a test system is not required. This evaluates the implementation of the strategy and assesses the distribution quality. The implementation itself is straightforward. We fill a static data structure with all the important information (starting points and end points of intervals, hash values for the Nearest Neighbour strategy) in a preprocessing step. For the generation of pseudo-random numbers we use an implementation of the SHA1 (US Secure Hash Algorithm 1) [EJ01]. The evaluation is done with a number of artificially generated randomly chosen inputs. Mainly, we are interested in the distribution quality of the Share strategy. We measure the quality by its maximal and mean deviation from the optimal.

The focus of the experiments lies on the induced overhead and stress of all components of the solution, mainly the device driver. This is crucial because we intercept the data path very close to the disks where requests are handled very fast and only small amounts of data are delivered. Larger amounts of data give the operating system some freedom to schedule these requests efficiently.

We test real-world scenarios with file system benchmarks[1] because the file system defines the access point to our virtualisation. To apply such benchmarks, we only have to build a file system onto a virtual device and mount it to some directory. Accessing this directory ensures that the virtual device is accessed.

The chosen benchmark, bonnie [Bra96], is a freely available, very simple program. We modified the original code slightly to meet our requirements and get more information concerning the induced overhead. Bonnie generates large files in a given directory and performs different read and write operations on it;

1. Create the files by writing each character separately (with *putc*).
2. Read the files character-wise (using *getc*).
3. Write the files in a more sophisticated way in block-sized chunks (using *write*).
4. Read the files block-wise (using *read*).
5. Rewrite the files in a block-wise manner, i.e. read a block, modify it slightly and write it back.
6. Seek an arbitrary block (using *lseek*).

All but the last test are sequential, i.e. the files are accessed in consecutive order. It follows that Bonnie captures the maximal possible performance rather than average throughput because the number of seeks is minimised. Only the last test resembles, in our point of view, a real-world situation.

Bonnie measures not only the throughput but also the CPU utilisation. In general, the higher the throughput the higher the CPU utilisation because the CPU is busy generating new requests if they can be answered very fast.

5.1 The Test System

The principle structure of the test system used is pictured in Figure 5.1.

The test system consists of two Pentium server connected to an FC-AL array with 8 fibre channel disks. Both server have 2 Pentium II processors with 450

[1] A good overview of existing file system benchmarks under Linux can be found in [BRS01].

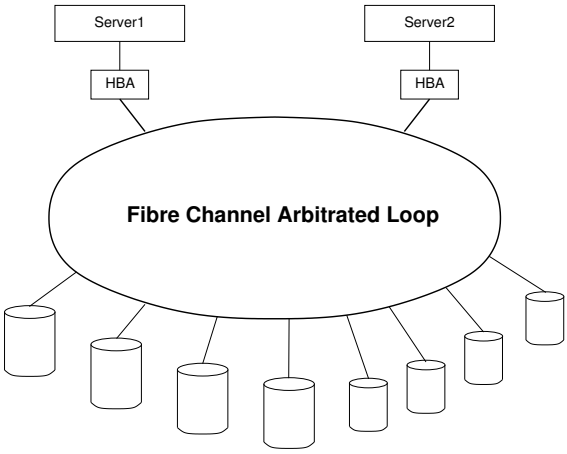

Figure 5.1: The principle structure of the test system.

MHz and 512 kilobyte cache. Moreover, they have local access to a mirrored disk drive containing the operating system and all relevant management information.

Both server run a Linux 2.4.18 kernel (Red Hat) and use the gcc version 3.2.2. as the C-Compiler. The access to the disks is enabled by a host bus adapter (HBA). Access to the disks is managed by the qla2300 driver.

The FC-AL array consist of four 17 gigabyte and four 35 gigabyte fibre channel disks. They are connected with a 1 Gigabit switch. Each disk is partitioned into one partition covering the whole disk.

5.2 Artificial Workloads

We begin the experiments by evaluating the quality of the Share strategy. The tests are divided into homogeneous and heterogeneous configurations. All experiments are conducted with a varying the number of disks. Due to the special definition of heterogeneous behaviour, we cannot run these experiments for all possible number of disks.

We apply two measures to assess the quality of the distribution. The first method, called implementation quality, derives the best possible distribution. As noted in Section 3.2.2 the Share strategy literally cuts the $[0, 1)$ interval into ranges and maps these ranges to disks. Hence, the quality could be measured by deriving the *weight* of each disk, i.e. the sum of ranges it is responsible for. Taking the first phase (reducing heterogeneity), this is easily done by charging each disk with the size of every frame it participates in divided by the number of disks in that frame.

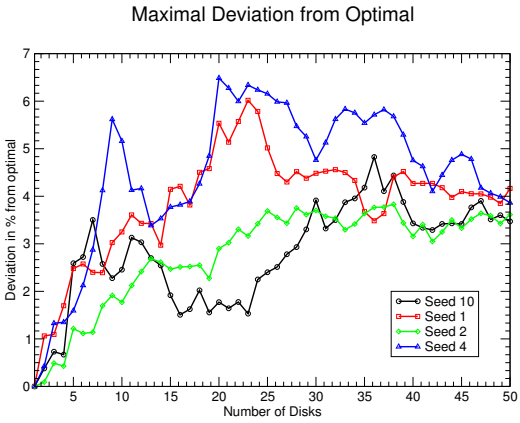

Figure 5.2: The figure shows the maximal deviation of the first phase for a number of different seeds in a uniform setting, i.e. it is assumed that the ranges in each frame are perfectly distributed among all disks in the frame in the second phase. The y-axis states maximal difference from the optimal share in percent.

Note that this derives only a measure for the first phase because we assume that the ranges are perfectly distributed in the second phase. To get the quality of the whole strategy, we determine the share of each disk in the second phase (simply by summing up the size of each of its arcs) and scale the weights of the first phase accordingly. The quality of the distribution strategy is the better, the closer the weight of each disk D_i is to its share c_i (its normalised capacity).

In a second batch of tests, we apply the Share strategy to a number of items and derive the deviation from the optimal. This gives some insight into the quality of the hashing of items prior to the reduction phase. From the analysis of the strategy we know that the distribution quality depends on the number of accessed items. By varying the number of items, the experiments give an answer to the question of how many items are needed to come close to the maximally achievable distribution. Obviously, this distribution is given by the weights of the disks.

The used inputs are distributed according to different patterns. This includes regular ones, such as a consecutive number of items or consecutive but separated by a constant stride, and randomly chosen pattern.

5.2.1 Homogeneous Setting

As noted before, the quality of the Share strategy depends on the stretch factor s. From the analysis of the strategy we know that $s = c \cdot \log n$ suffices to guarantee the theoretical results. However, instead of changing the stretch factor for every number of disks, we set $s = 50$ for the first set of experiments. Therefore, the constant c varies between 50 and 10. This gives us some idea about the effect of the size of the constant c.

Implementation Quality

The Share strategy is a randomised approach. Therefore, the distribution quality depends on the random function and, hence, on the chosen seed for that implementation. In a first set of experiments we compare the quality for a number of different seeds.

The maximal deviation from the optimal share for different random seeds is depicted in Figure 5.2 and Figure 5.3. The first graph considers only the first phase of the Share strategy, i.e. the size of each frame is proportionally ascribed to each of its participating disks. In the second figure, the size of each frame is divided according to the covered range of a participating disk in the second phase. The tie breaking of the disks participating in one frame (uniform phase) is done by the Nearest Neighbour strategy.

It can be seen that the choice of the random seed does not play an important role in the first phase of the Share strategy. Although a seed produces good results for a certain number of disks, it fails for others. For more than one hundred different seeds, the maximal deviation is always less than 7%. Surprisingly, the effect of the constant c is less than expected. While for most seeds (e.g. seed 2) the deviation increases, the distribution gets better with an increasing number of disks. We conclude that the effect of varying the constant factor is not very pronounced in the first phase.

A slightly different situation can be found when both phases are considered. Here, the second application of a randomised algorithm increases the deviation significantly. In the worst case, we would have to waste about a quarter of the storage resources on one disk. Interestingly, the effect of the constant stretch factor is more apparent when both phases are considered. There is a constant increase in the maximal deviation with growing n. This can be explained by the behaviour of the Nearest Neighbour strategy. Its quality also depends on the number of disks n. However, we apply it only to resolve the disks in a frame, which consists of a small number of disks. Therefore, the effect is much less apparent.

How likely is the worst case measure of the maximal deviation? We could shed some light on that subject by looking at the mean deviation presented in Figure 5.4 and Figure 5.5. Considering only the heterogeneity resolving phase,

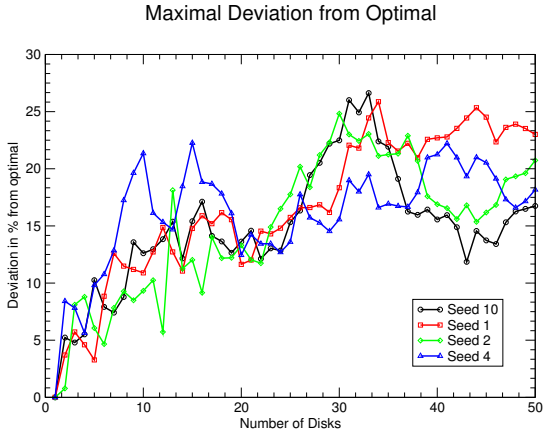

Figure 5.3: The figure shows the maximal deviation of the Share strategy, using the Nearest Neighbour approach to resolve ties in the second phase, for a number of different seeds in a uniform setting. The y-axis states the maximal deviation from the optimal share in percent.

