SCADDAR: An Efficient Randomized Technique to Reorganize Continuous Media Blocks *

Ashish Goel, Cyrus Shahabi, Shu-Yuen Didi Yao, and Roger Zimmermann
Computer Science Department
University of Southern California
Los Angeles, California 90089
[agoel, shahabi, didiyao, rzimmerm]@usc.edu

Abstract

Scalable storage architectures allow for the addition of disks to increase storage capacity and/or bandwidth. In its general form, disk scaling also refers to disk removals when either capacity needs to be conserved or old disk drives are retired. Assuming random placement of blocks on multiple nodes of a continuous media server, our optimization objective is to redistribute a minimum number of media blocks after disk scaling. This objective should be met under two restrictions. First, uniform distribution and hence a balanced load should be ensured after redistribution. Second, the redistributed blocks should be retrieved at the normal mode of operation in one disk access and through low complexity computation. We propose a technique that meets the objective, while we prove that it also satisfies both restrictions. The SCADDAR approach is based on using a series of REMAP *functions which can derive the location of a new block using only its original location as a basis.*

1. Introduction

Continuous media (CM) servers are becoming more popular as networks become increasingly able to deliver high-quality media. At the same time, the amount of media such as video, audio, and interactive virtual reality is rapidly growing. Today's CM servers may not be well suited to handle tomorrow's ever-increasing media sizes and bandwidth requirements. Thus, it is necessary to build a scalable CM server which allows the addition of new disks, the replacement of older model disks, and quite possibly the removal of older model disks. Adding disks to a CM server will increase overall server capacity and/or bandwidth without the need to judge and predict the amount of capacity and/or bandwidth needed during the initial setup of the server. Moreover, adding newer generation disks (with higher bandwidth and more capacity) to a CM server may cause the existing disks to become bottlenecks during data transfer [18]. These existing disks may eventually need to be replaced with newer disks or simply removed.

Note that removing disks is different from disk failure because disk removal is known *a priori* while disk failure is unpredictable. Hence, during disk removal, necessary steps can be taken before the actual removal so the data can be properly redistributed to other disks, thereby allowing the CM server to function in its normal mode of operation after disk removal. While the focus of this paper is on disk scaling (including disk removal), later in Section 6 we explain a simple extension to our proposed technique to address fault tolerance as well.

The need for a scalable continuous media server is clear, but the data residing on the CM server needs to be efficiently redistributed to the newly added disks without interruption to the activity of the CM server. Continuous media service providers, such as video-on-demand services, cannot afford to or may not be willing to stop services to its customers in order to add, remove, or upgrade the CM server disks. In addition to minimizing downtime, a CM server cannot be subjected to an inefficient method for disk scaling during uptime since this might affect its services. A CM service provider needs to deliver high-quality, uninterrupted service even during maintenance periods. This motivates the need for an efficient, online method to scale disks in a continuous media server.

Due to the large size and bandwidth requirements of CM objects and CM servers' large number of simultaneous users, a CM object is broken into fixed sized blocks which are distributed across all the disks. Several popular place-

*This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC) and ITR-0082826, NIH-NLM grant nr. R01-LM07061, NASA/JPL contract nr. 961518, DARPA and USAF under agreement nr. F30602-99-1-0524, and unrestricted cash/equipment gifts from NCR, IBM, Intel and SUN.

ment techniques include round-robin striping (e.g., [2]), RAID striping (e.g., [17]), and hybrid (e.g., [8, 16]). All these techniques can be categorized as *constrained placement* approaches where the location of a block is fixed and determined by the placement algorithm. However, an alternative placement approach is the *randomized placement* of blocks to disks. For example, the RIO (Randomized I/O) project at UCLA has demonstrated the many advantages of a randomized data placement technique when performing data accesses [14]. These advantages include: 1) load balancing by the law of large numbers, 2) no need for synchronous access cycles, 3) single traffic pattern, and 4) support for unpredictable access patterns as generated, for example, by interactive applications or VCR-type operations on CM streams.

