Dynamic and Redundant Data Placement (Extended Abstract)

A. Brinkmann, S. Effert, F. Meyer auf der Heide
Heinz Nixdorf Institute
Paderborn, Germany
brinkman@hni.upb.de, {fermat, fmadh}@upb.de

C. Scheideler
TU München
München, Germany
scheideler@in.tum.de

Abstract

We present a randomized block-level storage virtualization for arbitrary heterogeneous storage systems that can distribute data in a fair and redundant way and can adapt this distribution in an efficient way as storage devices enter or leave the system. More precisely, our virtualization strategies can distribute a set of data blocks among a set of storage devices of arbitrary non-uniform capacities so that a storage device representing x% of the capacity in the system will get x% of the data (as long as this is in principle possible) and the different copies of each data block are stored so that no two copies of a data block are located in the same device. Achieving these two properties is not easy, and no virtualization strategy has been presented so far that has been formally shown to satisfy fairness and redundancy while being time- and space-efficient and allowing an efficient adaptation to a changing set of devices.

1 Introduction

The ever-growing creation of and demand for massive amounts of data requires highly scalable data management solutions. The most flexible approach is to use a pool of storage devices that can be expanded as needed by adding new storage devices or replacing older devices by newer, more powerful ones. Probably the most elegant way of hiding the management of such a system from the users is to use storage virtualization, i.e. to organize the storage devices into what appears to be a single storage device. This will help to hide the complexity of the storage system from the users, but coming up with a highly effective and flexible storage virtualization is not easy.

Storage virtualization solutions vary considerably, especially because there is no generally accepted definition of how and at which level storage virtualization is supposed to provide an interface to the users. We will address the problem of block-level storage virtualization. That is, we assume that all data in the system is organized into blocks of uniform size that have to be assigned to storage devices in the given pool. One approach to keep track of this assignment as the system evolves is to use rule-based or table-based placement strategies. However, table-based methods are not scalable and rule-based methods can run into fragmentation problems, so a defragmentation has to be performed once in a while to preserve scalability.

In this paper, we will follow a different approach, which is based on hashing. The basic idea behind hashing is to use a compact function h in order to map balls with unique identifiers out of some large universe U into a set of bins called S so that the balls are evenly distributed among the bins. In our case, the balls are the data blocks and the bins are the storage devices. Given a static set of devices, it is easy to construct a hash function so that every device gets a fair share of the data load. However, standard hashing techniques cannot adapt well to a changing set of devices.

Fortunately, there are hashing schemes that can adapt to a changing set of devices without creating a large overhead. The most popular is probably the consistent hashing scheme by Karger et al. [8], which is able to evenly distribute a single copy for each data block about storage devices and to adapt to a changing number of disks. However, in real systems, storing just a single copy of a data block can be dangerous because if a storage device fails, all of the blocks stored in it cannot be recovered any more. A simple alternative is to store multiple copies of each data block, but as will be demonstrated in this paper, it is not an easy task to distribute these copies among the storage devices so that in the following every storage device has a fair share of the data load (meaning that every storage device with x% of the available capacity gets x% of the data and the requests) and that the redundancy property (meaning that the copies of each data block are distributed so that no two copies are stored on the same device) are kept.

In fact, all hashing strategies proposed so far for non-uniform storage systems can only satisfy the two conditions in special cases. In this paper, we first derive a formula for the maximum number of data blocks a system of arbitrary non-uniform storage devices can store if the fairness and the redundancy conditions above have to be satisfied. Then we illustrate why it is not easy to satisfy fairness and redundancy at the same time, and afterwards we propose the first hashing strategies that can satisfy both conditions for arbitrary non-uniform storage systems.

1.1 The Model

The model for our investigations is based on an extension of the standard balls into bins model (see e.g. [7, 9]). Let $\{1, \ldots, M\}$ be the set of all identifiers for the balls and $\{1, \ldots, N\}$ be the set of all identifiers for the bins. Suppose that the current number of balls in the system is $m \leq M$ and that the current number of bins in the system is $n \leq N$. We will often assume for simplicity that the balls and bins are numbered in a consecutive way starting with 0 (but any numbering that gives unique numbers to each ball and bin would work for our strategies).

Suppose that bin i can store up to b_i (copies of) balls. Then we define its relative capacity as $c_i = b_i / \sum_{j=1}^{n} b_j$. We require that for every ball, k copies have to be stored in the system for some fixed k. In this case, a trivial upper bound for the number of balls the system can store while preserving fairness and redundancy is $\sum_{j=1}^{n} b_j / k$, but it can be much less than that in certain cases, as we will see. The k copies of a ball will be called a *redundancy group*. Even though we just focus on plain mirroring, all results inside this paper are also valid for other redundancy techniques. Placement schemes for storing redundant information can be compared based on the following criteria (see also [2]):

- **Capacity Efficiency:** A scheme is called *capacity efficient* if it allows us to store a near-maximum amount of data blocks. A replication scheme is called ϵ-competitive if it is able to store at least $(1 - \epsilon)$-times the amount of data that could be stored by an optimal strategy. Capacity efficiency can be shown in the expected case or with high probability. We will see in the following that the fairness property is closely related to capacity efficiency.

- **Time Efficiency:** A scheme is called *time efficient* if it allows a fast computation of the position of any copy of a data block.

- **Compactness:** We call a scheme *compact* if the amount of information the scheme requires to compute the position of any copy of a data block is small (in particular, it should only depend on N and m in a logarithmic way).

- **Adaptivity:** We call a scheme *adaptive* if it allows us to redistribute a near-minimum amount of copies in the case that there is any change in the set of data blocks, storage devices, or their capacities. A placement strategy will be called c-competitive concerning operation ω if it induces the (re-)placement of (an expected number of) at most c times the number of copies an optimal strategy would need for ω.

Our aim is to find strategies that perform well under all of these criteria.

1.2 Previous Results

Data reliability and the support for scalability as well as the dynamic addition and removal of storage systems is one of the most important issues in designing storage environments. Nevertheless, up to now only a limited number of strategies has been published for which it has formally been shown that they can perform well under these requirements.