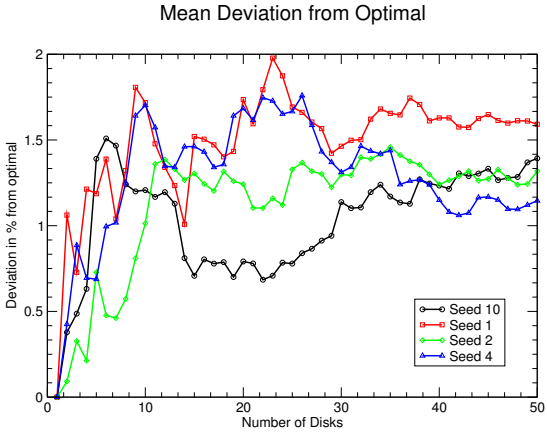

Figure 5.4: This diagram shows the mean deviation of the first phase for a number of different seeds in a uniform setting, i.e. it is assumed that the ranges in each frame are perfectly distributed among all disks in the frame in the second phase. The y-axis states the mean difference from the optimal share in percent.

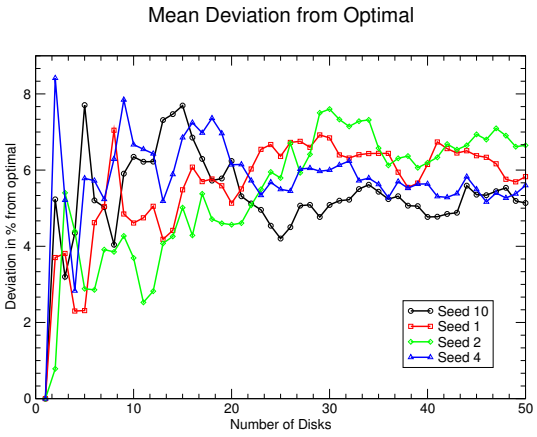

Figure 5.5: The figure shows the mean deviation of the Share strategy, using the Nearest Neighbour approach to resolve ties in the second phase, for a number of different seeds in a uniform setting. The y-axis states the mean deviation from the optimal share in percent.

the mean deviation is always less than 2% which is very promising. The strategy as a whole stays close to 6% with increasing n. Notice that the graph flattens to the end. This is due to the smoothing effect of larger n on the mean value. The difference in the deviation gets leveled out the more disks are participating.

More interesting than different seeds is a varying stretch factor because it significantly influences the quality as well as the needed space resources. The maximal deviation for increasing stretch factors are shown in Figure 5.6 and Figure 5.7 and the mean deviations are depict in Figure 5.8 and Figure 5.9. It can be seen that the deviance varies significantly for varying factors. While we achieve good results for stretch factors of about 50, we decrease the deviation with a factor of 200 to below 1% for the first phase. Again, the effect in not that pronounced for the whole strategy. Here, we get down to 10% loss but the difference between larger stretch factors decreases. This is be due to the Nearest Neighbour strategy, as we explain in the next paragraph.

Looking at the mean deviation in Figure 5.8 and Figure 5.9 it can be seen that the overall distribution quality of the Share strategy is very good. The first phase achieves a mean deviation of less than 0.4% for a stretch factor of 200. The charts give clear evidence of the influence of an increasing stretch factor. While the first phase almost perfectly follows the changing value of the stretch factor, the results for the full strategy are made murky by the influence of the Nearest

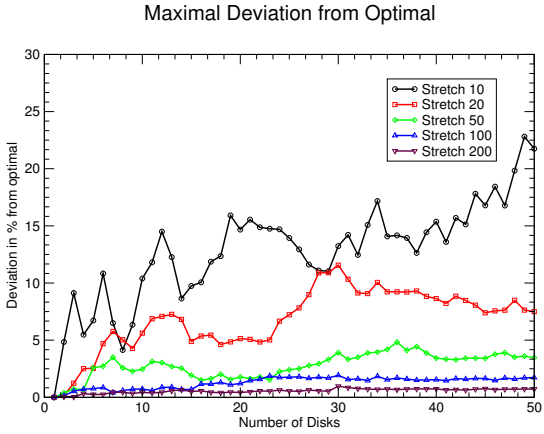

Figure 5.6: The figure shows the maximal deviation from the optimal share in percent for a uniform setting and varying stretch values. Only the first phase is considered, i.e. it is assumed that the second phase distributes perfectly.

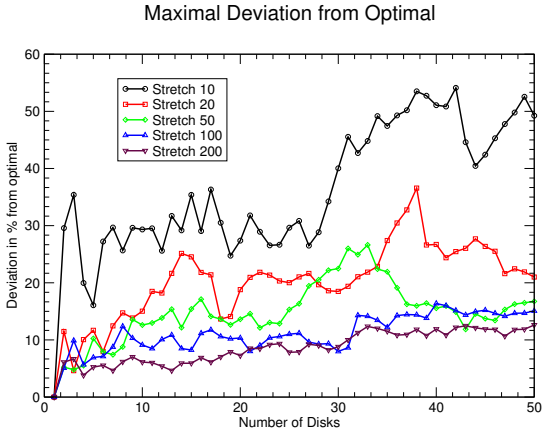

Figure 5.7: The maximal deviation from the optimal in percent is shown for the Share strategy. The disks are uniform and the Nearest Neighbour approach is used for the second phase.

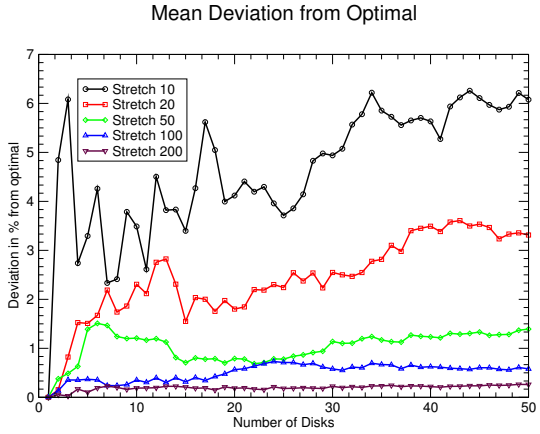

Figure 5.8: The figure depicts the mean deviation from the optimal for the first phase of the Share strategy in a uniform setting in percent. It is assumed that the second phase distributes the shares perfectly.

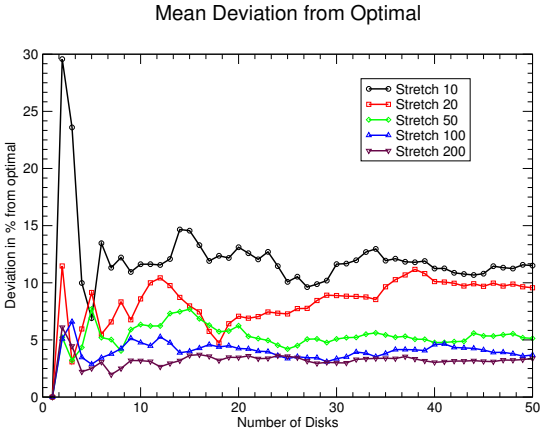

Figure 5.9: The mean deviation from the optimal share of the Share strategy in a uniform setting is shown in percent. The tie breaking in the second phase is realised by the Nearest Neighbour strategy.

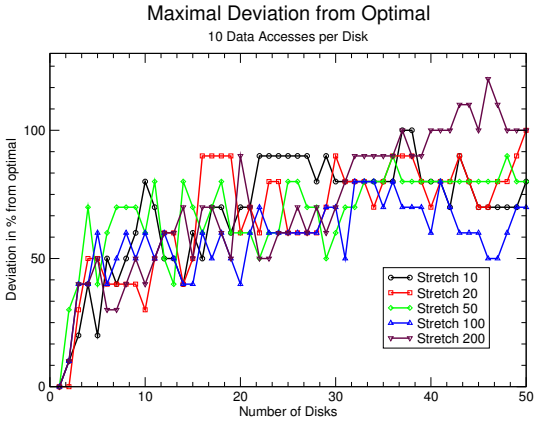

Figure 5.10: In this experiment we access a number of items via the Share strategy in a uniform setting such that, optimally, each disk gets 10 items and measured the maximal deviation in percent.

Neighbour strategy. This strategy cannot improve beyond a 3% barrier. This matches earlier tests that evaluated the quality of the Nearest Neighbour strategy [Sch00b]. Furthermore, a larger number of copies in the second phase has little effect on the quality. Consider the curves for a stretch factor of 50, 100 and 200. Although they can be clearly distinguished in the first phase, they look very similar when the full strategy is regarded. The improvement is only in the range of 2% to 3% (Figure 5.9).

To summarise this first part of experiments, it could be said that the strong part of the Share strategy lies in the first phase, i.e. in resolving the heterogeneity. We cannot judge the heterogeneous distribution quality, yet. Crucial for the quality is a good choice of the stretch factor. Furthermore, the distribution gets better the more disks participate, even if the stretch factor is not adapted. This smoothing effect with increasing n sets in earlier when the stretching factor is large.