In this paper, we also assume the random placement technique. However, one disadvantage with random placement is that some sort of a directory system is required for the media retrieval to keep track of the location of every block. To address this shortcoming, we propose a pseudo-random placement where we can always regenerate the random sequence for object retrieval using a standard pseudo-random number generator and the same seed.

Redistributing blocks placed in a random manner requires much less overhead when compared to redistributing blocks placed using a constrained placement technique. For example, with round-robin striping, when adding or removing a disk, almost all the data blocks need to be moved to another disk. Instead, with a randomized placement, only a fraction of all data blocks need to be relocated. That is, only enough data blocks are moved to fill an appropriate fraction of the new disks. For disk removal, trivially, only the data blocks on the removed disk must be moved. Meanwhile, redistribution with random placement must ensure that data blocks are still randomly placed after disk scaling in order to balance the load on the multiple disks.

With pseudo-random placement, for each block a random number X is generated and the block is placed on disk ($X \bmod N$), where N is the total number of disks. When adding or removing disks, one approach is to simply redistribute each block to disk ($X \bmod N_j$), where N_j is the new number of disks. The main disadvantage of this approach is that potentially all the blocks may need to be redistributed to new locations.

We propose an approach termed SCADDAR: SCAling Disks for Data Arranged Randomly. With SCADDAR, we are able to use pseudo-random placement without redistributing all the blocks after each scaling operation. Besides minimizing block movement, SCADDAR does not require a directory for storing block locations, only a storage structure for recording scaling operations, which is significantly less than the number of all block locations. In addition, SCADDAR computes the new locations of blocks on-the-fly for each block access by using a series of inexpensive **mod** and **div** functions. SCADDAR also maintains randomized block placement for successive scaling operations which, in turn, preserves load balancing of the disks.

The remainder of this paper is organized as follows. Section 2 reviews some of the related work. In Section 3, we provide a formal definition of the problem along with several initial approaches. Section 4 describes the Bounded and Randomized SCADDAR algorithms. Section 5 contains our performance evaluation. Finally, Section 6 concludes this paper and presents the future direction of this work.

2. Related work

Several studies have focused on data placement and retrieval scheduling of continuous media objects, for collections see [9, 13, 6, 10, 4, 5]. Among these, only a few studied the random placement of data blocks on continuous media servers [3, 12, 11, 14]. Traditional constrained placement techniques such as round-robin data placement allow for deterministic service guarantees while random placement techniques are modeled statistically. Overall, random placement increases the flexibility to support various applications while it maintains a competitive performance [15]. We assume a slight variation of random placement, pseudo-random placement, in order to locate the residence of a block at the retrieval time, without the overhead of maintaining a directory. This is achieved by exploiting the fact that we can regenerate the sequence of numbers generated via a pseudo-random generator function.

One study has addressed the redistribution of data blocks with round-robin data striping [7]. Inherently such a technique requires that almost all the blocks be relocated. The overhead of such block movement (bandwidth is consumed on both the source and the target disk drives) may be amortized over a certain amount of time but is, nevertheless, significant and wasteful.

Work has also been done to write continuous media data to a server [1]. This work is orthogonal to our approach since we also need a similar technique to write blocks during the redistribution.

To our knowledge, no previous work has identified and quantified the benefits of randomly placing data blocks on a scalable continuous media server when block redistribution is desired or required. Additionally, no prior work has addressed such redistribution while the CM server is online.

3. Problem statement

Pseudo-random placement We use pseudo-randomized placement of CM object blocks so that a block has roughly equal probabilities of residing on each disk. The quality of our pseudo-random number generator is assumed to be similar to that of pure randomized algorithms. Continuous media objects are split into fixed size blocks and distributed over a group of homogeneous disks such that each disk carries an approximately equal load. With pseudo-random distribution, blocks are placed onto disks in a random, but reproducible, sequence. We will show in Section 5 that load balancing is achieved through a uniform *and* random distribution.