Data reliability is achieved by using RAID encoding schemes, which divide data blocks into specially encoded sub-blocks that are placed on different disks to make sure that a certain number of disk failures can be tolerated without losing any information [10]. RAID encoding schemes are normally implemented by striping data blocks according to a pre-calculated pattern across all the available storage devices, achieving a nearly optimal performance in small environments. Even though extensions for the support of heterogeneous disks have been developed [4], adapting the placement to a changing number of disks is cumbersome under RAID as all of the data may have to be reorganized.

In the following, we just focus on data placement strategies which are able to cope with dynamic changes of the capacities or the set of storage devices (or bins) in the system. Good strategies are well known for uniform capacities without replication, that is, all available bins have the same capacities and only one copy of each ball needs to be stored. In this case, it only remains to cope with situations in which new bins enter or old bins leave the system. Karger et al. [8] present an adaptive hashing strategy that satisfies fairness and is 1-competitive w.r.t. adaptivity. In addition, the computation of the position of a ball takes only an expected number of $O(1)$ steps. However, their data structures need at least $n \log^2 n$ bits to ensure that with high probability the distribution of the balls does not deviate by more than a constant factor from the desired distribution.

Adaptive data placement schemes that are able to cope with arbitrary heterogeneous capacities have been introduced in [2]. The presented strategies Share and Sieve are compact, fair, and (amortized) $(1 + \epsilon)$-competitive for arbitrary changes from one capacity distribution to another, where $\epsilon > 0$ can be made arbitrarily small. Data placement schemes for heterogeneous capacities that are based on geometrical constructions are proposed in [11]. Their linear method combines the standard consistent hashing approach in [8] with a linear weighted distance measure. A second method, called *logarithmic method*, uses a logarithmic distance measure between the bins and the data.

First methods with dedicated support for replication are described in [5][6]. The proposed *Rush (Replication Under Scalable Hashing)*-strategy maps replicated objects to a scalable collection of storage servers according to user-specified server weighting. When the number of servers changes, the Rush variants try to redistribute as few objects as possible and all variants guarantee that no two replicas of an object are ever placed on the same server. The drawback of the Rush-variants is that they require that new capacity is added in chunks, where each chunk is based on servers of the same type and the number of disks inside a chunk has to be sufficient to store a complete redundancy group without violating fairness and redundancy. The use of sub-clusters is required to overcome the problem if more than a single block of a redundancy group is accidently mapped to a the same hash-value. This property leads to restrictions for bigger numbers of sub-blocks. Extensions of the Rush-family are presented in [12].

1.3 Contributions of this Paper

In this paper, we present the first data placement strategies which efficiently support replication for an arbitrary set of heterogeneous storage systems. We will show that for any number of storage devices, the strategies are always able to be capacity efficient, time efficient and compact and that they are able to support adaptivity with a low competitive ratio in comparison to a best possible strategy.

In order to design capacity efficient strategies for arbitrary heterogeneous systems, several insights are necessary. In Section 2, we start with a precise characterization of the conditions the capacities of the storage devices have to satisfy so that redundancy and fairness can be achieved at the same time without wasting any capacity in the system. With this insight, we derive a recursive formula for the maximum number of data blocks that we can store in a given system without violating the redundancy condition. This formula will be important to calculate the capacity each storage device can contribute to the redundant storage.

After showing that trivial placement strategies cannot preserve fairness and redundancy at the same time, we present in section 3 a placement strategy for two copies and then generalize this to a placement strategy for an arbitrary fixed number k of copies for each data block, which are able to run in $O(k)$. The strategies have a competitiveness of k^2 for the number of replacements in case of a change of the infrastructure. Hence, as long as k is fixed (which should normally be the case), we expect our placement strategies to perform well in practice. In fact, the experiments performed so far and reported in this paper support this claim.

2 Limitations of existing Strategies

In this chapter we will present limitations of existing data placement strategies. First, in Section 2.1, we derive a tight bound for the number of data items (balls) that can be stored in a system of n bins with given capacities, if, for each item, k copies have to be stored in different bins. In Section 2.2, we will demonstrate that the standard replication approach to produce the copies by using k data distributions independently (the trivial approach) is not able to fully exploit the available storage capacity.

2.1 Capacity Efficiency

We consider storage systems consisting of n bins. Let b_i be the number of balls that can be stored in a bin i, $B = \sum_{i=0}^{n-1} b_i$ the total number of copies that can be stored in all bins and $B_j = \sum_{i=j}^{n-1} b_i$. Assume that k divides B. Such a storage system is said to allow a *capacity efficient k-replication scheme*, if it is possible to distribute k copies of each of $m = B/k$ balls over all bins, so that the $i'th$ bin gets b_i copies, and no more than a single copy of a ball is allowed to be stored in one bin. In the following lemma we will give an necessary and sufficient condition for a storage system to allow a capacity efficient k-replication scheme.

Lemma 2.1. *Given an environment consisting out of n bins. Each bin can store b_i balls, where we assume w.l.o.g. that $\forall i \in \{0, \ldots, n-1\} : b_i \geq b_{i+1}$. A perfectly fair data replication strategy is able to optimally use the capacity of all bins if and only if $b_0 \leq B/k$.*

Sketch. We will assume w.l.o.g. inside this proof that $B = k \cdot m$, where $m \in \mathbb{N}$ is an arbitrary integer value. If m is not an integer value, it would be necessary to split copies of balls to be able to optimally use the available storage capacity.

We will show in a first step that it is not possible to optimally use the available capacity if $k \cdot b_0 > B$. To store $b_0 > B/k$ copies in bin 0 without violating the requirements of a valid k-replication scheme, additional $(k-1) \cdot b_0$ copies have to be stored in the bins $\{1, \ldots, (n-1)\}$. This is not possible, because from $B_1 < B - b_0$ and $b_0 > m$ it follows that $\sum_{i=1}^{n-1} b_i < (k-1) \cdot b_0$.

