International Journal on Digital Libraries (2005)
DOI 10.1007/s00799-005-0118-z

REGULAR PAPER

Shu-Yuen D. Yao · Cyrus Shahabi · Roger Zimmermann

BroadScale: Efficient scaling of heterogeneous storage systems

Published online: 25 October 2005

Abstract Scalable storage architectures enable digital libraries and archives for the addition or removal of storage devices to increase storage capacity and bandwidth or retire older devices. Past work in this area have mainly focused on statically scaling homogeneous storage devices. However, heterogeneous devices are quickly being adopted for storage scaling since they are usually faster, larger, more widely available, and more cost-effective. We propose BroadScale, an algorithm based on Random Disk Labeling, to dynamically scale heterogeneous storage systems by distributing data objects according to their device weights. Assuming a random placement of objects across a group of heterogeneous storage devices, our optimization objectives when scaling are to ensure a uniform distribution of objects, redistribute a minimum number of objects, and maintain fast data access with low computational complexity. We show through experimentation that BroadScale achieves these requirements when scaling heterogeneous storage.

Keywords Scalable storage systems · Random data placement · Load balancing · Heterogeneous disk scaling

1 Introduction

Computer applications typically require ever-increasing storage capacity to meet the demands of their expanding data sets. Because storage requirements oftentimes exhibit varying growth rates, current storage systems may not reserve a sufficient amount of excess space for future growth. Meanwhile, large up-front costs should not be incurred for a storage system that might only be fully utilized in the distant future. Therefore, a storage system that accommodates incremental growth would have major cost benefits. Incremental growth translates into a highly scalable storage system where the amount of overall storage space and throughput can dynamically expand according to the growth rate of its content data and/or application performance needs.

Our proposed scalable storage algorithms are generalized solutions for mapping a set of data objects to a group of storage units. Furthermore, the objects are striped independently of each other across all of the storage units for load-balancing purposes, that is, any object can be accessed with almost equal probability. This group of storage units has the quality that more units can be either added or removed, in which case the striped objects need to be redistributed to maintain a balanced load.

Scientific digital archives, distributed file systems, Web proxy servers, and continuous media (CM) servers are each examples of storage systems found in digital libraries. These systems experience growing data content and can benefit from a generalized and scalable storage solution. The Scientific Archive Management (SAM) system developed at the Pacific Northwest National Laboratory stores a very large, increasing amount of data generated from scientific experiments [21]. SAM relies on metadata and a virtual file system on its storage farms to allow scientists to store their data without worrying about the underlying data locality even when data is relocated due to system scale-up. Lustre[1] is an example of a highly scalable cluster file system that could benefit from efficient data redistribution techniques when adding more cluster nodes. A low-cost scaling technique is crucial for Lustre since it aims to scale up to tens of thousands of nodes. Distributed CM servers provide real-time access to libraries of digital media where the media files are declustered across a large set of disks. We assume CM servers in our discussions and use the terms “disk” and “file block” (i.e., of a file object) in place of “storage unit” and “data object,” respectively, throughout this paper. Finally, other familiar examples of exponentially growing data stores with high-scalability requirements are the Google[2] search

S.-Y. D. Yao (✉) · C. Shahabi · R. Zimmermann
University of Southern California, Computer Science Department,
Los Angeles, CA 90089
E-mail: didiyao@usc.edu

[1] http://www.lustre.org/
[2] http://www.google.com/

engine and the Internet Archive[3] library of digitized historical collections.

Our technique to achieve a highly scalable CM server begins with the placement of data on storage devices such as magnetic disk drives [11, 17]. More specifically, we break CM files (e.g., video or audio) into individual fixed-size *blocks* and apply a random placement [12] of these blocks across a group of homogeneous disks. Since any block can be accessed with an almost equal probability the random striping scheme allows the disks to be *load balanced* where their aggregate bandwidth and capacity (e.g., in bits/second and bytes, respectively) are maximized when accessing CM files. We actually use a pseudo-randomized placement of file object blocks, as in [6, 17], so that blocks have roughly equal probabilities of residing on each disk. With pseudo-random distribution, blocks are placed onto disks in a random, but reproducible, sequence.

The placement of block i is determined by its signature X_i, which is simply an unsigned integer computed from a pseudo-random number generator, p_r. p_r must produce repeatable sequences for a given seed. One way to derive the seed is from (StrToL[4](*filename*) $+ i$), which is used to initialize the pseudo-random function to compute X_i.

The storage system of a CM server requires that disks be scaled (i.e., added or removed) in which case the striped objects need to be redistributed to maintain a balanced load. Disks can be added to the system to increase overall bandwidth and capacity or removed due to failure or space conservation. We use the notion of *disk group* as a group of disks that is added or removed during a scaling operation. Without loss of generality, a scaling operation on a storage system with D disks either adds or removes one disk group. Scaling up will increase the total number of disks and will require a fraction of all blocks to be moved onto the added disks in order to maintain load balancing across disks. Likewise, when scaling down, all blocks on a removed disk should be randomly distributed across remaining disks to maintain load balancing. These block moves are the minimum needed to maintain an even load. Note that disk removals are only used for scaling down a system and conserving space. A disk removal and a disk failure are different in that data must be moved off of a disk before it is removed. With disk failures, data cannot be moved off a priori so a fault tolerance technique such as RAID-5 can be used in conjunction with RDL to prevent data loss. Our work is orthogonal to fault tolerance techniques such as RAID-5 since, in our scenario, each disk represents a logical storage unit and can potentially be a RAID device itself. Thus, our focus in this paper is on the issue of storage scalability.

We have previously developed the Random Disk Labeling algorithm to assign blocks to homogeneous disks using block signatures [23]. However, a homogeneous disk group may not be available at the time of scaling due to advancements in storage technology [7]. Thus, larger, faster, and more cost-effective heterogeneous disks must be used when scaling to increase the overall bandwidth and capacity characteristics of the storage system. The number of blocks on each disk should be proportional to *both* these characteristics. Load balancing according to just bandwidth may overflow some disks earlier than others since a disk with twice the bandwidth may not necessarily have twice the capacity.

In this paper, we propose the BroadScale algorithm for the disk assignment and scaling of heterogeneous disks. In addition to a block signature, a *disk weight* is assigned to each disk depending on its capacity *and* bandwidth. We will show that the system is load balanced after blocks are allocated according to both block signatures and disk weights.

As disks are added to and removed from the system, the location of a block may change. Our objective of course is to quickly compute the current location of a block, regardless of how many scaling operations have been performed. Moreover, we must ensure an even load on the disks and minimal block movement during a scaling operation. We summarize the requirements more clearly as follows.

Requirement 1 (even load): If there are B blocks stored on D disks, maintain the load so that the expected number of blocks on disk d is approximately $\frac{w_d}{\sum_{j=0}^{D-1} w_j} \times B$ where w_d is the weight of disk d.

Requirement 2 (minimal data movement): During the addition of n disks on a system with D disks storing B blocks, the expected number of block moves is $\frac{\sum_{j=D}^{D+n-1} w_j}{\sum_{j=0}^{D+n-1} w_j} \times B$. During the removal of n disks, $\frac{\sum_{w_j \in R} w_j}{\sum_{j=0}^{D-1} w_j} \times B$ blocks are expected to move where R is the set of disk weights of disks to be removed.

Requirement 3 (fast access): The location of a block is computed by an algorithm with space and time complexity of at most $O(D)$ and requiring no disk I/O. Furthermore, the algorithm is independent of the number of scaling operations.

We will show that the proposed BroadScale algorithm solves the problem of scaling heterogeneous disks while upholding Requirements 1, 2, and 3. With BroadScale, accurately computing disk weights could result in fractional weight values. Since BroadScale operates on integer weight values, the fractional portions, termed *weight fragments*, are wasted and cause load imbalance. For example, a disk weight of 3.5 has a wasted weight of .5. These weight fragments can be reclaimed through two techniques, disk clustering and fragment clustering. These techniques lead to less weight fragmentation even though additional block moves are incurred when scaling disks. However, we show through experimentation that this additional movement is marginal.

The remainder of this paper is organized as follows. Sect. 2 gives background on our Random Disk Labeling algorithm which is the basis for BroadScale. In Sect. 3, we describe our BroadScale algorithm. In Sect. 4, we introduce the concept of disk clustering and how they reduce inefficiencies in BroadScale. In Sect. 5, we describe another technique called fragment clustering. Section 6 describes related work.

[3] http://www.archive.org/

[4] StrToL is a C function that converts a string into a long integer.

In Sect. 7, we describe our experiments. Finally, Sect. 8 concludes this paper and Sect. 9 discusses future research.

2 Random Disk Labeling (RDL)

In this section, we provide background on our Random Disk Labeling (RDL) algorithm [23] for the scaling of homogeneous disks. Then, we give an initial attempt of using RDL for the scaling of heterogeneous disks.