Definition 3.1: Pseudo-random placement of CM object blocks is defined as a random placement of blocks whose random sequence can be reproduced. ∎

Definition 3.2: Given a particular block, X_0 is defined as the random number, with range $0 \ldots R$, generated by a pseudo-random number generator for this block before any scaling operations. ∎

The disk number, D_0, in which a block resides is defined as:

$$D_0 = (X_0 \bmod N_0) \tag{1}$$

where N_0 is the total number of disks.

To compute the disk number for any block, we obtain a new X_0 by calling the pseudo-random number function, $p\_r(s)$, i times where s is the unique seed of the object and i refers to the i^{th} block. The function, $p\_r(s)$, is assumed to return a random number in the range of $0 \ldots R$ where R is $2^b - 1$ and $p\_r(s)$ returns a b-bit random number. $p\_r(s)$ will produce an identical random sequence given a unique seed s. Table 1 lists all the parameters and their definitions used throughout this paper.

Scaling operation We use the notion of *disk group* as a group of k disks that is added or removed during a scaling operation. Without loss of generality, a scaling operation on a CM server with N disks either adds or removes one disk group. The initial number of disks in a CM server is denoted as N_0 and, subsequently, the number of disks after j scaling operations will be denoted as N_j.

During scaling operation j, a *redistribution function* $RF()$ redistributes the blocks residing on N_{j-1} to N_j disks. Consequently, after scaling operation j, a new *access function* $AF()$ is needed to identify the location of a block, since its location might have been changed due to the scaling operation.

Scaling up will increase the total number of disks and will require an $(N_j - N_{j-1})/N_j$ 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 moved and randomly distributed across remaining disks to maintain load balancing. (The number of block movements just described is the theoretical minimum needed to maintain an even load.) The original seed used to reproduce the sequence of disk locations can no longer be used to reproduce the blocks' new sequence. The problem is how to derive a new sequence using a simple computation with the least possible movement of blocks and the same seed no matter how many operations are performed, all while maintaining load balancing and a random distribution of data blocks.

We can formally state the redistribution problem as:

Problem 1: Given j scaling operations on N_0 disks, find $RF()$ such that:

- [RO1:] Block movement is minimized during redistribution. Only $z_j \times B$ blocks should be moved, where:

$$z_j \approx \frac{|N_j - N_{j-1}|}{\max(N_j, N_{j-1})} \tag{2}$$

  and B is the total number of object blocks.

- [RO2:] Randomization of all object blocks is maintained. Randomization leads to load balancing of all blocks across all disks where $\mathbf{E}[n_0] \approx \mathbf{E}[n_1] \approx \mathbf{E}[n_2] \approx \ldots \approx \mathbf{E}[n_{N_j-1}]$. $\mathbf{E}[n_k]$ is the expected number of blocks on disk k.

Problem 2: Find the corresponding $AF()$ such that:

- [AO1:] CPU and disk I/O overhead are minimized using a low complexity function to compute a block location.

Initial approaches We have considered several initial approaches for solving block redistribution such as a directory storage structure, complete redistribution after each scaling operation, and several hashing techniques.

This problem could be considered as a simple bookkeeping problem where $RF()$'s keep track of the location of every block of every CM object after scaling operations. Hence, the access function, $AF()$, is simply a directory lookup where a directory entry contains the disk location of that directory index (block number). For each scaling operation, new random locations must be determined per block similarly to how they are determined in SCADDAR to minimize block movement. However, instead of using a new function in SCADDAR to *generate* these new random locations, all the changed locations would need to be individually updated in the directory for later block retrieval. Adding and deleting CM objects will also cause directory updates, in addition to the directory updates needed after scaling operations. Hence, $AF()$'s and $RF()$'s become functions of the number of blocks and the number of objects in the CM server. Considering the fact that a typical CM

Table 1. List of parameters used repeatedly in this study and their respective definitions.