In the second part of the proof we will show that it is possible to store k-copies of m balls, if $k \cdot b_0 \leq B$ by constructing an optimal distribution of the balls. We will place in each step of the construction the copies for one ball. The remaining capacity of each bin i before each step j is denoted as b_i^j and $b_i^0 = b_i$. Furthermore, $B^j = \sum_{i=0}^{n-1} b_i^j$. Each step starts by taking the k bins with biggest b_i^j and places 1 copy of the ball on each of the k bins. Afterwards, the construction decrements their remaining capacities by 1 for the next step. It is shown in the full version of this paper that from

$\forall i \in \{0, \ldots, (n-1)\} : b^j < k \cdot B^j$ it immediately follows that $\forall i \in \{0, \ldots, (n-1)\} : b^{j+1} < k \cdot B^{j+1}$ and that the construction successfully terminates after m steps. $\square$

We have shown in Lemma 2.1 that it is possible to optimally use the available storage capacity for a k-replication scheme, if $\forall i \in \{0, \ldots, n-1\} : b_i \leq k \cdot B$. Nevertheless, it is not always possible to influence the construction of the storage environment. Therefore, it is interesting to know how many data can be stored in a storage environment, even if the environment does not fulfill the pre-conditions given in the previous Lemma.

Lemma 2.2. *Given an environment consisting out of n bins. Each bin can store b_i balls, where we assume w.l.o.g. that $\forall i \in \{0, \ldots, n-1\} : b_i \geq b_{i+1}$. The maximum number of balls that can be stored in the system using a k-replication scheme with $k \leq i$ is*

$$B_{max} = \frac{\sum_{i=0}^{n-1} b'_i}{k}, \text{ where } \begin{cases} b'_0 = \min(b_0, 1/k \cdot \sum_{j=0}^{n-1} b'_j) \\ b'_i = \min(b_i, b'_0) \quad \forall i > 0 \end{cases}$$

Sketch. In a first step, we will show that it is not possible to store more than k copies of B_{max} balls in the environment without violating the constraint that no more than a single copy of a ball is allowed to be stored in one bin. This part of the proof will be performed by contradiction:

If it would be possible to store copies of more than B_{max} balls in the environment, it would follow that at least one bin would have to store more than b'_i balls. The number of balls stored in a bin i can only be bigger than b'_i if $b'_{i-1} < b_{i-1}$, otherwise $b'_i = b_i$ and the capacity constraints of that bin i would be exceeded by storing more than b_i copies in that bin. According to the definition of b'_i, $b'_i < b_i$ is only possible if also $b_0 \neq b'_0$. Lets now assume that $b_0 > 1/k \cdot \sum_{j=0}^{n-1} b_j$. Based on Lemma 2.1, it is not possible in this case to optimally use the available storage capacity for a k-distribution of copies.

The second part of the proof directly follows from Lemma 2.1. Based on the definition of b'_0, all adjusted capacities b'_i follow the requirement that enables an optimal capacity utilization. $\square$

In the remainder of this paper we will assume that the input to the different replication strategies is set in a way that $b_i = b'_i$ for every $i \in \{0, \ldots, n-1\}$ and that it is therefore possible to optimally use the available capacity of the bins. The coefficients b'_i can be calculated in time $O(k \cdot n)$ based on Algorithm 1.

Algorithm 1 optimalWeights(k, $\{b_0, \ldots, b_{n-1}\}$)

```
1: if $b_0 > \frac{1}{k-1} \sum_{i=1}^{n-1} b_i$ then
2:     optimalWeights $((k-1), \{b_1, \ldots, b_{n-1}\})$
3:     $b_0 = \lfloor \frac{1}{k-1} \cdot \sum_{i=1}^{n-1} b_i \rfloor$
4: end if
```

2.2 The trivial data replication approach

Data replication can be performed by different strategies. The most well known strategies are RAID schemes, which evenly distribute data according to a fixed pattern over homogeneous bins. Another important strategy, which is often used in Peer-to-Peer networks, is to use a distributed hashing functions, like Consistent Hashing or Share, and to make k independent draws to get k different bins. We will show in the following that this strategy leads to the loss of the properties of fairness and capacity efficiency.

Definition 2.3. *A replication strategy is called trivial, if it performs k draws to achieve k-replication with $k \geq 2$ according to a fair data distribution scheme for $k = 1$, where the probability that a bin is the target for the i-th draw only depends on its constant relative weight compared to the other bins participating in the i-th draw and not on k. Furthermore, for a trivial replication strategy, exactly all bins that have not been the target of one of the draws $\{1, ..., i-1\}$ take part in draw i.*

We will show in this section that the trivial approach to data replication is not able to fulfill the requirement of fairness for any $k \geq 2$. Before formalizing this, we will give a simple example that explains the drawback of the trivial approach for data replication. Figure 1 shows a system that consists out of 3 bins, where the first bin has got twice the capacity of bin 2 and bin 3. The data distribution scheme should distribute data over the bins for $k = 2$. It is easy to see that it is possible to distribute the data for $k = 2$ perfectly fair by putting each first copy in the first bin and the second copy alternatingly in the second and third bin.

In the case of a trivial data replication strategy, the probability that the larger bin is not the target for the first copy in the setting from Figure 1 is $\overline{p} = 1 - 1/2 = 1/2$. If the first bin is not taken in the first draw, the probability that it also not taken in the second draw is $\overline{p} = 1 - 2/3 = 1/3$ and it follows that the probability that it is not taken at all is $1/6$. To enable an optimal packing of the balls, the larger bin has to be taken in every random experiment, while the trivial strategy wastes $1/6$-th of the capacity of the biggest bin and $1/12$ of the overall capacity.

The following lemma will show that the trivial approach to make k independent draws from the bins leads, especially for a small number of heterogeneous bins, to a significant waste of capacity.