Access Quality

As noted before, the disk weights limit the quality of the Share strategy, i.e. they set an optimal. In this section we are interested in the number of items needed to reach this optimal.

In a second set of tests, we access a varying number of items k in different uniform settings and with changing stretch factors. To evaluate the hashing of accessed items (identified by their addresses), we keep all parameters fixed and use different access patterns, e.g. k arbitrary consecutive items, k items which are

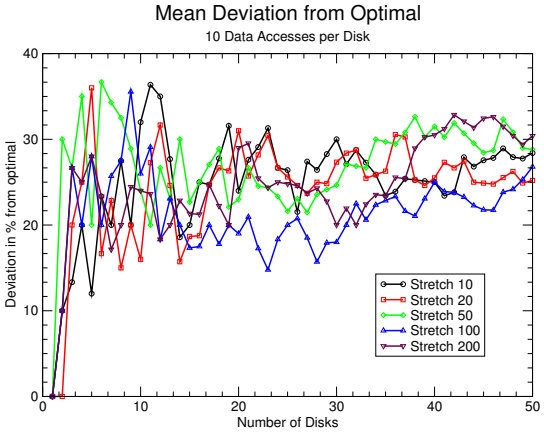

Figure 5.11: This figure shows the mean deviation from the optimal in a uniform setting when the number of items per disk is set to 10.

separated by a constant stride, or k arbitrarily chosen items. It turns out that maximal deviation for varying patterns is insignificant. In other words, the hashing of items breaks the pattern sufficiently to assume an even distribution. Hence, we conduct all following experiments for the first k items in the address space.

The number of items accessed is scaled with the capacity of the system, i.e. the optimal number of items per disk is set to 10, 100 and 1000 regardless of the number of disks. Our choice needs interpretation because it has an influence on size of an extent in the full system. Consider an off-the-shelf disk with a capacity of 10 gigabyte. If we demand a maximum of 10 extents per disk, the size of an extent is bounded from above by 1 gigabyte. In other words, we would try to achieve saturation with that extent size. If we allow 1000 extents per disk, the system would not saturate until the size of the extents reaches roughly 10 megabyte. Apart form that, using only a few items per disk yields to information about the distribution when only a small number of items is accessed. These tests give us an idea about the expected coarseness of the distribution.

As can be seen in Figure 5.10 and Figure 5.11, the distribution quality for only 10 items per disk is not very promising. We get around 80% maximal deviation and about 30% mean deviation. While these values are rather high, one should keep in mind that the strategy relies on randomisation, i.e. the distribution improves with growing n and with a increasing number of items. Furthermore, a bad distribution is only problematic if the network approaches saturation because then we would have to dimension the network rather lavishly. When we set the extent size to a smaller value, the only bad effect is less parallelism when a batch of items is accessed in parallel. We conclude that 10 items are not sufficient to

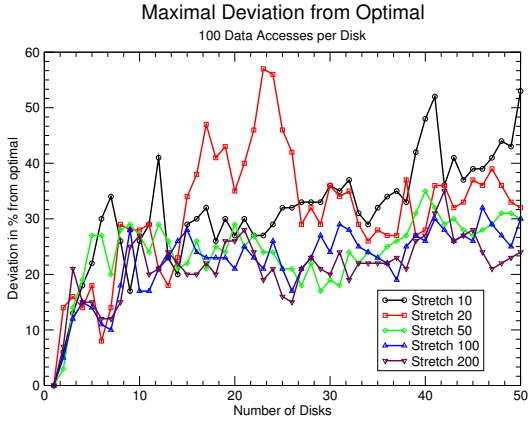

Figure 5.12: This figure shows the maximal deviation from the optimal for a uniform setting. Optimally, each disk would get 100 items.

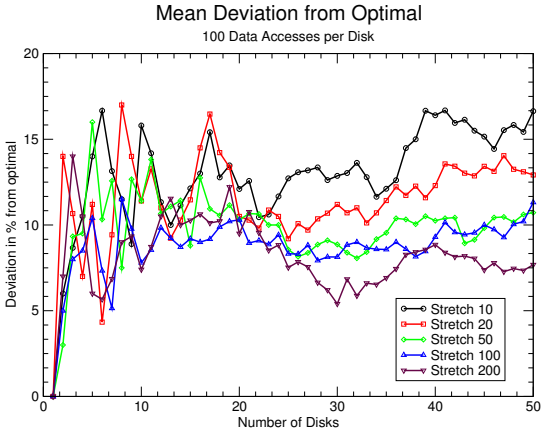

Figure 5.13: The mean deviation from the optimal in a uniform setting is depicted in this figure. Optimally, each disk would get 100 items.

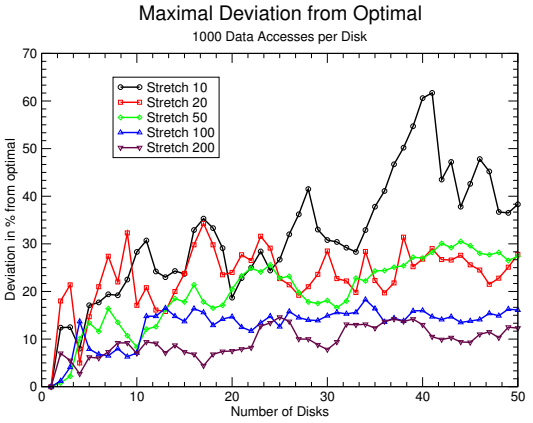

Figure 5.14: In this figure the maximal deviation from the optimal in a uniform setting is depicted. The optimal number of items per disk is set to 1000.

achieve a good random distribution.

Conspicuously, the size of the stretch factor is not important for the distribution. This is due to the fact that the fluctuations are too large to be compensated. Looking at the results with 100 items per disk (Figure 5.12 and Figure 5.13), this effect lessens but, still, the differences are not as pronounced as the weights suggest. The maximal deviation sways around 30% and the mean deviation lies between 16% and 8% depending on the stretch factor.

The best results are achieved with 1000 items per disk (see Figure 5.14 and Figure 5.15). The maximal deviation stays around 10% and the mean deviation is below 4% for a stretch factor of 200. Moreover, the effect of varying stretch factors is clearly visible.

Now let us compare Figure 5.7 with Figure 5.14, the best achievable distribution. It is apparent that the curves stay in the same value range, e.g. around 10% for a stretch factor of 200. Hence, we conclude that 1000 items per disk are sufficient to reach the theoretical bound induced by the disk weights. Furthermore, the maximal deviation of about 10% seems acceptable for practical purpose. This is further underpinned by the fact that the mean deviation is around 4%.

To summarise this section, it can be said that the Share strategy is capable of achieving its optimal distribution, i.e. the inherently best distribution, when the number of items per disk is about 1000. Assuming a disk capacity of about 10 gigabyte, this results in an extent size of up to 10 megabyte.

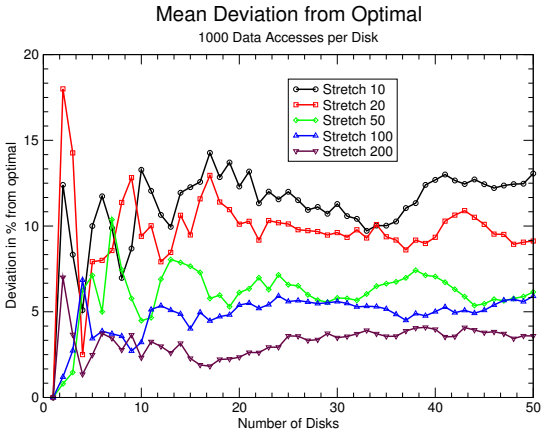

Figure 5.15: This figure shows the mean deviation from the optimal in a uniform setting. The number of items per disk is set to 1000.

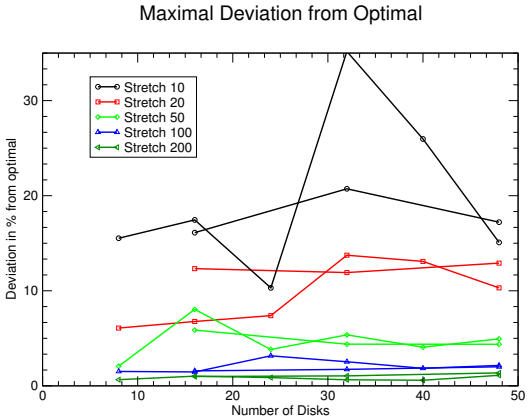

Figure 5.16: The figure shows the maximal deviation from the optimal for a heterogeneous setting. The 'shorter' curves are made for H_{16} ($t = 16$) and the 'longer' ones stand for the H_8 ($t = 8$) case.

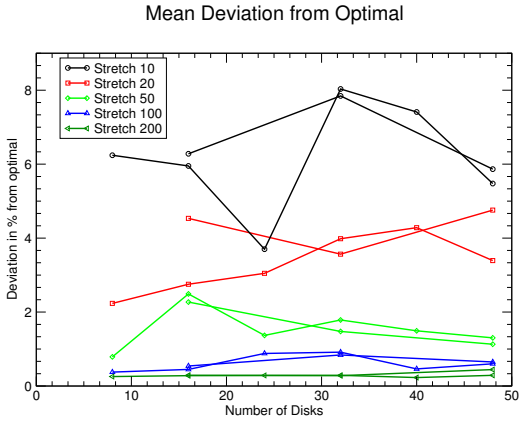

Figure 5.17: The figure depicts the mean variance from the optimal in a heterogeneous setting. The 'shorter' curves corresponds to the H_{16} case while the 'longer' represents H_8, i.e. 8 different capacity classes.