2.1 Scaling with homogeneous disks

We adapt a hashing technique called *double hashing* to solve our problem of efficient redistribution of data blocks during disk scaling. Generally speaking, double hashing applies to hash tables where keys are inserted into buckets. We view this hash table as an *address space*, that is, a memory-resident index table used to store a collection of *slots*. Each slot can either be assigned a disk or be empty. Some slots are left empty to allow for room to add new disks. We can think of block IDs as keys and slots as buckets.

We design our address space for P slots (labeled $0, \ldots, P-1$) and D disks where P is a prime number, D is the current number of disks, and $D \leq P$. For this approach, we use a random allocation of disks where we randomly place D disks among the P slots. We can simply think of D disks which are labeled with random slots in the range $0, \ldots, P-1$, but we use the concept of disk occupying slots to help visualize our algorithm.

As explained in Sect. 1, each block has a signature, X_i, generated by a pseudo-random number function, p_r1. To determine the initial placement of blocks, we use a block's signature, X_i, as the seed to a second function, p_r2, to compute a random start position, sp, and a random step length, sl, for each block. We want to probe slots until a slot containing a disk is found. The sp value, in the range $0, \ldots, P-1$, indicates the first slot to be probed. The sl value, in the range $1, \ldots, P-1$, is the slot distance between the current slot and the next slot to be probed. We probe by the same amount, sl, in order to guarantee that we search all slots in at most P probes. As long as P is relatively prime to sl, this holds true [10]. Since P is a prime number, we can guarantee at most P probes. The first slot in the probe sequence that contains a disk becomes the address for that block.

Example 1 In Fig. 1, assume we have 5 disks randomly assigned to 101 slots ($D = 5, P = 101$). Using the blocks' signature X_i, we compute sp and sl for blocks 0 and 1. For block 0, $sp = 3$ and $sl = 76$. Slot 3 has a disk so this becomes the address for block 0. For block 1, $sp = 46$ and $sl = 20$. Slot 46 does not contain a disk so we traverse block 1's probe sequence, probing by 20 slots and wrapping around if necessary, until we arrive at slot 5. ∎

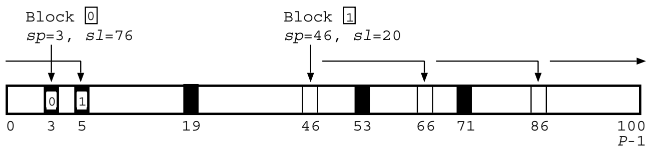

Fig. 1 Placement of two blocks. Block 0 initially hits and block 1 initially misses ($D = 5, P = 101$)

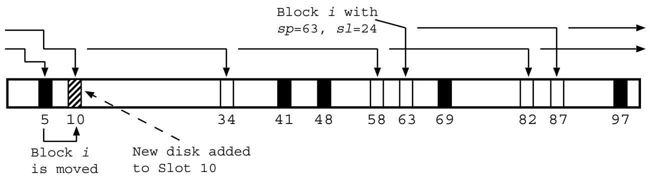

Fig. 2 Probe sequence of block i before and after a disk add operation j. Block i moves from disk 5 to disk 10 after disk 10 is added

For an addition operation, n disks are added to n randomly chosen empty slots. Then each block is considered in sequence $(0, \ldots, B-1)$ and, without actually accessing the blocks, we compute X_i, sp, sl, and the new location for block i. If the new location is an old disk, then the block must already lie on this disk so no moving is necessary. Clearly in this case, the probe length remains the same as before. However, if the new location is a new disk, then we continue with the probe sequence to find the block at its current location in order to move it to the new disk. In this case, the probe length becomes shorter since, before this add operation, this new location was also probed but was empty so probing continued.

Example 2 Figure 2 shows an example of adding a new disk to a set of 5 disks. The disk is added to the randomly chosen slot 10. Here, $sp = 63$ and $sl = 24$ so the probe sequence is 63, 87, 10, 34, 58, 82, 5 and block i belongs to the disk in slot 5. After scaling operation j, a disk is added to slot 10 and block i moves from the disk in slot 5 to the disk in slot 10 since slot 10 appears earlier in the probe sequence. The resulting probe sequence is 63, 87, 10. ∎

Without loss of generality, disks are randomly chosen for removal. For removal operations, we first mark the disks which will be removed. Then, for each block stored on these disks, we continue with its probe sequence until we hit an unmarked disk to which we move the block. The probe length is now longer (but no longer than P trials) to find the new location. This can be illustrated as the reverse of Example 2. We reiterate that disk removals and disk failures are different and that our scalability techniques are orthogonal to fault tolerance techniques.

In all cases of operations, the probe sequence of each block stays the same. It is the probe length that changes depending on whether the block moves or not after a scaling operation. Hence, the scaling operation and the existence of disks will dictate where along the probe sequence a block will reside. After any scaling operation, the block distribution will be identical to what the distribution would have been if the disks were initially placed that way. The amount of block movement is minimized since blocks only move from old disks to new disks. For any sequence of scaling operations, RDL will result in a uniform distribution of blocks across homogeneous disks since blocks have an equal chance of falling into any slot. Also, blocks are quickly accessible since locating blocks only requires a maximum of P probes within the memory-resident address space.

Finally, there is a trade-off between large and small P values [23]. We want to set P to a large enough value to allow for more scale-up room since the maximum number of disks that the storage system can scale up to is P. However, large P's require more probing since there are more empty slots to probe. Ideally, the growth rate of the storage system should be gauged beforehand to determine a good P. A system administrator could estimate the growth rate of the system and chose a P value which is effective for a certain amount of time. Once the number of disks reaches P (i.e., the slots are exhausted), a complete reorganization of the data blocks is required in order for P to be increased to allow for further scale-up.

2.2 Scaling with heterogeneous disks

In a heterogeneous disk system with different capacities and bandwidths, certain disks will tend to be favored more than others. If a disk has, for example, *twice* the bandwidth and capacity of the others, we want *twice* the amount of blocks hitting it. This means that the block assignments will not be uniform and must follow some weighting function, where each disk has an associated weight. We can achieve this by applying the *filter method* to RDL so that blocks do not end up on the first disk they hit in their probe sequence. Instead, they probe disks one-by-one until the filter method finds a target disk, based on the disk weight. The higher the weight, the more likely its corresponding disk will be a hit. We describe this method below as well as discuss its main drawback of extra block moves.

Given any block i, let the following denote its probe sequence: $P = \{d_0, d_1, \ldots, d_{D-1}\}$ where d_j is a disk and is unique within the probe sequence. Moreover, disk weights are assigned based on their bandwidth and capacity. We give details on determining disk weights in Sect. 3. The corresponding weights for the disks that are probed are given by: $W = \{w_0, w_1, \ldots, w_{D-1}\}$.

We now use the filter method for placing block i on a disk along its probe sequence. We define a *filter value* for each of block i's probes: $F = \{f_0, f_1, \ldots, f_{D-1}\}$ where:

$$f_j = \frac{w_j}{\sum_{k=j}^{D-1} w_k} \tag{1}$$

It is easy to see that $0 < f_j \le 1$ for all j and $f_{D-1} = 1$. In order to determine which disk this particular block belongs to, we use its signature X_i and the disk identifier, d_j, as seeds to a multi-seeded pseudo-random number function to generate a pseudo-random number r_j between 0 and 1. Starting with $j = 0$ to $D - 1$, we find the first disk d_j of P, where $r_j \le f_j$, to put block i.

We can now easily apply the filter method directly to RDL with varying disk weights by using filter values computed from Eq. 1. However, block movement after scaling will not be minimized in this way. The movement can only be minimized (moving only from old to new disks) if disks are added to or removed from the front of the probe sequence. Since every block has a different probe sequence, this will not be possible so some blocks will move from old disks to other old disks.

Thus, this characteristic of the filter method is undesirable and violates Requirement 2 (minimal data movement). Example 3 illustrates the additional amount of block movement when disks are not added to the front of the probe sequence using the filter method with RDL.

Example 3 Consider a homogeneous case of the filter method with RDL where initially $D = 2$. In Fig. 3a, $P = \{0, 1\}$, $W = \{1, 1\}$, and $F = \{0.5, 1\}$. Using X_i and $d_0 = 0$ as seeds, we compute a pseudo-random number, r_0. If $r_0 \le 0.5$ we place block i on disk 0, or disk 1 otherwise.