Term	Definition
R	$2^b - 1$ where b is the bit length of $p\_r(s)$'s return value
s	Seed used by $p\_r()$ to retrive block locations of an object
$p\_r(s)$	Function that returns a unique random sequence for each unique seed s. Each iteration returns the next element, in the range of $0 \ldots R$, of the sequence.
N_0	Initial number of disks before any scaling operations
X_0	i^{th} iteration of $p\_r(s)$ for the i^{th} block (i is ignored in our notations)
D_0	Disk on which a block of object m resides. $D_0 = X_0 \bmod N_0$
REMAP_j	Function that remaps X_{j-1} to X_j, where $\text{REMAP}_0 = p\_r(s)$
N_j	Total number of disks after j scaling operations
X_j	Derived from a series of REMAP functions, $\text{REMAP}_0 \ldots \text{REMAP}_j$
D_j	Disk on which a block of an object resides, after j scaling operations. $D_j = X_j \bmod N_j$

server can contain in the order of thousands of CM objects and each CM object contains tens of thousands of blocks, the directory can potentially expand to millions of entries. Moreover, using a centralized directory, where one node of a multi-noded server architecture holds the directory, could render it as a potential bottleneck since multiple directory accesses and updates require concurrency control. Instead, using a distributed directory where each node in a multi-noded server holds a copy of the directory, would require integrity checks to ensure that all directories are consistent. Therefore, we are in need of an $RF()$ and $AF()$ that are independent from the number of CM objects and that use data structures that need updating only when scaling operations occur (which are assumed to be less frequent events than adding or removing CM objects).

An alternative initial approach is for complete reorganization using $RF()$ = $AF()$ = (X_0 mod N_j) after every operation j. Here, a new initial state arises after every scaling operation and only the object's seed needs to be stored just as before any scaling operation occurs. However, this might require all blocks to be moved to a new location, thus violating RO1 which is to minimize block movement.

The method for finding a block location prior to the occurrence of any scaling operations is similar to using a hash function X mod N. We wish to extend this idea so that rehashing occurs at every scaling operation, or *overflow event* in hashing. Disks can be treated as hash buckets with one difference. Hashing attempts to minimize the number of bucket accesses whereas the number of disk accesses need not be minimized during overflow events because disk accesses can occur simultaneously. Therefore, hashing techniques do not take advantage of multiple bucket splits since they minimize the bucket accesses and split only one bucket during overflow. Extendible hashing seems to be a viable approach that uses hashing and a directory structure. Here, blocks are assigned to a specific directory entry which points to a bucket (or a disk in our case). Each directory entry is labelled with a d-bit binary number. Rehashing occurs when disks overflow and new disks must be added. Disk additions periodically cause the directory size to double since each entry now has a $(d+1)$-bit binary number label, which is required for hashing. At first, extendible hashing seems very analogous to reorganizing blocks on disks with buckets representing disks. The important difference is that in order to ensure load balancing and randomized placement of blocks, each directory entry can only point to one disk since each entry has equal probability of being hashed into. To accomplish this, the number of disks must be doubled during addition operations or halved during removal operations which is not a feasible or flexible solution. Linear hashing presents a similar problem for our application. Linear hashing does not guarantee that buckets will have approximately equal number of records (or data blocks) at all times since the bucket being split is not necessarily the overflowed bucket. Since buckets splits are performed on a round-robin basis, load balancing could be achieved after each round of splits. We need a method to guarantee a balanced load after *every* bucket overflow.

Although the above approaches do not offer complete solutions, useful principles from each can be obtained to formulate a better approach. We realize from the directory scheme that a new random number sequence, X_j, is needed to track new block locations. We also realize, from extendible hashing, that rehashing can be used to remap a random number sequence to a new sequence. We wish to devise a scheme to remap the random numbers, X_0, generated for each block to a new sequence of random number, X_j, which will indicate where a block should reside after the j^{th} scaling operation. We restate the fact that the block location is derived from D_j, similar to Eq. 1, which could either indicate a new location for a block or the previous location of that block. If a new sequence of X_j's can be found for each scaling operation, then we can simply compute D_j to find the block location after the j^{th} operation. Therefore, $AF()$ and $RF()$ need to compute the new X_j random numbers for every block while maintaining the objectives of RO1, RO2 and AO1. While none of these approaches satisfies all these objectives, we use specific principles from each one and introduce our approach for finding X_j, a remapping of X_0 for each block, called SCADDAR.