5.2.2 Heterogeneous Setting

Heterogeneity poses a great challenge for storage networks because each disk must be treated according to its capacity. Therefore, we like to uncover the influence of heterogeneity on the distribution quality. To get as much information as possible, we vary the capacities of participating disks significantly and call this property the *skewness* of the storage network.

We test a number of different heterogeneous settings to learn about the influence of the skewness. Each configuration consists of t different *capacity classes* $H_1, \ldots, H_t$ containing equal sized disks. To get a very high skewness, we set the capacity of disks in class H_i to be twice as large as the capacity of disks in class H_{i-1}. We test 4 different settings consisting of 2, 4, 8 and 16 capacity classes. Clearly, we could not test all these configurations for all numbers of disks. Instead, we started with 1 disk per class and increased each class by one disk until the configuration reaches 50 devices. Hence, for 16 classes we get only results for 16, 32 and 48 disks.

Implementation Quality

As in the uniform settings, we first look at the weights generated by the two phases of the Share strategy. The maximal and mean deviation for the first phase are depicted in Figure 5.16 and Figure 5.17, respectively. As before, we observe that the size of the stretch factor plays an important role in the distribution quality. For instance, a factor of 200 achieves a maximal deviation of less than 2% and a mean deviation of less than 0.5%. Both are very promising results. Furthermore, the

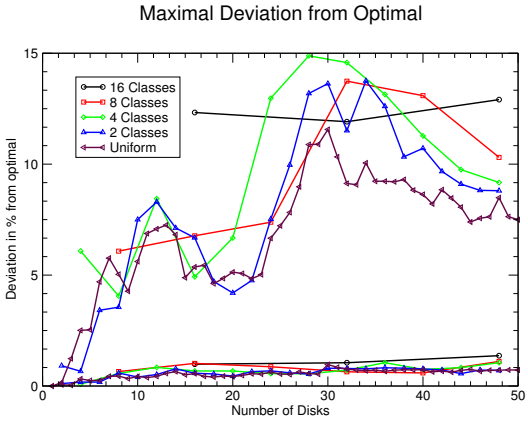

Figure 5.18: This picture shows the maximal deviation from the optimal of the first phase for four different heterogeneous settings. The upper set of curves are generated with a stretch factor of 20 while the lower set is achieved with a stretch factor of 200.

figures show the curves for two different settings, namely for 16 capacity classes and 8 capacity classes. Surprisingly, the graphs look very similar for the same stretch factors, i.e. there is no substantial difference between the different settings.

This becomes even more obvious in Figure 5.18 and Figure 5.19. Here, the maximal and mean deviation for all four heterogeneous settings are shown for two different stretch factors. The topmost set of curves corresponds to a stretch factor of 20 while the bottommost set is achieved with a stretch factor set to 200. Especially the second set shows very clearly that the differences between the settings are marginal. The graphs are crisscrossing and they differ only by less than 1%. Nevertheless, if we compare the heterogeneous results with the uniform ones, there seems to be an order. For both stretch factors the uniform curve achieves almost always the smallest deviation while the setting with 16 capacity classes tends to be the worst scenario. However, these differences are very small and the fluctuations in the random distribution weaken them even further.

Next consider the deviation for the both phases, as shown in Figure 5.20 and Figure 5.21. The clear distinction between the different stretch factors is not as pronounced as in the first phase only. This is consistent with the uniform experiments. As noted before, the reason lies in the quality of the underlying Nearest Neighbour strategy. Still, the different heterogeneous settings behave similarly for fixed stretch factors. This becomes even more apparent when we look at all four settings (Figure 5.22 and Figure 5.23). The figures show the curves for a stretch factor of 20 (topmost four curves) and a stretch factor of 200

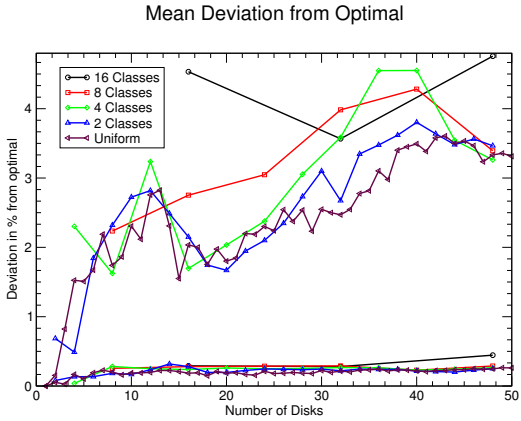

Figure 5.19: The figure depicts the mean deviation from the optimal of the first phase for four different heterogeneous settings. The upper set of curves are made with a stretch factor of 20 while the lower set is made with a stretch factor of 200.

(bottommost four curves). The latter set differs only by about 6% for the maximal deviation and by roughly 2% for the mean deviation.

The order induced by the skewness of the setting, as observed in the first phase of the strategy, vanishes completely, i.e. is superimposed by variations in the Nearest Neighbour strategy.

Overall, we conclude that the Share strategy handles heterogeneity efficiently. The deviations for the full strategy cannot be distinguished from the homogeneous results.

Access Quality

We performed the same experiments as in the uniform case to evaluate the quality when a certain number of items are accessed. When using only 10 and 100 items per disk, there is no observable difference from the curves corresponding to the uniform setting, i.e. the different curves could barely be distinguished.

The maximal and mean deviation when we demand 1000 items per disk can be seen in Figure 5.24 and Figure 5.25, respectively. Each figure consists of three different sets of curves. The topmost set corresponds to a stretch factor of 10. Here, the fluctuation is highest and there is no detectable difference between the different heterogeneous settings. The middle set is achieved using a stretch factor of 50. We observe that the graphs do not differ as much as in the first set. Still, the lines crisscross and there is no apparent order. The bottommost set is generated using a stretch factor of 200. Again, there are no significant differences between the different settings; the curves are concentrated within a few percent.

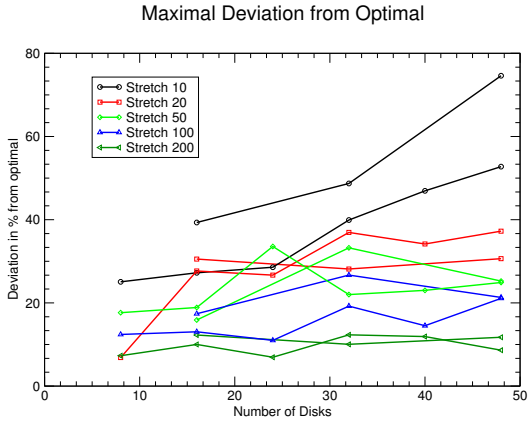

Figure 5.20: The maximal deviation from the optimal for four different heterogeneous settings is shown. The 'shorter' curve for each stretch factor corresponds to a 16 classes setting (H_{16}) while the longer represents H_8, i.e. 8 different classes.

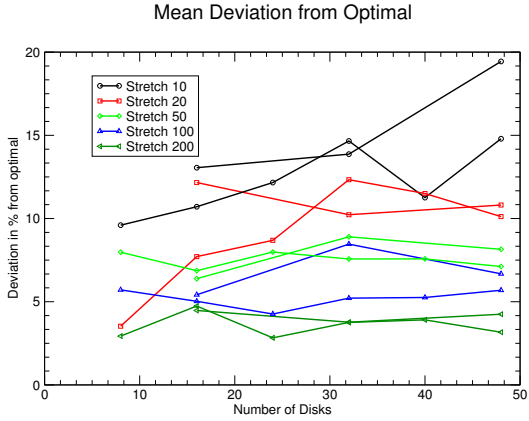

Figure 5.21: The picture shows the mean deviation for two different size groups and varying stretch factors in a heterogeneous setting. The 'shorter' curves correspond to the H_{16} case while the 'longer' graphs represent the setting of 8 different capacity classes (H_8).

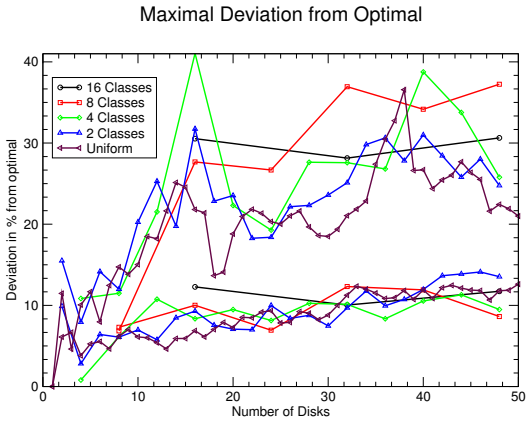

Figure 5.22: The figure illustrates the influence of different heterogeneous settings on the maximal deviation from the optimal. The upper set corresponds to a stretch factor of 20 while the lower set is made with a stretch factor of 200.