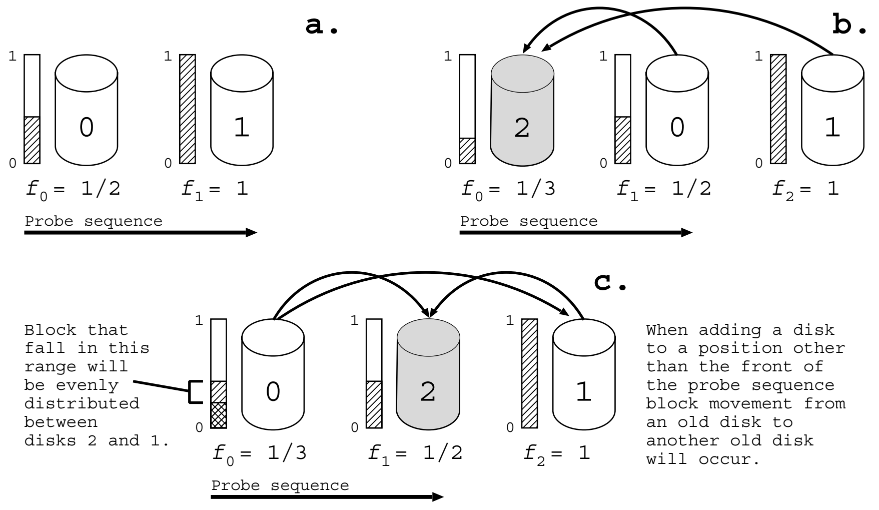
a.
b.
c.
1
0
f0 = 1/2
f1 = 1
Probe sequence
f0 = 1/3
f1 = 1/2
f2 = 1
0
1
2
Block that fall in this range will be evenly distributed between disks 2 and 1.
When adding a disk to a position other than the front of the probe sequence block movement from an old disk to another old disk will occur.

Fig. 3 Figure 3a shows an initial group of two disks. Disk 2 is added to the front of the probe sequence in Fig. 3b and to the middle in Fig. 3c

In Fig. 3b, we add disk 2 to the front of the probe sequence. Here, $P = \{2, 0, 1\}$, $W = \{1, 1, 1\}$, and $F = \{0.33, 0.5, 1\}$. We recompute r_0 using X_i and $d_0 = 2$ as seeds. If $r_0 \leq 0.33$ we place it on disk 2, otherwise we compute r_1 using X_i and $d_1 = 0$ as seeds. Now if $r_1 \leq 0.5$ block i is placed on disk 0, or disk 1 otherwise. In this case, if block i moves, it only moves from an old disk to a new disk. However, in Fig. 3c, if disk 2 is added between disk 0 and 1, then block i could move from disk 0 to disk 1, old disk to old disk. ▮

Even though the filter method does not work directly with RDL for heterogeneous disks, we will use a similar filter method as a component of our BroadScale algorithm described later in Sects. 4 and 5. In Sect. 3, we introduce BroadScale, an algorithm similar to RDL, which maps multiple slots to a single disk to support heterogeneous disks.

3 Disk weights

We have shown how RDL [23] can scale the size of a multi-disk system consisting of homogeneous disks using a random placement of the data. With the introduction of heterogeneous disks, a uniform distribution of blocks from RDL will not enable the disks to be fully utilized, assuming that all blocks are equally popular. In general, larger and faster disks should hold more blocks. Using the filter method with RDL attempts to achieve this, as described in Sect. 2.2, but leads to an undesirable characteristic of additional block moves.

In this section, we will describe our technique called BroadScale which extends RDL for the support of heterogeneous disks. BroadScale is based on RDL but the main difference is that each disk can be mapped to *multiple* slots depending on the weight value of the disk. In Sect. 3.1, we describe how to compute disk weights assuming a static group of disks that have different bandwidth to space ratios (BSR). In Sect. 3.2, we describe how to compute disk weights for a dynamically growing group of disks.

3.1 Disk weights for a static disk group

Instead of using the filter method directly with RDL, BroadScale assigns *multiple* slots to a single disk. The more slots assigned to a disk, the more blocks this disk will contain. We call the number of slots assigned to a particular disk the *weight* of the disk. Each disk may or may not have a different weight depending on its two characteristics: bandwidth[5] and capacity, measured in bits/second and bytes, respectively. Computing the weight from a combination of bandwidth and capacity is also described in [8]. Clearly, a disk of weight 10 will have twice as many blocks assigned to it than a disk of weight 5.

The weight of disk d is unitless and is computed by its normalized bandwidth B_d/B_{MAX} or normalized capacity C_d/C_{MAX} or a combination of both. When both bandwidth and capacity are considered, a system administrator could optionally set the weight to w'_d according to:

$$w'_d = \frac{B_d}{B_{\text{MAX}}} \times \beta + \frac{C_d}{C_{\text{MAX}}} \times (1 - \beta) \tag{2}$$

where β is the percentage of bandwidth contribution to w'_d. β is a tuning mechanism for the system administrator to adjust between the priorities of bandwidth and capacity. B_{MAX} and C_{MAX} can be set to estimated future maximum bandwidth and capacity values. Since w'_d's could be fractional numbers, we can divide them by w_G, which is the greatest common factor (GCF)[6] of the w'_d's, to obtain integer values for the weights. Hence, the disk weight, w_d, is computed using:

$$w_d = \frac{w'_d}{w_G} \tag{3}$$

Note that this assumption of w_d being an integer value may change when new disks are added resulting in weight fragmentation. Later, we reduce this fragmentation in Sects. 4 and 5.

Example 4 Suppose we have 2 disks where $B_0 = 10$ MB/s, $C_0 = 20$ GB, $B_1 = 20$ MB/s, and $C_1 = 40$ GB. If the disk weights should only depend on bandwidth ($\beta = 1$) and $B_{\text{MAX}} = 40$ then $w_G = 0.25$, $w_0 = 1$, and $w_1 = 2$. ▮

When computing disk weights, inefficiencies arise when the bandwidth to space ratio (BSR) of the disks are not all identical. If $\beta = 1$ then the number of blocks on disks depends solely on their bandwidth where $w_d = B_d/w_G$. The storage capacity utilized on each disk will be equivalent to the capacity of the disk with the highest BSR. Thus, some capacity will be left unused on those disks with lower BSRs. However, a restriction may occur on disks with higher BSRs. These disks will fill up more quickly than other disks since they have proportionally less capacity. In this case, the bandwidth of these disks will not be fully utilized. To create more room on these disks, more disks need to be added.

In Fig. 4a, $w_d = 1$ for $d = 0, 1, 2$ so each disk is assigned to one slot and receives an equal number of blocks. Disks 0 and 1 have under-utilized storage capacities because $\beta = 1$, indicating that the aggregate bandwidth should be fully-utilized. Hence, the maximum aggregate amount of useful storage is the capacity of the disk with the highest BSR multiplied by D.

On the other hand, if $\beta = 0$ then the amount of blocks on disks depends solely on their capacity so larger disks will contain more blocks, even if they have little bandwidth. Here, $w_d = C_d/w_G$. The bandwidth of disks with lower BSRs will be more stressed since they have proportionally less bandwidth than disks with higher BSRs. In this case, disks are restricted by their bandwidths since they might

[5] For simplicity, we use the average bandwidth since multi-zoned disks have various bandwidth characteristics.

[6] Since GCFs are traditionally integers, we can multiply the w'_d values by 10^x (where x is a user-defined precision value), truncate them to a whole number, and find the GCFs for these values instead.

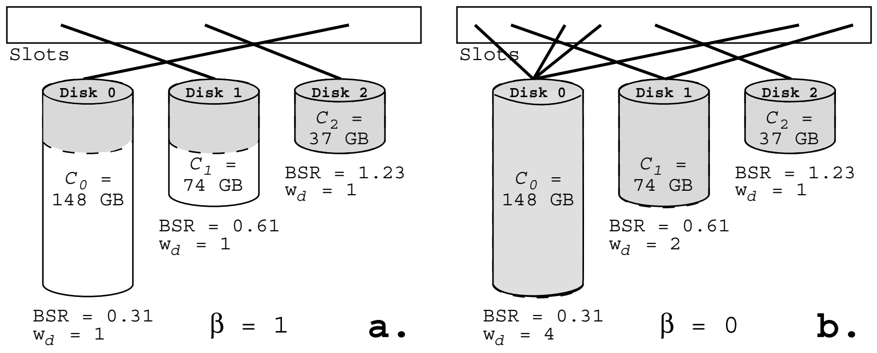

Fig. 4 Figure 4a shows unused capacity in disk 0 and 1 when $\beta = 1$. Figure 4b shows potential bottlenecks at disk 0 and 1 when $\beta = 0$. $B_d = 45.5$ MB/s for all disks

be slowed considerably. Figure 4b shows an example of this case where disks 0 and 1 contain more blocks than disk 2 but all have the same bandwidth. Since all blocks have an equal chance of being accessed, more block requests will be delivered to disks 0 and 1, creating potential bottlenecks.

Therefore, we can determine the weight of disk d using Eq. 3 to obtain an integer weight value that can map slots to disks. Next, we explore how to determine disk weights for a dynamically growing disk group.

3.2 Disk weights for a dynamic disk group

Since we allocate slots (and therefore blocks) to disks according to the weight of each disk, dividing the weight by a factor will have the effect of changing the number of slots allocated to the disk. Let's use w'_F to denote the dividing factor in general. The trade-off is that small w'_F values lead to larger w_d values, which tend to make any fractional value of the weight relatively insignificant but will require more slots. Having more slots increases the storage requirements of the address space, but more significantly, increases the total amount of probing to locate blocks [23]. On the other hand, larger w'_F values lead to smaller w_d values which could result in under-utilized disk resources due to the more significant fractional value of the weight.