Lemma 2.4. *Assume a trivial replication strategy that has to distribute k copies of m balls over $n > k$ bins. Furthermore, the biggest bin has a capacity c_{max} that is at least*

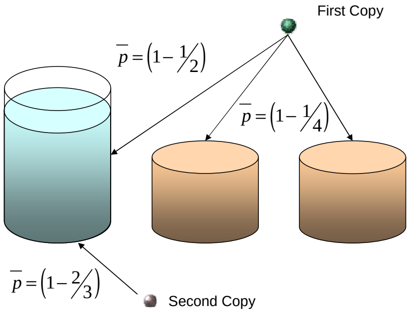
First Copy
$\overline{p} = (1 - 1/2)$
$\overline{p} = (1 - 1/4)$
$\overline{p} = (1 - 2/3)$
Second Copy

Figure 1. Distributing data according to a trivial strategy for $k = 2$.

$(1+\epsilon)\cdot c_j$ *of the next biggest bin* j*. In this case, the expected load of the biggest bin will be smaller than the expected load required for an optimal capacity efficiency.*

Sketch. Inside this proof we assume w.l.o.g. that it is possible to find a valid data distribution for a k-replication that is able to optimally distribute copies over n bins in a way that each bin l gets a share of exactly c_l of all copies. Furthermore, we assume that the bins are ordered according to their capacity and that $c_0 = c_{max} > c_1$. In this case, bin 0 has got to get an expected load of $k \cdot c_0 \cdot m$ balls. Therefore, the probability that no copy of a ball is placed on bin 0 has to be smaller than $\overline{p}_0 = (1 - k \cdot c_0)$ for each random experiment. In the following we will show by complete induction over the number of copies k that the probability $\overline{p}_0$ that no copy of a ball is placed in bin 0 is bigger than the required probability, which proofs Lemma 2.4.

For every strategy that distributes data fair over the bins for $k = 1$, the probability that a bin is not chosen in the first step is $(1 - c_i)$. In the following, γ_j denotes the relative capacity of the bin that has been chosen in draw j. The probability that a bin i is not chosen in the second step is $\left(1 - \frac{c_i}{1-\gamma_1}\right)$. We will start for $k = 2$. In this case, the probability that bin 0 is not chosen has to be bigger than $(1 - 2 \cdot c_0)$.

$$\begin{aligned} \overline{p}_0 = (1-c_0)\left(1 - \frac{c_0}{1-\gamma_1}\right) &> 1 - 2\cdot c_0 \\ \overset{\gamma_1 = \epsilon\cdot c_0}{\Leftrightarrow} \quad (1-c_0)\left(1 - \frac{c_0}{1-\epsilon\cdot c_0}\right) &> 1 - 2\cdot c_0 \qquad (1) \\ \Leftrightarrow \quad c_0 &> \epsilon\cdot c_0, \end{aligned}$$

which is true for each $\epsilon < 1$. The induction step for $(k-1) \rightarrow k$ is shown in the full version of the paper. □

As it can be seen in the proof of Lemma 2.4, it is necessary to ensure that larger bins get their required share of the replicated balls.

3 The Redundant Share Strategy

The analysis of trivial data replication strategies has shown that these strategies suffer from not being able to efficiently use larger bins, especially if the number of bins is small. The Redundant Share strategies presented inside this section overcome the drawback of trivial data replication strategies by favoring larger bins in the selection process. We will start by describing a data replication strategy for a 2-fold mirroring that orders the bins according to their weights c_i and iterates over the bins. Therefore, the strategy needs $O(n)$ rounds for each selection process. After presenting this restricted replication strategy, we will present a generalized k-replication strategy that is able to distribute balls perfectly fair over all bins and that also has linear runtime. Finally, we will present time efficient strategies built upon the restricted strategies, which are also able to work for k-fold mirroring with arbitrary k in $O(k)$.

All strategies presented in this section are not only able to distribute data fair about the bins, but also to keep the property of fairness in case of dynamic insertion or deletion of bins. We will show that the number of necessary replacements can be bounded against an optimal adversary. Furthermore, all strategies are always able to clearly identify the i-th of k copies of a data block. This property is a key requirement, if data is not replicated by a k-mirror strategy, but is distributed according to an erasure code, like Parity RAID, Reed-Solomon Codes or EvenOdd-strategies [1][3]. In case of an erasure code, each sub-block has a different meaning and therefore has to be handled differently.

3.1 Mirroring in linear Time

The input values for the Algorithm 2 LinMirror are the capacities of the bins b_i and the virtual address of the ball that should be distributed over the bins. It is important to notice that we will use k and 2 interchangeable inside this subsection.

The algorithm works in rounds. The task of the while loop is to select the bin for the first copy of the ball. Therefore it iterates over all bins in descending order of their capacities b_i and selects the location for the primary copy based on a random process. An invariant of the algorithm is that it is not possible to place a copy of a ball in a bin i after the i-th round of the while loop. It follows that the algorithm has to ensure that the probability that a ball is placed in bin i is equal to the required share $2 \cdot c_i$ after the i-th round of the while loop.

The calculation of the probability $\check{c}_i$ that bin i is the target for a ball is based on the recursive nature of the algorithm (even if the algorithm is formulated without explicit recursion). It just assumes that secondary copies have been evenly distributed over the bins $i, \ldots, n-1$, so the task of

Algorithm 2 LinMirror ($address$, $\{b_0, \ldots, b_{n-1}\}$)

```
Require: $\forall i \in \{0, \ldots, n-1\} : b_i \geq b_{i+1} \wedge 2 \cdot b_i \leq B$
1: $\forall i \in \{0, \ldots, n-1\} : \check{c}_i = 2 \cdot b_i / \sum_{j=i}^{n-1} b_j$
2: $i \leftarrow 0$
3: while $i < n-1$ do
4:     $rand \leftarrow$ Random value(address,bin i) $\in [0, 1)$
5:     if $rand < \check{c}_i$ then
6:         Set volume $i$ as primary copy
7:         Set secondary copy to
           placeOneCopy($address, \check{c}_i, b_{i+1}, \ldots, b_{n-1}$)
8:         return
9:     end if
10:    $i \leftarrow i + 1$
11: end while
```

Algorithm 3 placeOneCopy($address, \check{c}, \{b_0, \ldots, b_{n-1}\}$)

```
1: $\check{c}_{new} = 2 \cdot b_0 / \sum_{i=0}^{n-1} b_j$
2: if $\check{c} < 1$ and $\check{c}_{new} > 1$ then
3:     $b_0 = b^*$
4: end if
5: Call fair data distribution strategy for one copy with parameters $(adress, b_0, \ldots, b_{n-1})$
```

round i is to find a valid mirroring for the bins $i, \ldots, n-1$.