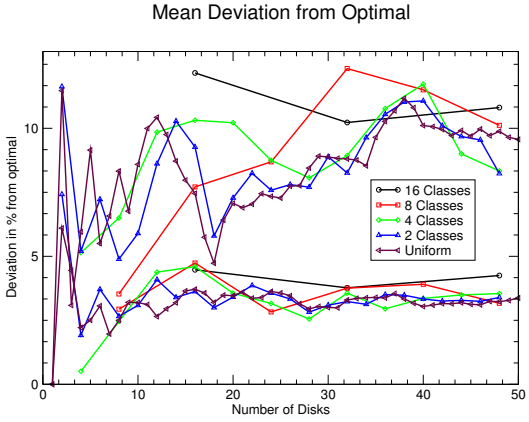

Figure 5.23: The figure shows the mean deviation from the optimal for four different heterogeneous settings. The upper set corresponds to a stretch factor of 20 while the lower set is achieved with a stretch factor of 200.

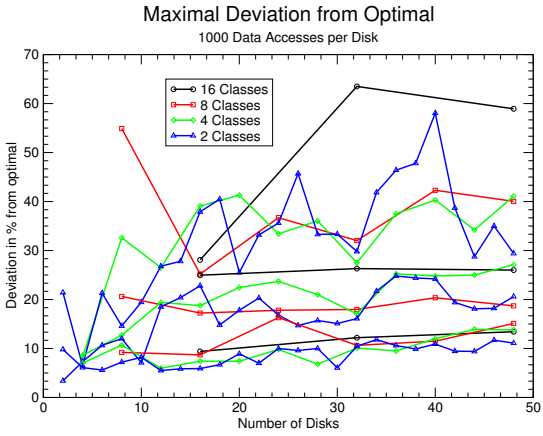

Figure 5.24: The figure depicts the influence of different stretch factors as well as the influence of different heterogeneous settings on the maximal deviation from the optimal. The uppermost set of four graphs corresponds to a stretch factor of 10 while the next four curves were achieved with a stretch factor of 50. The bottommost 4 curves correspond to a stretch factor of 200.

If we compare the values for 1000 items per disks with the weights of the full strategy (Figure 5.22 and Figure 5.23 with Figure 5.3 and Figure 5.5), we notice no significant difference. In other words, 1000 items per disk are sufficient to achieve the best possible distribution.

The important lessons learnt in this section can be formulated as follows: The Share strategy is very suitable for heterogeneous storage networks because the deviation of the distribution cannot be distinguished from the one made in a homogeneous network. Furthermore, the quality scales with the stretch factor. While the maximal deviation in the first phase of the Share strategy drops below 1%, the Nearest Neighbour strategy in the second phase is not capable of preserving this distribution quality. With the highest stretch factor we achieve a maximal deviation of around 10% and a mean deviation of roughly 4%.

5.2.3 Space Consumption

The last sections have shown that we need to set the stretch factor to a value greater than 50 to achieve a good distribution of items when using the Share strategy. However, the theoretical analysis shows that large stretch factors result in a higher space consumption for the Share data structure. In this section, we investigate the needed resources to implement the Share strategy.

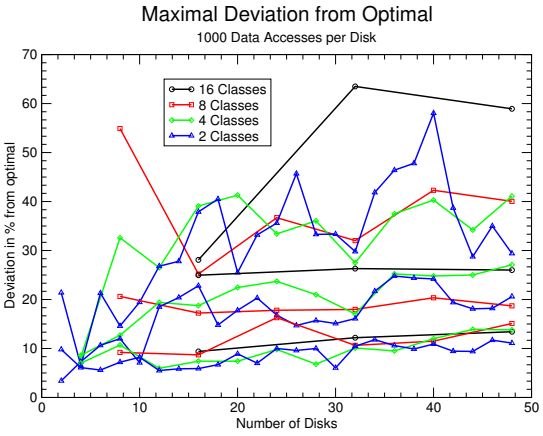

Figure 5.25: This figure shows the mean deviation from the optimal for different stretch factors and varying heterogeneous settings. The uppermost set of curves corresponds to a stretch factor of 10. The next set of four graphs is made with a stretch factor of 50 while the bottommost set of curves is achieved using a stretch factor of 200.

We start by looking at the information that is needed to implement the Share strategy. Recall that we use a static data structure that contains all the information needed to access any given extent.

To realise the first phase, each disk maps a number of intervals, represented by a start and a end point, into the $[0, 1)$ interval. This number depends linearly on the used stretch factor s only, as is shown in the analysis (Theorem 32). The second phase implements the Nearest Neighbour strategy. Hence, we need s real numbers per disk to realise the second phase. Summing up, we conclude that the space consumption is linear in the number of disks as well as the size of the stretch factor.

The space consumption for the Share data structure is shown in Figure 5.26. The expected linearity is preserved by our implementation. For a stretch factor of 10 and 50 disks, the data structure needs only 8 kilobytes of memory space while a stretch factor of 200 results in a consumption of about 160 kilobyte. Nowadays, the size of main memories, even for off-the-shelf machines, is at least 512 megabytes and, hence, the consumption of the Share data structure can be neglected even for larger stretch factors.

Number of disks	1	10	20	30	40	50
Stretch 10	0.3	1.7	3.3	4.8	6.4	8.0
Stretch 20	0.7	3.5	6.6	9.8	12.9	16.0
Stretch 50	1.7	8.7	16.5	24.4	32.1	40.0
Stretch 100	3.4	17.4	33.0	48.7	64.3	79.9
Stretch 200	6.7	34.8	66.0	97.3	128.6	159.9

Figure 5.26: The table lists the space consumption of the data structure implementing Share for various stretch factors and different numbers of disks in kilobytes.

5.3 Overhead Measurements

In the last section, we investigated the theoretical quality of the Share strategy in the context of storage virtualisation. Now we look into real-world scenarios. As noted before, we access the virtual devices via the file system. In the following experiments we build a file system on the virtual device and run a file system benchmark (bonnie [Bra96]) to derive the performance of our solution. The focus is not the overall performance in terms of running time but the overhead induced by the virtualisation. The reference point for all experiments is the performance of one disk (labelled SDA), without the interception of our driver. Clearly, this is exposing the overhead. To cancel out local deviations, we average all experiments over three different test runs.

We stress each component separately to learn about its impact on the overall performance. There are three main tasks that could slow down the data access:

1. the performance of the database,
2. the transfer of extent information over the network, and
3. the mapping of block addresses via Share.

In the first setting, a clean system is accessed, i.e. the driver has no information locally and contacts the database. The database has to find free extents on the accessed disks and deliver the information to the driver. Such a scenario is the worst possible case for the storage solution.

In a second case, we delete the extent information inside the driver before accessing the data files. That way, the driver contacts the database and the metadata is transfered via the network. The database does not have to derive the extent information but just look it up.

To evaluate the last point, we simply run bonnie without cleaning the driver or database beforehand. That way, the driver accesses only its local information

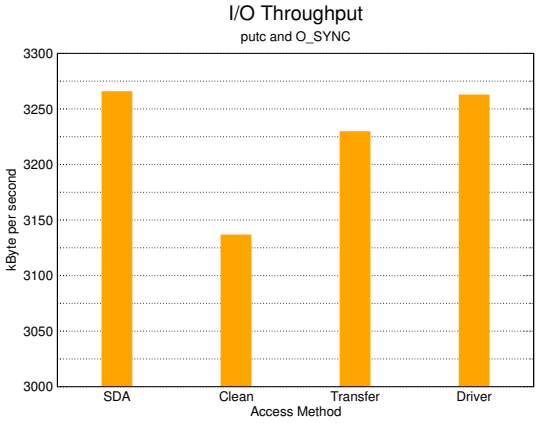

Figure 5.27: This figure shows the throughput of different settings when the files where written with *putc*. The left bar shows the access to a pure partition while the others depict the performance of our solution on a clean system, after cleaning the driver, and with full local information.

and needs no communication at all. The main task it has to solve is the mapping of addresses to extents, i.e. data blocks inside an extent.

All three settings are tested for two different access methods. In one trial, we write the data files with the *putc* routine, i.e. write each character separately. This is slow because it generates many updates of data blocks. The more intelligent way is to access the data block-wise. This is done in a second suite of experiments. Note that this access pattern explores the special access to block devices.

Moreover, we test all configurations for two different file access methods. Recall that the access to data on block devices always passes through the buffer cache. Hence, we have the effect of caching. We use two different methods to reduce that effect. First, the size of the files is at least four times bigger than the main memory. We derived this number experimentally by running bonnie for different file sizes. There is a significant drop (in order of magnitudes) in performance when the files do not fit into main memory. Accessing these files sequentially minimises the likelihood of caching.

Second, we use the switch *O_SYNC* for all data files. This ensures that the data blocks are written to the disks as soon as they have changed. Here, no caching effect can occur. We conducted experiments with activated caching as well as with the disabled caching feature.