Trend reports, such as [7], of the growth rate of magnetic disk technology allow us to estimate the characteristics of disks that will be manufactured in the *near*-future. We can use these estimations to help us determine w'_F. For example, if a system administrator anticipates adding new disks one year from now, w'_F can be computed from estimations of these disks' characteristics given the current trend of technology.

However, when disks are added to the system much later in the *far*-future, estimations of their characteristics (and therefore w'_F values) are more inaccurate. This inaccuracy could lead to fractional weight values and cause under-utilization of new disks. Higher disk utilization can be achieved by updating w'_F, however, this will impose high data reorganization costs.

We can better utilize unpredictable far-future disks without using an over-abundance of slots by using an estimation of w'_F which we call the estimated common factor (ECF) of the total weight, or $w'_{E,\alpha}$. The $w'_{E,\alpha}$ of a *new* disk group is computed such that the combined bandwidth and capacity usage will be at least α percent. $w'_{E,\alpha}$ is computed by Eq. 4:

$$\frac{\sum_{j=0}^{D-1}(w'_j \bmod w'_{E,\alpha})}{\sum_{j=0}^{D-1} w'_j} = 1 - \frac{\alpha}{100} \tag{4}$$

where D is the total number of disks in the new group and the numerator adds up all the fractional portions of the weights.

Therefore, $w'_{E,90}$ gives the ECF of the aggregate weights such that the utilization is at least 90%. Figure 5 shows the largest value of $w'_{E,90}$ that still achieves at least 90% utilization. Since far-future adds most likely involve disks with higher bandwidth and larger capacity, the weights of the new disks will be larger, thus the fractional weight portion will be smaller. For the rest of this paper, we refer to fractional weight values as the *weight fragmentation* of a group of disks. Fragmented weights, caused by the shaded regions in Fig. 5, lead to an under-utilization, or waste, of the disk weight (e.g., a weight of 3.5 has .5 of wasted weight).

We arrive at Eq. 5 which maintains at least an α percent of utilization for near- *and* far-future scaling operations.

$$w_d = \frac{w'_d}{w'_{E,\alpha}} \tag{5}$$

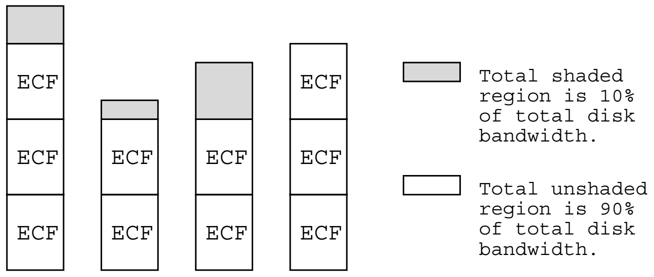
ECF
ECF
ECF
ECF
ECF
ECF
ECF
ECF
ECF
ECF
ECF
Total shaded region is 10% of total disk bandwidth.
Total unshaded region is 90% of total disk bandwidth.

Fig. 5 Total bandwidth of four disks. In this case, $w'_{E,90}$ is the ECF

Note that w_d may not be an integer value when a heterogeneous disk d is added. This is a problem since w_d is the number of slots assigned to disk d, which of course cannot be fractional. In Sects. 4 and 5, we describe two approaches for reducing the waste associated with weight fragmentation.

4 Disk clustering

The disk weights, as described in the previous section, might be fragmented and hence cannot be mapped directly to an integer number of slots in RDL's address space. In this section, we explore how to map fragmented disk weights to slots using *disk clustering* where clusters have almost integer weights. For example, a disk weight of 1.5 cannot be mapped to 1.5 slots, but it can be clustered with another disk of weight 2.5 and mapped together to four slots. The idea is to try to reduce the fractional portion of the aggregated disk weights in each cluster as much as possible. Then each cluster is assigned to one or more slots instead of each disk being assigned to slots. Actually, now the slots have no knowledge of the disks at all. The higher the cluster weight, the more slots it is assigned to and the more blocks that will be assigned to the cluster. A question that remains is that after a block is assigned to a cluster, which disk should it reside on within the cluster?

For the rest of this section, we first describe the simple case of clusters with only one disk. Then we show that using multiple disks per cluster can reduce the waste of disk resources. Finally, we describe how to locate a disk within a disk cluster for block assignment. We are not concerned with the terms disk bandwidth and capacity in our discussions in Sects. 4 and 5 since they have both been translated into the concept of disk weights.

4.1 Single disk clusters

One method to accommodate disks with fragmented weights is to simply use $\lfloor w_d \rfloor$ as the weight of disk d. Effectively, this method uses disk clusters that each contain only one disk. For each cluster, the maximum amount of waste would then be less than one unit of weight.

If the weights of new disks are relatively high then the percentage of waste may not be significant. However, some low disk weights may actually be less than 1 in which case they cannot be mapped to any slots and are, in effect, unusable. The single-disk cluster solution may suffice for steady or increasing disk weights as disks are added, but is clearly inadequate for fragmented, low disk weights.

4.2 Multiple disk clusters

By logically clustering multiple disks together, we can reclaim the fractional portions of the disk weights and reduce the amount of waste. Instead of using individual disk weights, cluster weights map a cluster of one or more disks to the appropriate number of randomly chosen slots.

The fragmentation of a cluster's weight can decrease as disks are added to the cluster. The cluster's weight is the sum of its disks' weights. The objective is for the cluster weight, w_c, of cluster c to be as close to $\lfloor w_c \rfloor$ as possible. Since it may be hard for w_c to be exactly equal to $\lfloor w_c \rfloor$, the cluster is said to be *full* when $(w_c - \lfloor w_c \rfloor) \leq (1-\epsilon)$. ϵ is specified by the user to indicate when a cluster is full (i.e., $\epsilon \times 100$ percent full). Thus, when $\epsilon = 0.95$, a cluster is full when the fractional portion of w_c is less than 0.05. A new disk, d, is either added to an existing non-full cluster or it is added to a new empty cluster by itself. The disk is added to a cluster such that the fractional portion of w_c is reduced after the inclusion of w_d. In other words, disk d is added in such a way that the overall waste of the storage system is reduced.

To decide where a new disk is added, we can generalize our problem to the classical *bin packing* [4] problem. The objective of bin packing is to pack variable-sized items in as few bins as possible thus, each bin becomes as full as possible. Our main objective is also to pack each cluster as full as possible with disks. For our problem, we will only consider the fractional portion of the disk and cluster weights. Disks are items and clusters are bins, but the fractional portion of the disk weights are the item sizes and the clusters (bins) are of size 1. A slight difference to traditional bin packing is that a disk can only be packed into a cluster if it *reduces* the fractional portion of the sum of the disk sizes in that cluster. An easy way to translate this back to traditional bin packing is to use Eq. 6 to compute disk sizes:

$$s(w_d) = 1 - (w_d - \lfloor w_d \rfloor) \tag{6}$$

where w_d is a disk weight and $s(w_d)$ is the disk size. Hence, by packing disks into clusters of size 1, the weight fragmentation of clusters is reduced.

With traditional bin packing, all items (disks) are known beforehand so an optimal packing arrangement does exist even though it cannot be found in polynomial time (an NP-hard problem). However, since we have no prior knowledge of how many future disks there are, we must optimally rearrange the entire packing for *each* new disk, requiring most blocks to be moved each time. Obviously, this solution is infeasible. Therefore, we use a heuristic such as the *Best Fit* [4] algorithm to optimize the placement of just the *next* disk to be added. Using *Best Fit*, disk d should be placed in a cluster that has current size closest to, but not exceeding, $1 - s(w_d)$. If this results in multiple clusters then the one with the lowest index is chosen as the tiebreaker. If disk d does not "fit" into any of the clusters, a new cluster is created with disk d. Thus, *Best Fit* searches all the clusters and picks the best one to which the disk should be added.

Large-scale storage systems may have on the order of $10,000$ disks (e.g., Lustre) so an exhaustive search to find the best cluster to place a disk may be computationally intensive. In these cases, the *First Fit* [4] algorithm may be more appropriate since it simply picks the first cluster in which the disk fits. *First Fit* and *Best Fit* are both good approximations

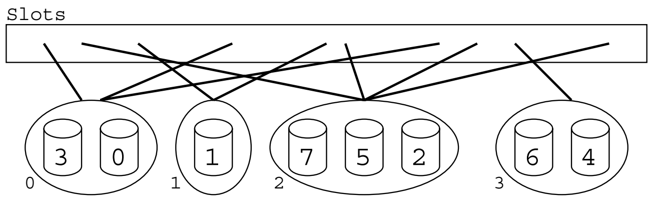

Fig. 6 Four clusters of disks. Each cluster is mapped to one or more slots. The disks are heterogeneous but do not appear so in order to simplify the figure

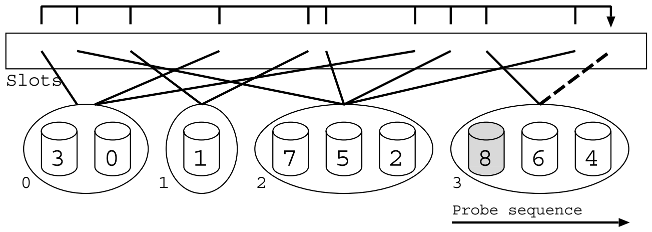

Fig. 7 After disk 8 is added, a new slot is mapped to cluster 3 since w_3 increases. Some blocks in clusters 0, 1, 2 are moved to disks 8, 6, and 4

to the optimal solution of traditional bin packing since they require at most 70% more bins than the optimal number of bins [4].