Each round starts by selecting a random value between 0 and 1, which is calculated based on the address of the data block b and the name of the bin i, which has to be unique inside the environment. If the random value is smaller than the adapted weight $\check{c}_i$ of the bin i in an iteration step of the while loop, the bin is chosen as primary copy. The second copy is chosen by a strategy *placeOneCopy* that has to be able to distribute balls fair over bins without replication. The input of placeOneCopy are all bins which have not yet been examined in the while-loop. Therefore, the second copy of the mirror is distributed over all bins with smaller weight c_j than the weight of the primary copy c_i.

Algorithm 2 contains an inhomogeneity in the usage of placeOneCopy if $\check{c}_i > 1$ for a bin i and $\forall j < i : \check{c}_j < 1$. The inhomogeneity is based on the recursive nature of the algorithm and occurs, if the subset $\{i, \ldots, n-1\}$ of bins does not fulfill the requirements of the algorithm that $2 \cdot b_i < \sum_{j=i}^{n-1} b_j$. In this case it is not sufficient to evenly distribute the secondary copies of preceding bins over the subset, instead it is required to favor bin i. The function placeOneCopy overcomes this problem in round $(i-1)$ by adjusting the probabilities for the data distribution of the secondary copies for primary copies placed in bin $(i-1)$ before calling a fair data distribution strategy for the placement of the secondary copy.

The adjustment $b_0 = b^*$ in Algorithm 3 has to ensure that bin i is able to get the required amount of balls. This is done in the following way. In a first step, the percentage of secondary copies already assigned to bin i up to step $i-2$ is calculated by setting

$$s_i^{i-2} = \sum_{j=0}^{i-2} \left(\check{c}_j \cdot \frac{b_i}{\sum_{l=j+1}^{n-1} b_l} \cdot \prod_{o=0}^{j-1} (1 - \check{c}_o) \right) \tag{2}$$

Equation 2 sums the percentage of the secondary copies assigned to bin i for primary copies stored in the bins $0, \ldots, (i-2)$. $\check{c}_j$ denotes the probability that bin j is chosen as primary copy in round j, $b_i / \sum_{l=j+1}^{n-1} b_l$ the probability that bin i is chosen as secondary copy for that primary copy and $\prod_{o=0}^{j-1}(1 - \check{c}_o)$ the probability that the while loop reaches the j-th round for a ball. In the next step, we will calculate the maximum percentage of primary copies that can be assigned to bin i by setting $\check{c}_i = 1$.

$$p_i = \prod_{j=0}^{i-1} (1 - \check{c}_o) \tag{3}$$

Therefore, the number of secondary copies that have to be assigned to bin i from the primary copies stored in bin $i-1$ is

$$s_i = 2 \cdot c_i - s_i^{i-2} - p_i \tag{4}$$

Based on Equation 4, we are now able to calculate b^* from:

$$s_i = \frac{b^*}{b^* + \sum_{l=i+1}^{n-q} b_l} \cdot \check{c}_{i-1} \cdot \prod_{j=0}^{i-2} (1 - \check{c}_j) \tag{5}$$

To test the fairness of the described algorithm we implemented it in a simulation environment. We started the tests with 8 heterogeneous bins. The first has a capacity of 500,000 blocks, for the other bins the size is increased by 100,000 blocks with each bin, so the last bin has a capacity of 1,200,000 blocks. To show what happens if we replace smaller bins by bigger ones we added two times two bins. The new bins are growing by the same factor as the first did. Then we removed two times the two smallest bins. After each step we measured how much percent of each bin is used. As it can be seen in figure 2, the distribution for heterogeneous bins is fair.

Lemma 3.1. *LinMirror is perfectly fair in the expected case if a perfectly fair distribution scheme inside placeOneCopy is chosen.*

Proof. The expected share of bin i is the sum of the first and second copies of balls stored in that bin. A second copy of a ball can only be assigned to a bin, if the first copy of that ball is stored in a bin j with $j < i$.

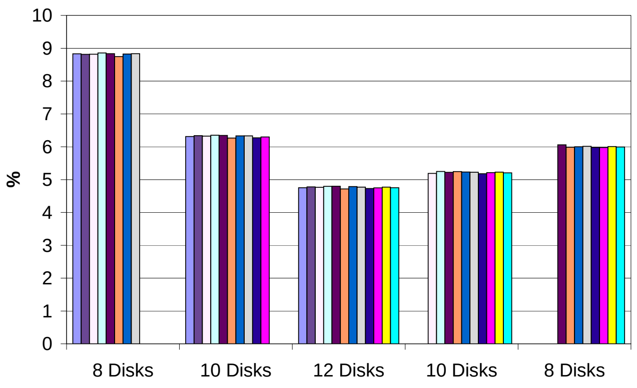

Figure 2. Distribution for heterogenous bins.

The proof is based on the recursive nature of the algorithm. We start the proof for bin 0. Its demand can only be satisfied by primary copies. Therefore, the probability that this bin is chosen as primary copy has to be equal to its demand, which is fulfilled by setting $\check{c}_0 = 2 \cdot c_i$. It is important to notice that placeOneCopy evenly distributes the second copies according to the demands of the remaining bins $\{1, \ldots, (n-1)\}$. Therefore, the problem in the next round reduces to the problem to finde a valid mirroring scheme for the bins $\{1, \ldots, (n-1)\}$, which is achieved by the same mechanisms as for bin 0.