4. SCADDAR: SCAling Disks for Data Arranged Randomly

The main idea behind SCADDAR is that the random numbers used to determine the location of each block are to be mapped into a new set of random numbers (one for each block) so that these new numbers can be used to determine the block locations after a scaling operation. Although the initial set of random numbers are generated from the pseudo-random generator, successive sets of numbers are generated using SCADDAR. Because every block could either move off of or remain on a disk, the new set of random numbers should reflect the accurate location.

Every block always has a random number, X_j, associated with it. After a scaling operation, each block should have a new random number. From this point on, we will speak in terms of finding a random number, X_j, for each block after a certain scaling operation since the disk location of the block can be easily found by computing D_j, similar to Eq. 1, from X_j. New random numbers are found using a function, REMAP_j, which takes X_{j-1} as input and generates X_j after scaling operation from $j-1$ to j. Note that REMAP_0 is the original pseudo-random generator function.

REMAP functions are used within both the $RF()$ and $AF()$ functions. In particular, during scaling operation j:

- if disks are added, then $RF()$ applies a sequence of REMAP functions (from REMAP_0 to REMAP_j) to compute X_j for every block on *all* the disks. This should result in a random selection of blocks to be redistributed to the added disks.

- if disks are removed, then $RF()$ applies a sequence of REMAP functions (from REMAP_0 to REMAP_j) to compute X_j of every block residing *only* on those removed disks. This should result in a random redistribution of blocks from the removed disks.

Similarly, after scaling operation j, to find the location of block i, $AF()$ applies a sequence of REMAP functions (from REMAP_0 to REMAP_j) to compute X_j. Subsequently, $RF()$ and $AF()$ compute the location of a block from its random number, X_j, similar to Eq. 1. That is, the sequence $X_0, X_1, \ldots, X_j$ can be used to determine the location of a block after each scaling operation.

Now, we can restate the objectives of Section 3: RO1, RO2 and AO1 for SCADDAR as follows. The REMAP functions should be designed such that:

RO1: $(X_{j-1} \bmod N_{j-1})$ and $(X_j \bmod N_j)$ result in different disk numbers only for z_j (see Eq. 2 in RO1) blocks and not more.

RO2: For those X_j's that $D_{j-1} \neq D_j$, there should be an equal probability that D_j is any of the newly added disks (in case of addition operations), or any of the non-removed disks (in case of removal operations).

AO1: The sequence $X_0, X_1, \ldots, X_j$, and hence D_j can be generated with a low complexity.

The design of the REMAP function hence becomes a challenging problem. For the remainder of this section we first explain our initial, naive approach on the design of REMAP in Section 4.1 and show that while it satisfies RO1 and AO1, it fails to satisfy RO2 after more than one scaling operation. Subsequently in Section 4.2, we discuss a bounded approach which satisfies all the objectives for up to k number of scaling operations. The upper-bound for k is computed in Appendix A proving that RO2 will again be violated after k scaling operations. In this case, we suggest a redistribution of all the blocks. Finally in Section 4.3, we give a superior, randomized approach which satisfies all of our objectives. The effectiveness of this approach is shown through experimentation and simulation in Section 5.

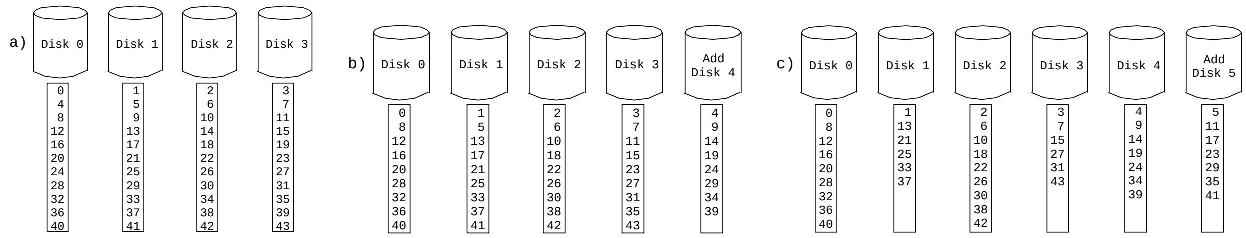
4.1. A naive approach

We first describe a simple scheme that allows scaling operations to be performed. This scheme simply applies a $\bmod N_j$ function, where N_j is the number of disks after scaling operation j, to all the random numbers. However, it is different from a complete redistribution by enforcing a block movement only if the resulting disk number is one of the new disks.