After disk d is added to cluster c, the number of slots assigned to the cluster is $\lfloor w_c \rfloor$. If this represents an increase in slots, then more empty slots are randomly assigned to the cluster. Figure 6 shows four clusters of disks, each mapped to one or more slots. Without loss of generality, removing a disk from cluster c reduces w_c and slots are randomly chosen to be unassigned to this cluster. Using RDL and the concept of disk clusters, blocks are uniformly distributed across the slots and are thus proportionally distributed across the clusters based on $\lfloor w_c \rfloor$. Now we can find which cluster a particular block belongs to using the slot to cluster mapping. In the next section, we describe how to find a particular disk *within* a cluster for the block.

4.3 Locating a disk within a cluster

Once a disk cluster is found for a block using RDL, we must locate a disk within the cluster for the block to reside. The likelihood that a block will land on disk d within a cluster c is w_d/w_c since this is the percentage of weight of this disk among all the disks in the cluster. To achieve this distribution, we use the filter method described in Sect. 2.2 except that the probe sequence is only of the disks within the cluster with the first disk of the sequence being the most recently added one. More importantly, the probe sequence is now the same for all blocks.

To locate a disk, first we logically arrange the disks within the cluster in decreasing order of their disk identifiers, d. Note that these disk numbers may not be contiguous within a cluster since new disks could have been added to different clusters at different times. Then, for each disk in this sequence starting with the first disk (the one with the highest disk identifier), we use the signature, X_i, of block i and the disk identifier, d, as seeds to a pseudo-random number function to generate a random value, r_0, in the range $0 \ldots 1$. One example of a well-performing function,[7] as suggested by [20], is:

```
srand(d)
srand(rand()^X)
r = rand()/R
```

[7] There are other types of pseudo-random number functions to consider, but finding a good one is hard [13].

where R is the range of `rand()`. Next, if r_0 is less than or equal to the filter value, f_0, then block i should reside on the 0th disk of cluster c's probe sequence. If $r_0 > f_0$ then we compare r_1 and f_1 to determine if the block should reside on the 1st disk, and so on. This filter value is computed using Eq. 7, similar to Eq. 1.

$$f_j = \frac{w_j}{\sum_{k=j}^{D_c-1} w_k} \tag{7}$$

where D_c is the number of disks in cluster c and w_j is the weight of the j-th disk in the cluster's probe sequence.

Since new disks are always added to the front of the probe sequence for cluster c, we can guarantee that blocks will only move from the old disks to the new disks within the cluster. However, some blocks from outside the cluster may move to the old disks within the cluster as shown in Fig. 7. This occurs when adding a disk to a cluster increases the cluster weight, w_c, by more than 1 and causes an increase in slots that are mapped to the cluster. Thus, RDL will move some blocks from every old slot to the new slots. Blocks that are assigned to the new slots all have a chance of landing on any disk in the cluster, so some blocks may end up on an old disk of the cluster. We will show in Sect. 7 that the amount of this additional movement is not significant. Note that the movement from old disks to old disks will not cause any unevenness in the block distribution, only the consumption of additional disk bandwidth and possibly additional network bandwidth if the disks are separated across a network. We will also explain in Sect. 7 that having larger clusters requires more computation to locate disks within a cluster. Thus, the trade-off is between less computation or less wasted disk weight.

Finally, disk removals are handled in a similar reverse fashion as disk additions. If a disk is removed from a cluster and the new $\lfloor w_c \rfloor$ is less than the previous $\lfloor w_c \rfloor$, then this cluster should be mapped to fewer slots. In this case, slots are randomly chosen to be unmapped from the cluster and blocks are moved off of the cluster accordingly. A small amount of block reorganization will occur within the cluster if the removed disk is not the first disk in the probe sequence of the cluster since we are using the filter method to locate blocks within the cluster. However, since the number of these old disk to old disk block moves is small and occurs only within one cluster, the amount of these additional moves is insignificant compared to the overall number of moves.

5 Fragment clustering

As described in the previous section, disk clustering is a way to reduce the overall weight fragmentation by selectively adding disks to clusters, thereby reducing cluster weight fragmentation. Another approach to reducing weight fragmentation is *fragment clustering* where only the fractional portions of the disk weights are clustered together. With fragment clustering, physical disks are mapped to 1 or more randomly chosen slots. Then the fractional disk weight portions are grouped together into unit-sized logical disks and each logical disk is mapped to 1 randomly chosen slot. The details of this approach are described as follows.

When a new heterogeneous disk d is added to the storage system, it is mapped to $\lfloor w_d \rfloor$ randomly chosen slots in the virtual address space. If the fractional portion of w_d, namely $w_d - \lfloor w_d \rfloor$, is greater than 0, its fractional value, along with a pointer to disk d, is appended as an entry in the unit-sized *Pending Logical Disk* (PLD). Figure 8 shows an example of mapping a disk of weight 3.5 to 3 slots with the fractional weight .5 appended as an entry in the PLD. The fragment clustering algorithm maintains only one PLD, which stores the currently unutilized weight fragments. The PLD becomes full when the sum of its fractional values is greater than or equal to 1.0. Once this sum is greater than 1.0, the PLD becomes an *Active Logical Disk* (ALD), and the excess value of the sum (i.e., $sum - 1.0$), along with a pointer to d, is stored as an entry in a newly allocated PLD. This ALD is now mapped to a randomly chosen slot. Of course, the number of ALDs increases as more disks are added, whereas we only maintain one PLD.

Once the disks are mapped to slots in the manner just described, data blocks can be assigned to disks using RDL as normal by probing the slots. Figure 9 illustrates the placement of block 1 directly onto a physical disk and block 2 onto a physical disk via a logical disk (i.e., an ALD). When a slot mapped to a physical disk is probed, that block is assigned to the disk. However, when a slot mapped to an ALD is probed, further computation must be done to determine which physical disk this block eventually resides on. To accomplish this, we use the filter method from Sects. 2.2

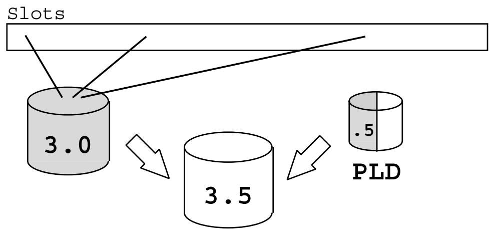

Fig. 8 A disk of weight 3.5 is mapped to three slots. The .5 fractional value along with a pointer to the disk is stored as an entry in the PLD. The PLD is activated into an ALD when it is full

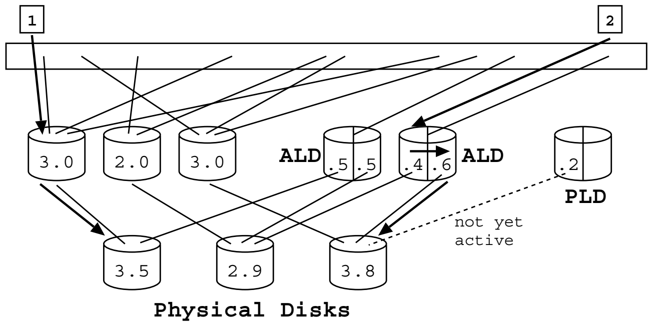

Fig. 9 Block 1 probes slots using RDL until it lands on the disk with weight 3.5. Block 2 probes slots and lands on an ALD. Within the ALD, the filter method determines Block 2 to hit the entry with value .6 and is forwarded to the disk with weight 3.8. Note that the pointer of the entry in the PLD is not yet active since the PLD is not yet full

and 4.3 on the ALD, which contains entries of fractional values summing up to 1.0. Once an entry is found using the filter method, the pointer within that entry is followed to the physical disk where the block should be placed.

Since adding new disks will never cause updates to entries in the ALDs (but do cause PLD updates), we do not need a specific initial ordering (i.e., least recently added to most recently added) of the entries. However, once an initial entry ordering is decided from the construction of the PLD, this ordering must remain the same after conversion to ALDs.