This recursive nature is broken for the case that $\check{c}_i$ becomes bigger than 1. This is the case if the precondition $\forall i \in \{0, \ldots, n-1\} : c_i = b_i / \sum_{j=0}^{n-1} b_j$ of the algorithm can not be fulfilled for bin i. This inhomogeneity is overcome inside placeOneCopy for bin $(i-1)$, where the weight of bin i is adapted so that the demand of the bin is exactly met after round i of the while loop.

Algorithm 2 will terminate after bin i with $\check{c}_i \geq 1$. Based on the fact that all bins in the range $\{0, \ldots, i\}$ exactly get the required demand in the expected case and that the data is evenly distributed over the bins $\{(i+1), \ldots, (n-1)\}$ according to their capacities as well as the precondition of the algorithm and Lemma 2.1, the fairness of the algorithm follows. □

In the next lemma we will show that Algorithm 2 is not only perfectly fair but is also competitive compared to an optimal algorithm in a dynamic environment.

Lemma 3.2. LinMirror *is* 4*-competitive in the expected case concerning the insertion of a new bin* i*.*

Sketch. Assume that a new storage system i with a demand c_i is inserted into the environment. After inserting bin i into the environment, the storage system consists out of $(n+1)$ bins. To keep the property of fairness, part of the copies have to be moved from their current locations. In an optimal case, a strategy has to redistribute at least a share of $\xi = c_i / \sum_{i=0}^{n} c_i$ of all balls to keep this property. In the case of LinMirror, all data blocks for which the strategy produces a new target disk have to be moved from their current location to the new disk.

Lets assume that the new storage system has the biggest share and we will set $i = 0$ and the index of all other bins will be increased by one. W.l.o.g. we will assume that the index of the other bins has already bin increased by one before the insertion of the new bin 0. In this case, an expected share of $\xi = c_0 / \sum_{i=0}^{n} c_i$ of all balls will be placed as primary copy on bin 0 and an additional share of ξ of all balls will become the second copy of a first copy that is placed on bin 0, leading to an expected number of $2 \cdot \xi$ block movements, which directly involve the new disk. The random value *rand* for step i of the while loop only depends on the block number and the name of the bin i, which does not change by the insertion of a new disk. Therefore, additional changes of the placement of first copies can only occur, if $\check{c}_i$ changes after the insertion of bin 0. Based on Algorithm 2, $\check{c}_i$ does not change after the insertion of bin 0 for all $i > 0$. Therefore, no additional balls will be replaced after the insertion of bin 0 and LinMirror is 2-competitive in the expected case, if a new bin with the biggest capacity is inserted.

What happens if the new disk has not the biggest capacity? If a bin i is inserted that has not the biggest share, it has only an influence on the probabilities $\check{c}_j$ for $j < i$. The expected number of primary copies that have to be redistributed for bin j is bounded by $\left(1 - \frac{(\sum_{l=j}^{n-1} b_l) - b_i}{\sum_{l=j}^{n-1} b_l}\right)$-times the number of primary copies stored in bin j. This term can be simplified to $b_i / \sum_{l=j}^{n-1} b_l$.

It has to be shown in a first step, that the expected total number of first copies to be moved from bin j is always at least as big as the expected number of copies that have to be moved from bin $j+1$ weighted by the size of bin j and bin $j+1$. We will bound the number of primary copies inside bin j and $j+1$ by the probability that the while loop of LinMirror reaches step j, resp. step $j+1$. and the probability that bin j, resp. $j+1$ is taken if the algorithm reaches the corresponding step and the number of balls m. This can be done by showing that

$$\frac{m \cdot b_i \cdot \check{c}_j}{\sum_{l=j}^{n-1} b_l} \cdot \prod_{k=0}^{j-1} (1 - \check{c}_k) \geq \frac{b_j}{b_{j+1}} \cdot \frac{m \cdot b_{j+1} \cdot \check{c}_{j+1}}{\sum_{l=j+1}^{n-1} b_l} \cdot \prod_{k=0}^{j} (1 - \check{c}_k)$$

It is now possible to bound the expected number of first copies Γ_{first} to be moved after the insertion of a new bin i from the bins 0 to $(i-1)$ by:

$$\Gamma_{first} \leq \sum_{j=0}^{i-1} \frac{b_j}{b_0} \cdot \frac{b_i}{\sum_{l=0}^{n-1} b_l} \cdot b_0 \leq \frac{b_i}{\sum_{l=0}^{n-1} b_l} \cdot \sum_{j=0}^{i-1} b_j \leq b_i$$

Assuming that the movement of Γ primary copies also triggers the movement of Γ secondary copies, the total number of data movements induced by the movement of a primary copy can be bounded by $2 \cdot b_i$.

Besides the replacements induced by the movement of first copies, it also happens that second copies are moved according to the insertion of bin i. For all primary copies stored in bin j with $j < i$ it holds that the number of replacements for the corresponding secondary copies is bounded by $b_i / \sum_{l=j+1}^{n-1} b_l$-times the number of first copies stored in bin j. Notice that this term is very similar to the term used for the replacement of primary copies, but in this case the denominator only starts by $(j+1)$ and not by j. Based on this observation, it is again possible to show that the number of secondary copies that have to be replaced for a bin constantly decreases with the size of the bins. This can be done by showing that

$$\frac{m \cdot b_i \cdot \check{c}_j}{\sum_{l=j+1}^{n-1} b_l} \cdot \prod_{k=0}^{j-1}(1-\check{c}_k) \geq \frac{b_j}{b_{j+1}} \cdot \frac{m \cdot b_i \cdot \check{c}_{j+1}}{\sum_{l=j+2}^{n-1} b_l} \cdot \prod_{k=0}^{j}(1-\check{c}_k)$$

followed by

$$\Gamma_{second} \leq \sum_{j=0}^{i-1} \frac{b_j}{b_0} \cdot \frac{b_i \cdot b_0}{\sum_{l=1}^{n-1} b_l} \leq \frac{b_i}{\sum_{l=1}^{n-1} b_l} \cdot \sum_{j=0}^{i-1} b_j \leq 2 \cdot b_i$$

It is interesting to observe that it is not negligible, where a disk is inserted. Assume for example that all disks have got the same size. Then the number of replaced first copies is much bigger, if the disk is added as last disk than it would be, if the disk would be added as first disk. This can also be seen in the following experimental evaluations. □

Corollary 3.3. *The algorithm LinMirror is* 4*-competitive in the expected case concerning the deletion of a bin* i.