The synchronised results for the two different writing methods are shown in Figure 5.27 and Figure 5.28. The left bar depicts the performance without the intervention of our driver. It gives us the maximal possible performance. The next

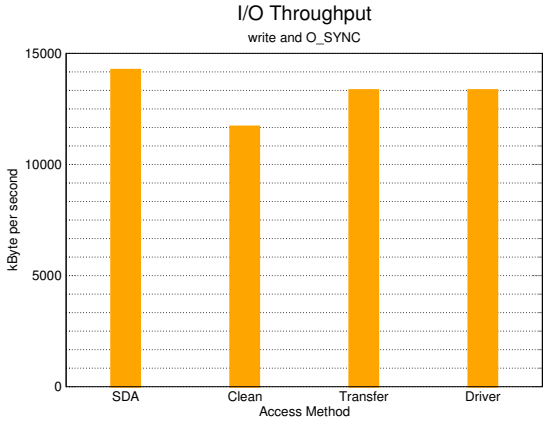

Figure 5.28: This figure shows the throughput of different settings when the files where written with *write*. The left bar shows the access without our driver while the others depict the performance of our solution on a clean system, after cleaning the driver, and with full local information.

three bars show the performance of our solution for the three different settings: starting from a clean system, transfer of extent information, and the overhead induced by the driver alone.

Surprisingly, the overhead in the worst case setting is only about 4% of the maximal performance. This is due to the special access pattern induced by the *putc* routine. First of all, it generates many requests for the same extent. Note that an extent is allocated and transferred only once. Thereafter it is local to the driver and can be accessed very fast. Hence, the overhead of allocating an extent and transferring it to the driver is credited to many data accesses and, therefore, is very minor. Furthermore, the synchronising option of the files (*O_SYNC*) ensures that each data access is written to a disk as soon as possible. The slowness of the disks ensures that the overhead does not have a significant impact. It follows that we cannot distinguish the performance of a plain disk from the performance of our driver having all information locally.

The behaviour is different if we write the data block-wise as can be seen in Figure 5.28. Still, all data accesses are written as soon as possible but we cannot credit the overhead to many different accesses. This results in an overhead of about 18% for the clean system and roughly 6% overhead when the database is not involved. Note that the transfer of data to the driver seems to have no impact. This is due to the fact that we write every single data access. The writes interleave with the extent transfer. Why is that possible? The reason is the buffering of disk accesses. As soon as the write request is handed over to the buffer cache, the

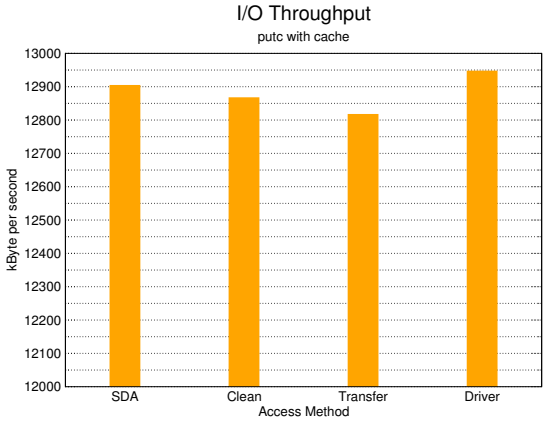

Figure 5.29: The figure shows the results for the sequential write when each character is written separately (*putc*). The first bar gives the throughput of a plain disk. The other bars show the performance for a clean system, after cleaning the driver and with full information.

processor is not involved anymore. Hence, the transfer of extent information and the write to the disks can be done simultaneously.

More interesting than the direct writing of data accesses is the behaviour when the buffer cache functionality is enabled. The throughput for both access methods is shown in Figure 5.29 and Figure 5.30. One notes first of all that the differences for the *putc* writing are marginal. The main reason is the crediting of the overhead to many data requests. Furthermore, the cache smoothes out differences. Consider the two bars in the middle of Figure 5.29. We start the experiments by building a file system on the device. Hence, the data blocks containing the file system information are already in the main memory before the test of the clean system starts. When we check the transfer performance, these blocks have to be loaded from the secondary memory because the main memory is saturated from the test before. This results in a marginal performance loss. Overall, the throughput is almost indistinguishable for the different settings.

The behaviour is substantially different when we access the data block-wise as depicted in Figure 5.30. The figure shows not only the throughput for all the different settings but also for a different number of disks. We test each of the overhead settings – clean system (C), transfer of extent information using the database only for lookup (T), and driver with full information (D) – with a storage pool consisting of one, two, and three disks.

The most apparent observation is that we get only half the throughput when we start from a clean system. Why is this so? There are a number of different

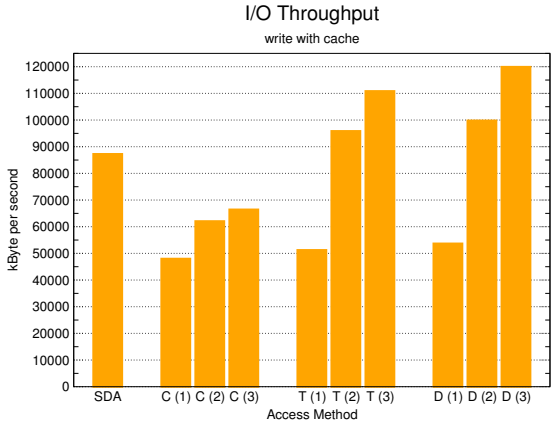

Figure 5.30: This figure illustrates the throughput for block-wise disk accesses in the case that caching is allowed. The first bar shows the performance of a plain disk. The results with our driver are grouped into three categories: starting from a clean system (C), transferring extent information (using the database only for lookup) (T), and having all information local to the driver (D). Each group shows the throughput for a storage pool consisting of one, two or three disks separately.

reasons that influence the drivers' performance. First of all, the access to the files is sequential. This fact combined with the activated caching effects results in a very special access pattern for magnetic disk drives. Note that the most restrictive factor in disk accesses is the search time t_{search} because it involves the mechanical movement of the disk head. Therefore, the maximum throughput can be achieved if the disk head has to move as little as possible (sweeps over the disks to read all the data). Such a behaviour can only be accomplished if the data blocks are placed accordingly by the file system and there are enough data accesses such that they can be issued in that order. For all these reasons, every disk has two characteristics describing its throughput: maximal and sustained performance. The former is the best possible throughput, i.e. when the disk head sweeps over the disk, while the latter describes the average performance, i.e. when the disk head has to move arbitrarily.

So why does that influence the performance of our solution when we access only one disk? The apparent reason is the layout of data blocks. The file system ensures that sweeping is possible. However, we break this layout by using extents. The principle behind the allocation of extents is first fit, i.e. we take the first possible position on the disk where the extent fits in. This destroys the carefully designed data layout of the file system and, hence, makes maximal performance

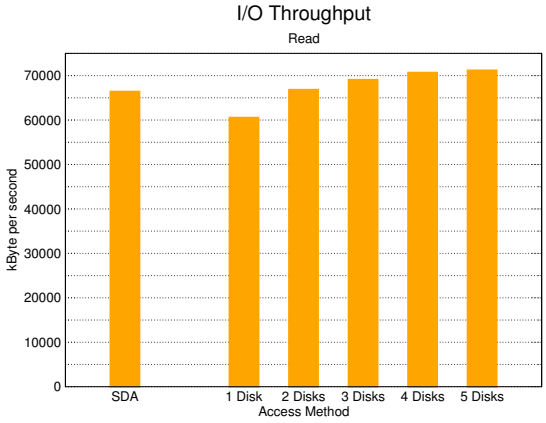

Figure 5.31: The figure shows the read throughput for our solution with a varying number of disks to feed the virtual device. Note that all information to access the data is local to the server.

impossible.

That the overhead induced by the driver it not very high becomes evident from the experiments with more than one disk. If the overhead is the limiting factor, we would not achieve any improvement by using more than one disk. Instead, we increase the throughput significantly. Note that we could not expect many parallel accesses because of the sequential access pattern. It seems that the database is a limiting factor because the performance is not increased for the clean system. Nonetheless, we get an even higher throughput for the other two settings. Note that the difference between a driver having all information locally and one contacting the database is much more pronounced than in all previous experiments – 14% for two disks and even 17% performance increase for three disks. This hints that it matters how fast we can fill the disk queues and that we get parallel accesses even for sequential accesses. Additional experiments showed that more disks do not increase the throughput any further. This is due to the fact that there is too little parallelism to keep more disks busy. The most important result of this experiment is the fact that we are able to increase the performance by roughly 30% for an access pattern that is not parallel at all.

Keeping the above reasoning in mind, we expect the read performance to behave similarly to the write performance. However, looking at Figure 5.31 shows a different picture. The figure depicts the read throughput of our driver with a varying number of disks feeding the virtual device. Note the all information is local to the driver because we had to generate the files before reading them. Hence, we measure only the overhead induced by the driver itself.

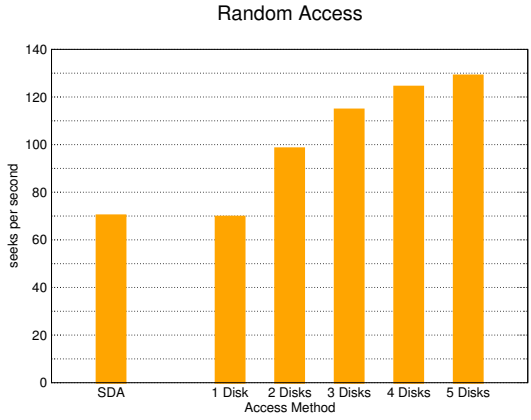

Figure 5.32: The figure depicts the number of seeks per second for a varying number of disks compared to a plain disk (SDA). Note that the data accesses are done locally and do not involve data transfer or the database.