With disk removals, there will be a small amount of additional block moves incurred due to the removal of entries within ALDs. When a disk is removed, the entry containing the fractional portion of its weight is also removed causing an ALD to have a weight of less than one. The remaining entries in the ALD can be combined with any entries in the PLD to become a full ALD and/or a partially filled PLD. Additional block moves will result when some blocks are moved from non-removed disks to other disks. This will occur during disk removals since removing an entry in an ALD will unmap it from a slot and cause blocks in the remaining entries to be relocated. However, these additional block moves are also a small percentage of the overall moves.

Intuitively, using fragment clustering, the overall amount of weight fragmentation at any given time will always be less than 1.0. This fragmentation will only be contributed by the fractional values in the PLD. When the PLD fills to capacity (i.e., 1.0), it becomes active and its weight fragments are utilized. We show that this is true in Sect. 7.

In sum, BroadScale is a technique which first involves computing a weight for each disk based on bandwidth and/or capacity as described in Sect. 3. If these weights are not integer values, then we have weight fragmentation and wasted disk resources will arise. BroadScale reduces weight fragmentation through two approaches, disk clustering and fragment clustering. Disk clustering strategically clusters disks together using either the Best Fit algorithm or the First Fit algorithm to reduce fragmentation. Fragment clustering is another approach where the fractional portion of the weights are grouped as logical disks. A comparison of disk clustering with fragment clustering is discussed in Sect. 7 along with benefits and drawbacks of each.

6 Related work

We describe related work on two categories of applications to which we can apply our BroadScale algorithm. These categories are redistributing CM blocks on CM server disks and remapping Web objects on Web proxy servers.

Previous literature on CM servers have discussed areas such as distributed architectures and retrieval scheduling [11, 17]. The topic of homogeneous and heterogeneous disk scaling in CM servers has been the focus of a few past studies. One study mixes popular (hot) and unpopular (cold) CM data objects together on heterogeneous disks with different BSRs [3]. Their objective is to maximize the utilization of both bandwidth and capacity while maintaining the load balance. However, the popularity of the objects need to be known ahead of time to be properly placed. Moreover, their popularity might change over time (e.g., new movies tend to be accessed more frequently) so the objects may need to be moved depending on their current popularity. Other techniques stripe fixed-size object blocks, described below, as opposed to storing them in their entirety.

Disk scaling with round-robin data striping is discussed in [5]. With round-robin, almost all blocks need to be relocated when scaling. The overhead of such block movement may be amortized over a period of time but it is, nevertheless, significant and wasteful. Wang and Du [22] describe a technique which assigns weights to disks based on bandwidth and capacity. However, they also distribute data blocks in a round-robin fashion, requiring large block movement overhead when scaling. Another technique called Disk Merging [24] merges a static group of heterogeneous physical disks into homogeneous logical disks to maximize bandwidth and capacity for striped data. This technique is not intended for dynamic scaling since the system must be taken off-line and reconfigured, potentially reshuffling many blocks.

While traditional constrained placement techniques such as round-robin placement allow for deterministic service guarantees, random placement techniques are modeled statistically. The RIO project demonstrated the advantages of random data placement such as single access patterns and asynchronous access cycles to reduce disk idleness [12]. However, they did not consider the *dynamic* rearrangement of data due to disk scaling. Although they do not require prior knowledge of object popularity for full utilization of heterogeneous disks' aggregate bandwidth, their solution requires data replication for short- and long-term load balancing [15]. In one scenario, they require at least 34% block replication for 100% bandwidth utilization. Another study focused on the trade-off between striping and replication for load balancing [2]. For large systems, the extra storage needed for replication becomes more significant. In general, random placement, or pseudo-random in our case, increases the flexibility to support various applications while maintaining a competitive performance [16]. We developed a prior technique called SCADDAR that redistributes data blocks after homogeneous disk scaling in a CM server by mapping the block signatures to a new set of signatures for an even, randomized distribution [6]. SCADDAR supports disk additions just as well as disk removals. SCADDAR adheres to the requirements of Sect. 1 except that the computation of block locations become incrementally more expensive. Finding a block's location requires the computation of that block's location for every past scaling operation, so a history log of operations must be maintained. In comparison, our RDL and BroadScale algorithms are fast in computation even though they are limited by the total number of disks (i.e., P).

Another technique to scale heterogeneous disks is described in [8]. This technique attempts to achieve similar requirements in load balancing, minimal block moves, and fast data access. However, its major drawback is that is does not allow disk removal scaling operations. Our SCADDAR, RDL, and BroadScale algorithms support both additions and removals of homogeneous and heterogeneous storage devices.

Several past works have considered mapping Web objects to proxy servers using requirements similar to those described in Sect. 1. Below we describe two relevant techniques called highest random weight (HRW) and consistent hashing along with their drawbacks.

HRW was developed to map Web objects to a group of proxy servers [20]. Using the object name and the server names, each server is assigned a random weight. The object is then mapped to the highest weighted server. After adding or removing servers, objects must be moved if they are no longer on the highest weighted server. The drawback here is that the redistribution of objects after server scaling requires $B \times D$ random weight function calls where B is the total number of objects and D is the total number of proxy servers. A simple heterogeneous extension to HRW is described in [14], but suffers from the same computational complexity. We show in [23] that in some cases HRW is several orders of magnitude slower than our RDL technique. An optimization technique for HRW involves storing the random weights in a directory, but the directory size will increase as B and D increase causing the algorithm to become impractical.

Consistent hashing is another technique used to map Web objects to proxy servers [9]. Here objects are only moved from *two* old servers to the newly added server. A variant of consistent hashing used in a peer-to-peer lookup server, Chord, only moves objects from *one* old server to the new server [19]. In both cases, the result is that objects may not be uniformly distributed across the servers after server scaling since objects are not moved from *all* old servers to the new server. With Chord, a uniform distribution can be achieved by using virtual servers, but this requires a considerable amount of routing meta-data [1].

7 Experiments

In this section, we describe our simulation experiments to validate our BroadScale algorithm. First, we show that data blocks are distributed across the disks according to the disk weights. The higher the weight, the more blocks will reside on the corresponding disk. Next, we measured the amount of weight fragmentation from disk clustering and fragment clustering. With disk clustering, varying the size of the clusters affects the amount of fragmentation. Then, we show that the additional amount of block movement using disk clustering is not significant compared to the overall number of moves. This movement is even lower with fragment clustering. Finally, the average and maximum number of probes is shown for disk and fragment clustering.

For all of our experiments, we distributed approximately 750, 000 blocks across 10 initial disks, which is a realistic starting point. We set the total number of slots to 1, 511 (*i.e.*, $P = 1,511$) because we need room to add disks and multiple slots are mapped to each disk depending on the disk weight. We computed disk weights for a dynamic disk group using Eq. 5 by setting $\beta = 1$, where the number of blocks on disks depends solely on disk bandwidth. We set $\alpha = 90\%$ so that at least 90% of the aggregated disk weight is utilized to determined the number of blocks per disk. When simulating disk scaling, we assume a 10-disk add operation is performed every 6 months. For this time period, industry trends suggest that disk bandwidth increases 1.122× in [7], and disk capacity increases 1.26× following Moore's Law. Our added disks follow these trends.

The disk weights are used to indicate how many blocks should reside on a disk relative to other disks. Since the number of slots assigned to a disk is roughly equal to the disk weight, more slots assigned to the disk will result in more blocks for the disk. Figure 10 shows blocks distributed across 10 disks by BroadScale. For illustration purposes, these disks vary widely in bandwidth and, therefore, in weight. After distributing the blocks, the trend of the amount of blocks per disk follows the trend of the disk weights. The blocks per disk (w.r.t. the left axis) and the disk weights (w.r.t. the right axis) are overlaid together on the same figure to show their similarity. Moreover, as expected, Fig. 10 shows that the normalized curve (w.r.t. the left axis) is quite uniform across disks. The normalized curve is computed as (blocks on disk d/weight of disk d).

We assume that all data blocks have a similar probability of being accessed. However, the block access will be load balanced even for skewed access distributions, such as a Zipf distribution, when storing a large number of files using random block placement. If certain *files* are more popular than others, the blocks in these popular files will be randomly distributed across all the disks just as the blocks in unpopular files. In this case, the number of popular blocks on each disk is similar. Furthermore, certain *blocks* within a file may be more popular than other blocks within that file. Again, randomly distributing all blocks will result in a similar number of popular blocks on each disk.

Disk clustering and fragment clustering were two techniques introduced in Sects. 4 and 5. The purpose of clustering is to reduce the fragmentation of the disk weights, thus reducing the waste, so that each disk will hold a more accurate number of blocks. Figure 11 shows the aggregated waste of disk weights using both techniques as the storage system is scaled by adding 10 disks at a time with 10 initial disks. For disk clustering, we want to show that the total amount of unutilized disk weight decreases as clusters increase in size. When the maximum cluster size, K_{MAX}, is 1, the effect is that there is no clustering. Here, disk d is assigned to $\lfloor w_d \rfloor$ slots and the amount of wasted disk weight is significant. However, increasing K_{MAX} to 2 gives us much better weight utilization since the clusters are combining fragmented weights. Furthermore, setting $K_{\mathrm{MAX}} = 3$ leads to even greater improvement. We found that disk clusters became full at 3 disks, which is the expected value of the cluster size, so increasing K_{MAX} beyond 3 gave no improvement.