Sketch. Bin i is removed from the environment. The proof for the competitiveness is based again on the change of the probability $\check{c}_j$ for all bins j with $j < i$. This probability changes by the factor $\frac{\check{c}_{j_{new}}}{\check{c}_{j_{old}}} = \frac{\sum_{l=j}^{n-1} b_j}{\sum_{l=j}^{n-1} b_j - b_i}$. Again, it can be shown that this probability change, weighted with the probability that the algorithm proceeds up to step j, decreases with increasing j and has a maximum for $j = 0$. □

We will now have a closer look at the competitiveness of the adaptivity with the help of the simulation environment. Looking at the proofs for Lemma 3.2 it seems to make a difference at which position a new bin is added or removed. If changes take place at the end of the list (where the smallest bins are placed), $\check{c}_i$ changes for every bigger bin and LinMirror is 4-competitive. Therefore we do not only have to replace blocks placed for the new / deleted bin but also for other bins. If a bin is added at the beginning of the list, LinMirror is 2-competitive.

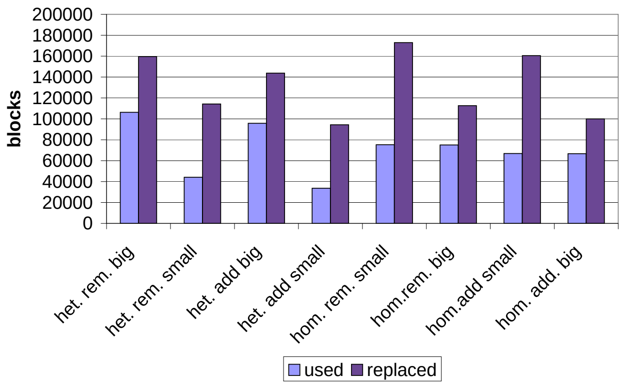

Figure 3. Adaptivity of LinMirror.

To reproduce this effect we checked different scenarios. We made four tests, removing and adding a bin at the begin and the end of the list for heterogeneous and homogeneous environments. Figure 3(a) shows the blocks placed on the affected bin and the number of replaced blocks for every test. It can be easily seen that it makes a big difference at which end the changes take place. For changing the biggest bin we replaced about 1.5 times of the blocks affected by the disk, while changing the smallest bin gives us a factor of about 2.5.

To analyze this, we tested how this factor behaves for a different number of bins. Therefor we added a bin to 4 up to 60 bins and measured the factor of replaced blocks divided by the block used on the newest disk. We performed these tests for adding the new bin as the smallest and as the biggest one. Again, we get nearly constant competitive ratios of about 1.5 for adding the biggest disk and 2.5 for adding the smallest disk.

3.2 k-fold Replication in linear time

In the previous section, we have presented the algorithm LinMirror that is able to distribute mirrored data perfectly fair among a set of heterogeneous bins. Furthermore, we have shown that the adaptivity of the algorithm to changes of the environment can be bounded against a best possible algorithm. In this section, we will generalize the approach of LinMirror to k-fold replication of data.

The algorithm *k-Replication()* is based on Algorithm 2 from the previous section. There is one significant change in the algorithm. If a bin is chosen as primary copy for a ball and $k > 2$, then the following copies are chosen by a recursive decent into *k-Replication()* with k decreased by one[1]. Again, it is necessary to introduce b^*, which can be

[1]The algorithm can also be used for $k == 1$ without loosing its prop-

Algorithm 4 k-Replication (k, $address$, $\{b_0, \dots, b_{n-1}\}$)

```
Require: $\forall i \in \{0, \dots, n-1\} : b_i \geq b_{i+1} \wedge k \cdot b_0 < B$
1:  $i \leftarrow 0$
2:  $\forall i \in \{0, \dots, n-1\} : \check{c}_i \leftarrow k \cdot c_i / \sum_{j=i}^{n-1} c_k$
3:  while $i \leq n$ do
4:    $rand \leftarrow$ Random value($address$) $\in [0, 1)$
5:    if $rand < \check{c}_i$ then
6:      Set volume $i$ as primary copy
7:      if $k == 2$ then
8:        Set last copy to placeOneCopy($b_{i+1}, \dots, b_{n-1}$)
9:        return
10:     else
11:       if $\check{c}_i < 1$ and $\check{c}_{i+1} > 1$ then
12:         $b_{i+1} \leftarrow b^*$
13:       end if
14:       k-Replication($(k-1)$, $address$,
          $\{b_{i+1}, \dots, b_{n-1}\}$)
15:       return
16:     end if
17:   end if
18:   $i \leftarrow i + 1$
19: end while
```

calculated similar to b^* for $k = 2$ and which handles the case that $\check{c}_i$ can become bigger than 1 for the last bin that can get the first copy of a k-replication scheme.

Lemma 3.4. *k-Replication is perfectly fair in the expected case if a perfectly fair distribution scheme placeOneCopy is chosen.*

Proof. The proof of the fairness of *k-Replication* is based on complete induction. The start of the proof is given, based on Lemma 3.1, for $k = 2$. We now assume that the strategy is perfectly fair for a replication degree of m and will show that it is then also perfectly fair for a replication factor $m+1$. $\check{c}_i$ ensures that the share for bin i is perfectly fair for $m+1$ after round i of the while loop, if *k-Replication(m)* is also perfectly fair, given by the constraint of the induction. □

Lemma 3.5. k-Replication *is k^2-competitive in the expected case concerning the insertion or deletion of a new bin i.*

Sketch. The proof of Lemma 3.2 has shown that we have not to move more than b_i first copies. In the worst case we have to move all following copies of these first copies, leading to $k \cdot b_i$ movements. Additional to Lemma 3.2, we have also to consider second, third, ... copies which are moved independent of the primary, secondary,... copy according to the insertion of bin i. This can be done via a recursive decent into the algorithm. For each additional copy, at most

erties of fairness and adaptivity.