This algorithm can be expressed as follows for the j^{th} scaling operation, where D_j gives the disk number:

$$\mathrm{D}_j = \begin{cases} X_0 \bmod N_j & \text{if } X_0 \bmod N_j \geq N_{j-1} \\ \mathrm{D}_{j-1} & \text{otherwise} \end{cases} \qquad (3)$$

We illustrate an add operation through an example. Fig. 1a shows an initial group of 4 disks using $X_0 \bmod 4$ which makes the blocks appear to be placed in a round-robin fashion. However, the numbers below the disks are actually the random numbers, X_0, generated by the pseudo-random number generator and **not** the actual block numbers (the ordering shown below each disk is not significant). In Fig. 1b, when the next scaling operation is an addition of one disk, then 1/5 of all blocks, where $(X_0 \bmod 5) \geq 4$, are moved onto the new disk, Disk 4. However, with another operation of adding one more disk using the same approach, 1/6 of all the blocks are not evenly picked from the disks and moved onto the new disk, Disk 5. Fig. 1c shows that only certain blocks from disks 1, 3 and 4 are moved onto disk 5 while disk 0 and disk 2 are ignored. The reasoning here is because disk 5 will contain blocks with random numbers that satisfy $X_0 \bmod 6 = 5$ which are all odd numbers. Disks 0 and 2 resulting from $X_0 \bmod 4 = 0$ and



a. Initial State b. 1^{st} 1-disk add operation c. 2^{nd} 1-disk add operation

Figure 1. Initial, naive approach. The load is not balanced starting from the 2^{nd} operation.

$X_0 \bmod 4 = 2$, respectively, are all even numbers. Therefore, because disk 5 contains only odd numbers, blocks from disks 0 and 2 do not qualify and are not moved.

Even though RO1 and AO1 are upheld, this does not guarantee load balancing of blocks since the initial equally distributed load deteriorates after subsequent operations. The reason is that we are using the same random number, X_0, in the range of $0 \ldots 2^b - 1$, where b is 32 bits, for finding D_j. We need to choose from a new set of random numbers during each successive scaling operation to guarantee random placement. The same results are observed during disk removal operations.

4.2. Bounded SCADDAR approach

Bounded SCADDAR extends the naive approach by allowing successive scaling operations to be performed while also maintaining RO2. We now explain Bounded SCADDAR in greater detail.

First, we show REMAP_j for deriving X_j after a disk group removal during the j^{th} operation because it is simpler than REMAP_j for removals. Next, REMAP_j for deriving X_j after a disk group addition during the j^{th} operation is shown. In each case, X_j results after remapping X_{j-1}. We restate that the initial random number of any block before any scaling operations is $\text{REMAP}_0 = X_0 = p\_r(s)$.

To simplify our notations,we use Def. 4.1 as the underlying basis of reasoning for computing REMAP_j.

Definition 4.1 : Let $q_j = (X_j \text{ div } N_j)$ and $r_j = (X_j \bmod N_j)$, i.e., $X_j = q_j \times N_j + r_j$. ∎