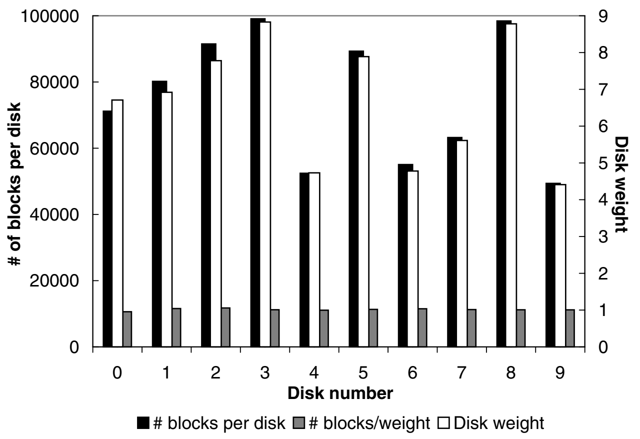

Fig. 10 The number of blocks per disk follows the same trend as the disk weight

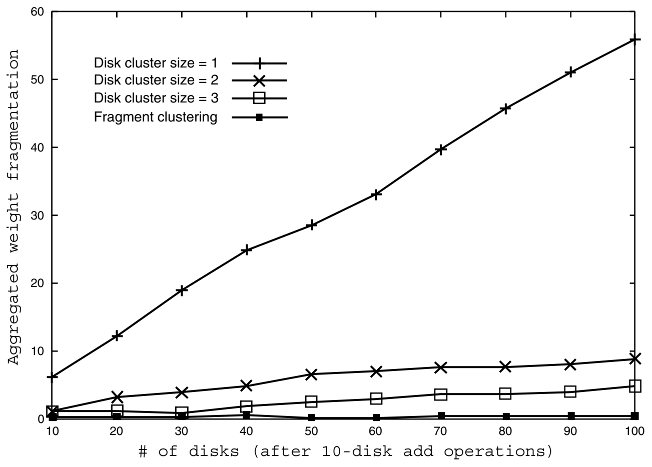

Fig. 11 The overall amount of weight fragmentation using disk clustering and fragment clustering

Using the Best Fit algorithm for disk clustering, the expected value, $E(D_c)$, of the number of disks on cluster c can be determined by analyzing the fractional part of the disk weights. Given a disk weight, the expected value of the fractional portion is .5. The second weight must reduce the fractional portion when summed with the first, so the expected value becomes .25. Each time a weight is added in this way, the expected fractional value is halved so we have:

$$0.5^{E(D_c)} = 1 - p \tag{8}$$

where p is the precision of $E(D_c)$ since $0.5^{E(D_c)}$ will never equal 0. For example, with a precision of 0.94, $E(D_c) = 4$ disks. Solving for $E(D_c)$, we arrive at the following:

$$E(D_c) = \log_{0.5}(1 - p) \tag{9}$$

Fragment clustering demonstrates the best performance since the maximum amount of total fragmentation is always less than a weight of 1.0. This is attributed to the fractional values stored in the PLD. However, with fragment clustering, newly added disks are almost never fully utilized since the PLD contains weight fragments only from these recently added disks. Nevertheless, this may become insignificant as the disk weights increase.

There exists a trade-off between low computation and low weight fragmentation for disk clustering since finding which disk within a cluster a block resides requires less computation for smaller clusters. To find a block located in cluster c using the filter method, on average, the pseudo-random function is invoked for half of the disks in c. Hence, finding a disk within small clusters requires less computation, but results in more weight fragmentation. However, since the expected number of disks per cluster, from Eq. 9, is low and we observe low weight fragmentation in Fig. 11 with small clusters (of size 3), high computation is not required. For fragment clustering, we cannot change the size of the PLD, but the number of PLD entries is low so finding a particular entry using the filter method is not costly. Below, we observe the overall amount of block movement of disk and fragment clustering.

In Sect. 4.3, we explained that adding disks could cause more blocks to be moved than the minimum that is required to fill the new disks. This is true of both clustering approaches. With disk clustering, the additional moves depends on the cluster size. If a disk is added to a new empty cluster, no extra moves are incurred. If a disk is added to a non-empty cluster, some blocks will be moved to the old disks in that cluster in addition to the new disks. Similarly, fragment clustering will result in these redundant moves when adding a disk causes the conversion of a PLD to an ALD. Since the PLD contains fractional entries from old disks, activating the PLD to an ALD will redistribute data from old disks to these old entries. However, the amount of these data moves is low since one ALD is a small component of the entire storage system.

Figure 12 shows the total amount of block movement when scaling disks 10 at a time with 10 initial disks. Here

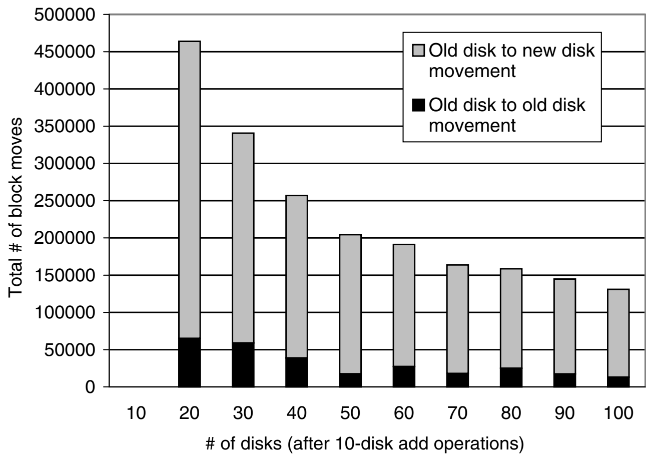

Fig. 12 The additional block movement (old disk to old disk) for disk clustering represents a small fraction of the total block movement

$\beta = 1$ so disk weights only represent the disk bandwidth, which grows 1.122× every scaling operation. We observe that the percentage of block moves from an old disk to another old disk is on average 13% of the total moves for disk clustering. Since cluster sizes tend to be small, from Eq. 9, and using small clusters is effective, the additional movement will not be a significant percentage of the total. We notice a decreasing trend in total block moves since the 10 disks that are added each time require fewer and fewer blocks to fill them, assuming the number of blocks is constant. A similar test on fragment clustering results in only around 3% redundant moves since a new ALD is small compared to the actual added disks.

Figure 12 employs the industry growth rate of disk bandwidth. For other growth rates, the percentage of old disk to old disk block movement decreases as higher growth rate disks are added for both clustering techniques. This is due to the fractional weight portion being proportionally smaller than the whole weight of these growing disks. Figure 13 shows the percentage of this redundant block movement

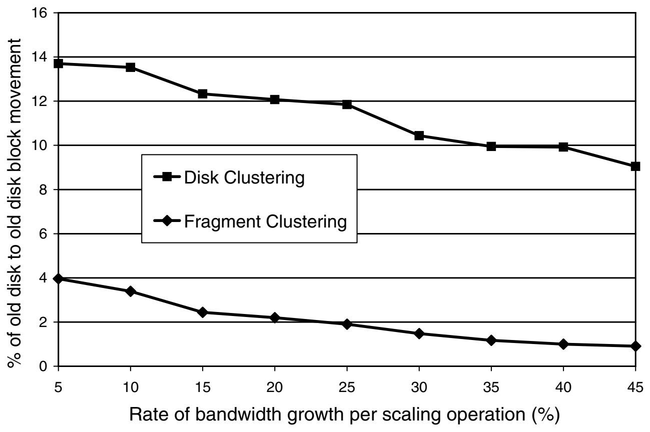

Fig. 13 The average percentage of redundant block movement for various growth rates

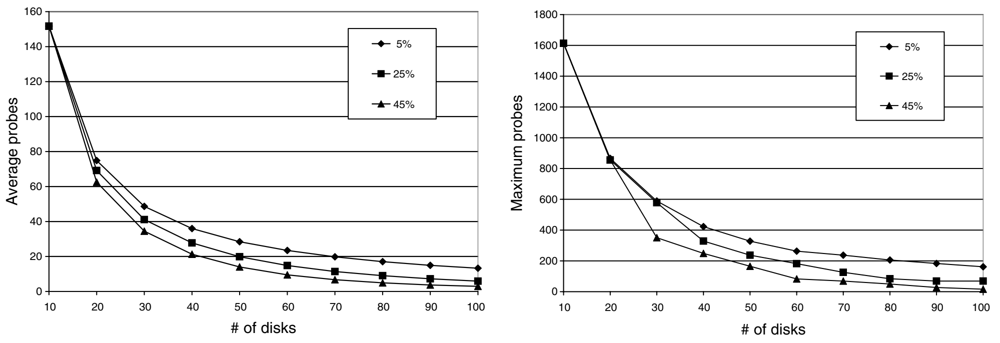

Fig. 14 Average and maximum probes for bandwidth growth rates of 5, 25, and 45% ($P = 10,000$)

with respect to the growth rate. For each growth rate value, the average percentage of redundant movement is measured as disks are scaled. The average percentage is calculated from 25 trials of each growth rate value with each trial using a different randomness factor to slightly vary the disk characteristics. From Fig. 13, fragment clustering exhibits less redundant movement than disk clustering. Redundant moves results from blocks being redistributed into old disks of a cluster and old fractional entries of an ALD in disk and fragment clustering, respectively. Fragment clustering has fewer redundant moves since it isolates these moves to just one ALD whereas disk clustering isolates these moves across an entire disk cluster.