Obviously, the read throughput does not suffer so much from a more scattered data layout – our solution loses only about 9% compared to a plain disk. This can be explained by the fact that a read is blocking while a write returns as soon as the request is issued to the buffer cache. A further indicator for this behaviour is the reduced performance of reads compared to write requests (roughly 66 megabyte/s compared to 87 megabyte/s). We conclude that the operating system has more time to arrange requests. Thus, the performance of our driver is close to the performance of one disk alone. For the same reason, we do not gain much throughput by increasing the number of disk drives. That the inherent parallelism is limited can be seen on the very slight performance increase with a larger number of disks. In fact, we only get parallel accesses at the end of each extent – in less than 2% of all accesses.

Last but not least, we compare the number of seeks per second. In our opinion, this is the most relevant test because it resembles the access pattern of real world applications best. In general, many different applications access a virtual device which results in an unpredictable access pattern. The results of the experiments are shown in Figure 5.32. Here, we try to access as many randomly chosen data blocks as possible. As can be seen in the figure, we get a very high rate of parallel accesses; the number of seeks increases by roughly 40% when we use two instead of one disk. Apparently, the increase in seeks decreases when the number of disks grows. This is due to the fact that the number of simultaneously scheduled requests is constant and, hence, the number of parallel accesses decreases the more disks are participating.

Extent size	1 MB	2 MB	4 MB	8 MB
1 GB	91 KB	46 KB	23 KB	12 KB
4 GB	364 KB	182 KB	91 KB	46 KB
1 TB	91 MB	46 MB	23 MB	12 MB

Figure 5.33: The table lists the space consumption needed in the driver for different extent sizes and varying total storage spaces.

5.3.1 Space Consumption

The last overhead we consider is the memory consumption inside the kernel. Each server, more specifically, each driver, needs information about the virtual devices, the corresponding storage pools and, most importantly, the used extents. Clearly, this information can always be provided by the database, but, as the results have shown, this slows down the data access significantly.

Obviously, the memory is mostly consumed by extent entries. In the current implementation we need 92 bytes per entry. Based on our current driver implementation, Figure 5.33 shows the estimated space consumption for different extent sizes.

Given current main memory sizes our approach stays implementable even for very large storage networks. Furthermore, the extent information can always be requested from the database. This gives rise to a performance versus storage resources trade-off. We simply limit the size of the main memory usable for extent information. Whenever we run out of space, we evict some least recently used extent structures such that we have to request them from the database the next time they are accessed. This leads to a caching behaviour which is similar to the general handling of block devices under Linux.

CHAPTER 6

Conclusions

This thesis aims to introduce dynamic behaviour into storage networks. By dynamic we mean that it is possible to add and remove disks from a storage system without losing the previous data distribution.

We devised data distribution strategies that are not only capable of keeping the data evenly distributed over all participating disks but also move only a small fraction of the data to preserve this even distribution if the number of participating disks changes. We also considered the case of heterogeneity, i.e. disks with different storage capacity. Our strategies could guarantee that every disk gets an amount of data according to its capacity.

In the first part, we introduced the different strategies and analysed them in a theoretical model. As expected, the homogeneous case is much easier to handle than the heterogeneous counterpart. This is due to the fact that we do not need to know anything about the other disks because all disks get the same amount of data. In the heterogeneous case, the amount of data that must be moved to a certain disk depends on the capacity of that disk. We introduced two randomised data distribution strategies – namely Share and Sieve – that are capable of handling heterogeneity, adaptivity, and space balance efficiently.

These new strategies lead to a previously unrealised flexibility in handling storage networks and, thus, a possible decoupling of the logical from the physical view. The central idea is to dissipate the disk space into manageable chunks called extents and apply the distribution strategy to find a good data layout. It has been implemented using the Share strategy for the extent distribution. We built a complete virtualisation solution for Linux and tested its quality concerning distribution and overhead.

The first part of the evaluation concerns the best possible, theoretically achievable distribution quality of the Share strategy. The experiments show that we reach a deviation of about 10% for feasible parameters. Surprisingly, the bulk of the de-

viation is caused by the simpler uniform strategy. We were able to show that the reduction of heterogeneity results in a deviation of less than 1% only. Hence, a better uniform strategy is needed.

Using the virtualisation in a real-world environment has given us an idea of the overhead induced by our virtualisation. Surely, we have to pay for the management of the extents. However, the influence on the overall throughput is by far not as important as expected. Only when using a storage pool with one disk, we roughly halve the performance for block-wise writes. As soon as we increase the number of participating disks, we exceed the throughput despite a sequential access pattern.

The results for random seek operations are ever better. When accessing the data in an unpredictable way, the performance scales with the number of participating disks. This shows that the distribution quality is very good and the induced overhead is acceptable.

Are better strategies possible? This question is difficult to answer. It would be a great achievement if the usage of randomisation can be reduced. Hence, it can be worthwhile to investigate strategies that use randomisation to distribute the data items evenly over a fixed size interval but map that interval with deterministic methods to the disks.

Regarding randomised algorithms, we have to look for a better concentration around the expectation value, i.e. achieve results with high probability.

On the practical side, there is a huge demand for more evaluations. The experiments conducted mark a first step into the direction of practical storage virtualisation. The next stage is to use the lessons learnt and devise better distribution strategies.

CHAPTER 7

Acknowledgments

Thanking everybody to whom I owe a little part of my thesis is impossible. Instead, I humbly invite you all to search for yourself in the following little story.

Once upon a time a little sheep had to look for a new meadow because his old grazing land could not feed it anymore. He bid goodbye to his father and his family and went off to new horizons. It did not take long until he came upon a huge meadow full of tasty green grass. 'Great', thought the little sheep, 'let's give it a try!'. And indeed, the wise ram, whom everybody referred to as The Lord of the Meadow, gave him a warm welcome.

Everything, in the world of a sheep, has to do with grass, or better with the cutting of grass. Here, the sheep were specialised in doing that simultaneously all together. Imagine the surprise of the little sheep when he found out that they all only thought about that! That was a funny thing and rather new to him. So, he went to a kind of nursery school with lots of other sheep from all over the place. That was a very jolly bunch, indeed. Some tried to build a lawn mower, but they had to do in the outskirts of the meadow. Maybe because of their drinking habits. Some wanted to partition the meadow into pieces such that each sheep gets the same amount of grass, and they did that very well because they had red-framed glasses. Some pretended that they cannot see the strip of meadow they are currently eating. And some simply wanted to sell the grass. Anyway, the little sheep liked it there and was grateful for all the diversity.

One day two sheep came to him and wanted his participation in a grass-storing project. All this cut grass must be put somewhere. So, they thought about putting it into large silos of different size. That was fun! Now the sheep could think about grass problems, too.

Life on the meadow was never uninteresting. There were so many things to discover and so many sheep to meet. For instance the small sheep who rocked to and fro and got lost from time to time. He was such a jolly good fellow with his

greasy black hair, and the little sheep is very grateful for that. Often they played ball even when a fence was between them. In the same shed lived another small sheep who always had a smile on her face. When the other sheep felt in teasing mood, they called her Ewe and run away as fast as their little feet could carry them. The little sheep had very ofter tea with her. Some sheep were not completely under the supervision of The Lord of the Meadow. They wanted to keep everybody from knowing that they cut grass at all. So, they simply painted the grass in different colours. To mention all the sheep and activities on the meadow would take days. Noteworthy are all of them. Just take the administrator sheep from down below. The holy trinity never failed to be easy with a joke about operator skills but also never failed to solve problems.

Thinking about big grass silos was sometimes very tiring. So, the sheep played a funny little game. They went into a maze and pretended that they could throw little bombs at each other. Our little sheep was never really good at that even with professional training. Maybe one has to be born for that.

Time went by and the silos took shape. Occasionally, there came other sheep from meadows far away. They bleated in different accents but the little sheep could understand them. Maybe it was because it spent a lot of time on a little pond where everybody was bleating in that accent. If you want to get a decent pint of grass juice you had to bleat like them, didn't you? Our little sheep was always happy to take those other sheep to the little pond and speak to them in their own tongue. How interesting they were. One was very fond of frogs and needed some guidance. The other was turned upside down because she grew up on the other side of the world. The little sheep liked their stories.

So the little sheep grew up – but just a little – and all his thinking ended up in a new grass storing silo complex that is capable of extension. He looks back at his time at the meadow with gratitude, awe and respect. Lest to forget the one sheep that kept him company over all the years. Without her nothing would have been possible.

Kay A. Salzwedel

Bibliography

[AHH+01] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A.R. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An experimental study of data migration algorithms. *Lecture Notes in Computer Science*, 2141:145–158, 2001.

[AT97] J. Alemany and J.S. Thathachar. Random striping news on demand servers. Technical Report TR-97-02-02, University of Washington, Department of Computer Science and Engineering, 1997.

[BBB+01] M. Blaum, J. Brady, J. Bruck, J. Menon, and A. Vardy. The EVENODD code and its generalization: An efficient scheme for tolerating multiple disk failures in RAID architectures. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 14, pages 187–208. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[BBBM94] 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. IEEE Computer Society, 1994.

[BCSV00] A. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. In *ACM Symposium on Theory of Computing*, pages 745–754. ACM Press, 2000.

[BEY98] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.