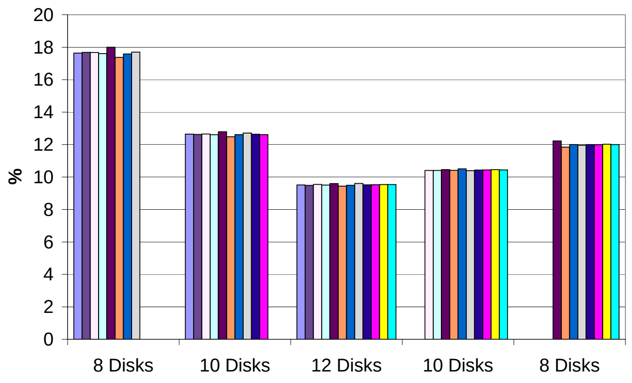

Figure 4. Distribution for heterogeneous bins and $k = 4$.

$k \cdot b_i$ additional data blocks have to be moved, leading to a competitiveness of k^2. □

To test the practical behavior we simulated the behavior for the k-replication. Therefore we performed the same tests for competitiveness and adaptivity as for LinMirror, in this case for $k = 4$ (see Figure 4). As can be seen, all tests resulted in completely fair distributions.

The adaptivity behaves more complex. We have already shown that it matters where in the list of bins a change happens. In Figure 5, we evaluate how the factor of replaced blocks to the blocks on the affected bin behaves for different numbers of homogeneous bins. For adding bins at the beginning of the list, we get nearly a constant factor. For adding it as smallest bin we get more interesting results. The more disks are inside the environment, the worse the competitiveness becomes. This is a result of the influence of the smallest disk to all other disks. In Lemma 3.5, we have shown that the competitiveness converges against an upper bound of k^2. Setting $k = 4$, we get an upper bound of 16. The graph in Figure 5 lets us assume that there is a much lower bound at least for this example.

3.3 k-fold Replication in $O(k)$

Using more memory and additional hash functions for data distribution on heterogenous disks without redundancy, we can improve the runtime of Redundant Share to $O(k)$. For the first copy we only need a single hash function, which computes for each disk the probability that it is used as first copy. We can compute the probabilities by $p_i = \check{c}_i \cdot \prod_{j=0}^{i-1}(1 - \check{c}_j)$, with $\check{c}_i$ based on algorithm 3.2. The sum of the probabilities has to be scaled to become 1. Then we use an algorithm for the placement of a single copy with the scaled probabilities as input. For every following copy we need $O(n)$ hash functions, one for each disk that could be chosen as primary disk in the previous step. For

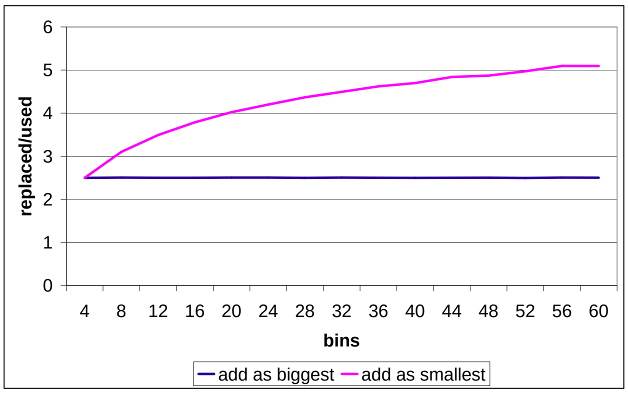

Figure 5. Adaptivity of k-Replication for $k = 4$ and homogeneous bins.

each disk l that could be used as primary copy in the previous step we take the subset $\{l+1, \cdots, n-1\}$ of the disks and compute a hash function that can derive the next copy.

To compute the group of target bins we now start with the hash function for the first disk to get the first copy. Depending on the chosen disk in the step before we choose the next hash function to get the next copy. The hash functions can be chosen in $O(1)$ and there are hash functions with runtime $O(1)$. Therefore, we get to a runtime of $O(k)$. The fairness and the adaptivity are granted by the hash function. The memory complexity is $O(k \cdot n \cdot s)$ where s is the memory required for each hash function.

4 Conclusion and Acknowledgements

In this paper we presented the first data placement strategies which efficiently support replication for an arbitrary set of heterogeneous storage systems. These strategies still leave room for several improvements. For example, can the time efficiency be significantly reduced with less memory overhead? We also believe that it should be possible to construct placement strategies that are $O(k)$-competitive for arbitrary insertions and removals of storage devices. Is this true and is this the best bound one can achieve?

We would like to thank Dr. Fritz Schinkel from Fujitsu Siemens Computers for very helpful comments.

References

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

[2] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform distribution requirements. In *Proc. of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, Winnipeg, Manitoba, Canada, 2002.

[3] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, 2004.

[4] T. Cortes and J. Labarta. Extending heterogeneity to RAID level 5. In *USENIX 2001*, Boston, 2001.

[5] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In *Proc. of the 17th International Parallel & Distributed Processing Symposium (IPDPS 2003)*, 2003.

[6] R. J. Honicky and E. L. Miller. Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, 2004.

[7] N. L. Johnson and S. Kotz. *Urn Models and Their Applications*. John Wiley and Sons, New York, 1977.

[8] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the 29th ACM Symposium on Theory of Computing (STOC)*, 1997.

[9] M. Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California at Berkeley, 1996.

[10] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, 1988.

[11] C. Schindelhauer and G. Schomaker. Weighted distributed hash tables. In *Proc. of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2005)*, July 2005.

[12] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, Scalable And Decentralized Placement Of Replicated Data. In *Proc of SC2006*, 2006.