Lastly, a higher growth rate when scaling disks should lead to less probing. The reason is that disks with larger weights will require more slots. This causes probing to be more successful in general since there are fewer empty slots and misses will be less frequent. Figure 14 shows the average and maximum number of total probes as disks are scaled 10 at a time beginning with 10 disks. The probing results of disk clustering and fragment clustering are similar and indistinguishable in the figure since the number of cluster groupings in each technique are similar. Figure 14a shows that the average number of probes is lower when scaling disks at a bandwidth growth rate of 45% than at a growth rate of 5%. Similar results are shown in Fig. 14b for the maximum number of probes.

Fragment clustering appears to be superior to disk clustering in weight fragmentation and redundant block moves. However, one drawback of fragment clustering is the additional bookkeeping required for the ALD entry pointers to physical disks. Moreover, within the ALDs and the PLD, the fractional values must be stored with these pointers. Another drawback is that newly added disks may not be fully utilized since their fractional weight portions are stored in the PLD and not yet activated.

8 Conclusions

BroadScale is a storage scaling algorithm that can benefit the storage systems of digital libraries and archives. BroadScale allows additions or removals of heterogeneous disks in a storage system where weights are assigned to disks depending on their bandwidth and capacity characteristics. Blocks are distributed among the disks proportional to these weights. Since only the integer portions of the weight values can be used to direct block placement, the fractional portions are wasted. However, these wasted portions, or weight fragments, can be strategically combined using either our disk clustering or fragment clustering approaches. BroadScale satisfies our scaling requirements of an even load according to disk weights, a minimum amount data movement when scaling disks, and the fast retrieval of data before and after scaling.

We have shown through experimentation that blocks are distributed proportionally to the disk weights using BroadScale. Disk scaling could lead to wasted disk weight (i.e., weight fragmentation), but can be substantially reduced through clustering. We observed significant improvement using larger cluster sizes in disk clustering. However, our fragment clustering technique is superior in overall weight fragmentation as well as average percentage of redundant block moves with a few minor drawbacks such as some extra bookkeeping. Although fragment clustering outperforms disk clustering, the additional block moves in either case was not significant compared to the total moves.

9 Future work

For future work, BroadScale can be extended to allow for scaling beyond P number of total disks by using an algorithm such as our previous algorithm SCADDAR [6]. For heterogeneous scaling with SCADDAR, we could use a similar weight function and assign disk d to $\lfloor w_d \rfloor$ slots.

We believe BroadScale can be generalized to map any set of objects to a group of scalable storage units. These objects might also require a redistribution scheme to maintain a balanced load. Examples of other applications include Web proxy servers and extent-based file systems. Scalability in integrated file systems that support heterogeneous applications [18] may also benefit from BroadScale.

Furthermore, we wish to integrate fault tolerance mechanisms with our scaling techniques. Each RDL slot is a storage unit which we assume to be a single disk drive. However, this storage unit can also represent a entire RAID device. So when scaling, a RAID device can be either added to or removed from a slot. Another more integrated approach is to keep primary block copies on the first disk of that block's probe sequence as usual and to add block replicas to the second disk of the probe sequence. We have shown that disks appear exactly once on every probe sequence so replicas are guaranteed to never be placed on the same disk as the primary copies.

Finally, we wish to investigate how BroadScale could be applied to storage systems that need to efficiently store a high influx of data streams such as those generated by large-scale sensor networks. We also want to explore data retrieval in large, scalable peer-to-peer systems or distributed hash tables. This requires a distributed implementation of BroadScale on top of these peer-to-peer search techniques.

Acknowledgements This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0082826 (ITR), IIS-0238560 (PECASE), IIS-0324955 (ITR), IIS-0307908, CMS-0219463 (ITR), and a NASA/JPL grant and unrestricted cash gifts from Okawa Foundation and Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Byers, J., Considine, J., Mitzenmacher, M.: Simple load balancing for distributed hash tables. In: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03) (February 2003)
2. Chou, C.-F., Golubchik, L., Lui, J.C.S.: Striping doesn't scale: how to achieve scalability for continuous media servers with replication. In: Proceedings of the International Conference on Distributed Computing Systems, pp. 64–71 (April 2000)
3. Dan, A., Sitaram, D.: An online video placement policy based on bandwidth to space ratio (BSR). In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 376–385, San Jose, California (May 1995)
4. Garey, M.R., Johnson, D.S.: Computer and intractability: A guide to the theory of NP-completeness, Chapter 6, pp. 124–127. W. H. Freeman and Company, New York (1979)
5. Ghandeharizadeh, S., Kim, D.: On-line reorganization of data in scalable continuous media servers. In: 7th International Conference and Workshop on Database and Expert Systems Applications (DEXA'96) (September 1996)
6. Goel, A., Shahabi, C., Yao, S.-Y.D., Zimmermann, R.: SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In: Proceedings of the 18th International Conference on Data Engineering, pp. 473–482 (February 2002)
7. Gray, J., Shenoy, P.: Rules of thumb in data engineering. In: Proceedings of the 16th International Conference on Data Engineering, pp. 3–10 (February 2000)
8. Honicky, R.J., Miller, E.L.: A fast algorithm for online placement and reorganization of replicated data. In: 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), Nice, France (April 2003)
9. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: Proceedings of the 29th ACM Symposium on Theory of Computing (STOC), pp. 654–663 (May 1997)
10. Knuth, D.E.: The Art of Computer Programming, vol. 3. Addison-Wesley, Reading, MA (1998)
11. Martin, C, Narayan, P.S., B. Özden, Rastogi, R., Silberschatz, A.: The fellini multimedia storage server. In: Chung S.M. (eds.) Multimedia information storage and management, Chapter 5. Kluwer Academic Publishers, Boston (August 1996). ISBN: 0-7923-9764-9
12. Muntz, R., Santos, J., Berson, S.: RIO: A real-time multimedia object server. In: ACM Sigmetrics Performance Evaluation Review, vol. 25 (September 1997)
13. Park, S.K., Miller, K.W.: Random number generators: Good ones are hard to find. Commun. ACM, **31**(10), 1192–1201 (1988)
14. Ross, K.W.: Hash-routing for collections of shared web caches. IEEE Netw. Mag., **11**(6), 37–44 (1997)
15. Santos, J.R., Muntz, R.R.: Performance analysis of the RIO Multimedia Storage System with Heterogeneous Disk Configurations. In: ACM Multimedia, pp. 303–308, Bristol, UK (September 1998)
16. Santos, J.R., Muntz, R.R., Ribeiro-Neto, B.: Comparing Random Data Allocation and Data Striping in Multimedia Servers. In: SIGMETRICS, Santa Clara, California (June 2000)
17. Shahabi, C., Zimmermann, R., Fu, K., Yao, S.-Y.D.: Yima: A Second Generation Continuous Media Server. IEEE Comput. pp. 56–64 (June 2002)
18. Shenoy, P., Goyal, P., Vin, H.M.: Architectural Considerations for Next Generation File Systems. Multimedia Syst., **8**(4), 270–283 (2002)
19. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 ACM SIGCOMM Conference, pp. 149–160 (2001)
20. Thaler, D.G., Ravishankar, C.V.: Using name-based mappings to increase hit rates. IEEE/ACM Trans. Network. **6**(1), 1–14 (1998)
21. Thomson, J., Adams, D., Cowley, P.J., Walker, K.: Metadata's role in a scientific archive. IEEE Comput. **36**(12), 27–34 (2003)
22. Wang, Y., Du, D.H.C.: Weighted striping in multimedia servers. In: Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS '97), pp. 102–109 (June 1997)
23. Yao, S.-Y.D., Shahabi, C., Larson, P.-Å.: Hash-based labeling techniques for storage scaling. The VLDB journal: The international journal on very large data bases (2004). ISSN: 1066-8888 (Paper) 0949-877X (Online), DOI: 10.1007/s00778-004-0124-6, Issue: Online First.
24. Zimmermann, R.: Continuous media placement and scheduling in heterogeneous disk storage systems. Ph.D. Dissertation, University of Southern California, Los Angeles, California (December 1998)