[BHM+04] A. Brinkmann, M. Heidebuer, F. Meyer auf der Heide, U. Rückert, K. Salzwedel, and M. Vodisek. V:drive - costs and benefits of an out-of-band storage virtualization system. In *Proceedings of the 12th NASA*

Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST), College Park, Maryland, USA, 13 - 16 April 2004.

[Bir01] Y. Birk. Random RAIDs with selective exploitation of redundancy for high performance video servers. *Readings in multimedia computing and networking*, pages 671–681, 2001.

[BMGJ94] S. Berson, R. Muntz, S. Ghandeharizadeh, and X. Ju. Staggered striping in multimedia information systems. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 23(2):79–90, 1994.

[BMS+03] A. Brinkmann, F. Meyer auf der Heide, K. Salzwedel, C. Scheideler, M. Vodisek, and U. Rückert. Storage management as means to cope with exponential information growth. In *Proceedings of SSGRR 2003*, July 2003.

[Bra96] T. Bray. The Bonnie file system benchmark. http://www.textuality.com/bonnie, 1996.

[BRS01] R. Bryant, D. Raddatz, and R. Sunshine. PenguinOMeter: A new file-I/O benchmark for Linux. In USENIX, editor, *Proceedings of the 5th Annual Linux Showcase and Conference, November 5–10, 2001, Oakland, CA*, pages 87–98, Berkeley, CA, USA, 2001. USENIX.

[BSS00] A. Brinkmann, K.A. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th Annual Symposium on Parallel Algorithms and Architectures*, pages 119–128. ACM Press, 2000.

[BSS02] A. Brinkmann, K.A. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform requirements. In *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures*, pages 53–62. ACM Press, 2002.

[CGL99] C. Chou, L. Golubchik, and J.C.S. Lui. A performance study of dynamic replication techniques in continuous media servers. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, volume 27,1 of *SIGMETRICS Performance Evaluation Review*, pages 202–203, New York, May 1–4 1999. ACM Press.

[Che52] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[CL00] T. Cortes and J. Labarta. A case for heterogenenous disk arrays. In *Proceedings of the International Conference on Cluster Computing*, pages 319–325. IEEE Computer Society Press, 2000.

[CL01] T. Cortes and J. Labarta. Extending heterogeneity to RAID level 5. In *Proceedings of the USENIX 2001*, pages 119–132. USENIX Association, 2001.

[CLG+94] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[Cor99] T. Cortes. Software RAID and parallel file systems. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 463–496. Prentice Hall PTR, Upper Saddle River, NJ, 1999. Chap. 19.

[CP90] P.M. Chen and D.A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th Annual Symposium on Computer Architecture, Computer Architecture News*, volume 18(2), pages 322–331. ACM Press, 1990.

[DR01] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Hot Topics in Operating Systems*, pages 75–80, Schloss Elmau, Germany, 2001. IEEE Computer Society Press.

[EJ01] D. Eastlake, 3rd and P. Jones. US secure hash algorithm 1 (SHA1). IETF RFC 3174, 2001.

[Gib91] G.A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, University of California at Berkeley, December 1991.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman, New York, 1979.

[GKK+00] L. Golubchik, S. Khanna, S. Khuller, R. Thurimella, and A. Zhu. Approximation algorithms for data placement on parallel disks. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 223–232, N.Y., January 9–11 2000. ACM Press.

[GS93] S. Ghandeharizadeh and C. Shahabi. Management of physical replicas in parallel multimedia information systems. *LNCS, Foundations of Data Organisation and Algorithms 1993*, 730:51–68, 1993.

[Hal67] M. Hall Jr. *Combinatorial Theory*. Blaisdell Publishing Company, 1967.

[HG92] M. Holland and G.A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 27(9), pages 23–35. ACM Press, 1992.

[HGK+94] L. Hellerstein, G.A. Gibson, R.M. Karp, R.H. Katz, and D.A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2/3):182–208, 1994.

[HHK+01] J. Hall, J. Hartline, A.R. Karlin, J. Saia, and J. Wilkes. On algorithms for efficient data migration. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01)*, pages 620–629, New York, January 7–9 2001. ACM Press.

[HMS99] J.H. Hartman, I. Murdock, and T. Spalink. The Swarm scalable storage system. Technical Report TR99-06, The Department of Computer Science, University of Arizona, 1999.

[JH00] H. Jin and K. Hwang. Optimal striping in RAID architecture. *Concurrency: Practice and Experience*, 12(10):909–916, 2000.

[Joh84] O.G. Johnson. Three-dimensional wave equation computation on vector computers. In *Special issue – Supercomputers - Their Impact on Science and Technology computational physics*, volume 72(1), pages 90–95. IEEE Computer Society Press, 1984.

[KKW03] S. Khuller, Y. Kim, and Y. Wan. Algorithms for data migration with cloning. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 27–36. ACM Press, 2003.

[KLL+97] D.R. Karger, E. Lehman, T. Leighton, M. Levine, D.M. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, 1997. ACM Press.

[KSB+99] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1203–1213, May 1999.

[Lew98] D.M. Lewin. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Department of Electrical Engeneering and Computer Science, Massachusetts Institute of Technology, 1998.

[MH00] I. Murdock and J.H. Hartman. Swarm: A log-structured storage system for linux. In *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference*, pages 1–10. USENIX Association, 2000.

[PGK88] D.A. Patterson, G.A. Gibson, and R.H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data: 1988 Annual Conference*, pages 109–116. ACM Press, 1988.

[RC01] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 2001.

[RD01] A. Rowstron and P. Druschel. Storage management and caching in PAST, A large-scale, persistent peer-to-peer storage utility. In Greg Ganger, editor, *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 188–201. ACM Press, 2001.

[RS98] M. Raab and A. Steger. "Balls into Bins" – A simple and tight analysis. In *RANDOM: International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 1518, pages 159–170. LNCS, 1998.

[RW94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[Sal03] K.A. Salzwedel. Algorithmic approaches for storage networks. In *Algorithms for Memory Hierachies*, pages 251–272. LNCS, 2003.

[San01] P. Sanders. Reconciling simplicity and realism in parallel disk models. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms*, pages 67–76. ACM Press, 2001.

[Sch00a] C. Scheideler. *Probabilistic Methods for Coordination Problems*, volume 78 of *HNI-Verlagsschriftenreihe*. HEINZ NIXDORF INSTITUT, Universität Paderborn, 2000.

[Sch00b] G. Schütte. Evaluierung von Datenplatzierungsverfahren für Multimedia-Server. Diplom, University of Paderborn, 2000.

[SEK00] P. Sanders, S. Egner, and J.H.M. Korst. Fast concurrent access to parallel disks. In *Proceedings of the Eleventh Annual Symposium on Discrete Algorithms*, pages 849–858. ACM Press, 2000.

[SGM86] K. Salem and H. Garcia-Molina. Disk Striping. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 336–342. ACM Press, 1986.

[SM98a] J.R. Santos and R.R. Muntz. Performance analysis of the RIO multimedia storage system with heterogeneous disk configuration. In *Proceedings of the Sixth International Conference on Multimedia*, pages 303–308. ACM Press, 1998.

[SM98b] J.R. Santos and R.R. Muntz. Using heterogeneous disks on a multimedia storage system with random data allocation. Technical Report 980011, University of California, Los Angeles, Computer Science Department, 1998.

[SMRN00] J.R. Santos, R.R. Muntz, and B. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *Proceedings of the international conference on Measurements and modeling of computer systems*, pages 44–55. ACM Press, 2000.

[SS92] D.P. Siewiorek and R.W. Swarz. *Reliable Computer Systems: Design and Evaluation*. Digital Press, Bedford, Massachusetts, USA, 1992.

[SS96] E.J. Schwabe and I.M. Sutherland. Flexible usage of parity storage space in disk arrays. In *8th Annual Symposium on Parallel Algorithms and Architectures*, pages 99–108. ACM Press, 1996.

[SS99] E.J. Schwabe and I.M. Sutherland. Flexible usage of redundancy in disk arrays. *Theory of Computing Systems*, 32(5):561–587, 1999.

[SSH96] E.J. Schwabe, I.M. Sutherland, and B.K. Holmer. Evaluating approximately balanced parity-declustered data layouts for disk arrays. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 41–54. ACM Press, 1996.

[ST01] H. Shachnai and T. Tamir. On two class-constrained versions of the multiple knapsack problem. *ALGRTHMICA: Algorithmica*, 29, 2001.

[ST02] H. Shachnai and T. Tamir. Tight bounds for online class-constrained packing. *Lecture Notes in Computer Science*, 2286:569–583, 2002.

[TPBG93] F.A. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming RAID: A disk array management system for video files. In *Proceedings of the First ACM International Conference on Multimedia*, pages 393–400. ACM Press, 1993.

[vH02] W. von Hagen. *Linux Filesystems*. SAMS Publishing, Indianapolis, IN, USA, 2002.

[VRG95] H.M. Vin, S.S. Rao, and P. Goyal. Optimizing the placement of multimedia objects on disk arrays. In *International Conference on Multimedia Computing and Systems*, pages 158–166. IEEE Computer Society, 1995.

[ZG00] R. Zimmermann and S. Ghandeharizadeh. HERA: Heterogeneous extension of raid. In H.R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 4, pages 2159–2165. CSREA Press, 2000.