4.2.1 Block location after disk removal

For this section, we focus on disk removals as the scaling operation. Fig. 1 depicts that using the same range of random numbers, $0 \ldots R$, to generate D_j may not lead to a random distribution for all blocks, thus violating RO2. All blocks on a removed disk need to be reassigned to the remaining disks, according to RO1. Now that a disk is removed, REMAP_j should return X_j, using X_{j-1}. In order to avoid the problems of the naive approach, X_j should have a different source of randomness from X_{j-1} so REMAP_j uses $(X_{j-1} \text{ div } N_{j-1})$ as a new source of randomness even though this is a smaller range. The shrinking range leads us to believe there exists a threshold for the maximum number of performable scaling operations as confirmed in Appendix A. Using q_{j-1}, the range shrinks to $0 \ldots \lfloor R/N_{j-1} \rfloor$. Eq. 4 defines REMAP_j if scaling operation j is a removal of disks, where the new() function maps from the previous disk numbers to the new disk numbers taking into account gaps that might occur from disk removals:

$$\text{REMAP}_j = X_j = \begin{cases} q_{j-1} \times N_j + \text{new}(r_{j-1}) \\ \quad \text{if } r_{j-1} \text{ is not removed (case a)} \\ q_{j-1} \\ \quad \text{otherwise (case b)} \end{cases} \tag{4}$$

We wish to construct X_j to contain two retrievable pieces of information: 1) a new source of randomness used for future operations, and 2) the disk location of the block after the j^{th} operation. First, the new source of randomness (q_{j-1}, from the above paragraph) is packaged as the quotient of X_j. Second, the new disk location is determined by applying D_j, similar to Eq. 1, to this new source of randomness. In Eq. 4b, X_j is set only to q_{j-1} since we need a new location for the block and applying D_j to q_{j-1} achieves this. In Eq. 4a, the block remains so we want to construct X_j using the block's current disk location (as the remainder) as well as a new source of randomness (as the quotient) in case of future scaling operations.

4.2.2 Block location after disk addition

Now consider that disks are added during operation j. Here a certain percentage of blocks need to be moved and are chosen at random depending on how many disks were added (RO1). Again, we need to use a new range of random numbers upon each add operation to avoid the non-random distribution problem of the naive approach. We use $(q_{j-1} \text{ div } N_j)$ as the new source of randomness which, again, has the shrinking range problem. The task is to derive a $\textsc{Remap}_j$ function in order to find D_j. Eq. 5 defines $\textsc{Remap}_j$ if scaling operation j is an addition of disks:

$$\textsc{Remap}_j = X_j = \begin{cases} (q_{j-1} \text{ div } N_j) \times N_j + r_{j-1} \\ \quad \text{if } (q_{j-1} \text{ mod } N_j) < N_{j-1} \text{ (a)} \\ (q_{j-1} \text{ div } N_j) \times N_j + (q_{j-1} \text{ mod } N_j) \\ \quad \text{otherwise (b)} \end{cases} \tag{5}$$

Similarly to Eq. 4, we will construct X_j to contain the new source of randomness (as the quotient) and the disk location of the block (as the remainder). We use $(q_{j-1} \text{ div } N_j)$ as the quotient for future operations to maintain RO2.

To decide whether a block moves to an added disk or stays, we use $(q_{j-1} \text{ mod } N_j)$. Eq. 5a states the block remains while Eq. 5b states the block will move to a newly added disk. To uphold RO1 we want to move blocks only if they are randomly selected. In other words, if $(q_{j-1} \text{ mod } N_j) \geq N_{j-1}$ for a particular block then that block is moved to an added disk during operation j. The target disk is packaged as the remainder of X_j which we can extract using Eq. 1.

We further simplify $(q_{j-1} \text{ div } N_j) \times N_j + (q_{j-1} \text{ mod } N_j)$ to $(q_{j-1} - (q_{j-1} \text{ mod } N_j)) + (q_{j-1} \text{ mod } N_j)$ and finally to q_{j-1}, resulting in Eq. 6:

$$\textsc{Remap}_j = X_j = \begin{cases} (q_{j-1} - q_{j-1} \text{ mod } N_j) + r_{j-1} \\ \quad \text{if } (q_{j-1} \text{ mod } N_j) < N_{j-1} \text{ (a)} \\ q_{j-1} \\ \quad \text{otherwise (b)} \end{cases} \tag{6}$$

All the objectives of $RF()$ and $AF()$ are met using Bounded SCADDAR approach. RO1 is satisfied since only those blocks which need to be moved are moved. Blocks either move onto an added disk or off of a removed disk. RO2 is satisfied since $\textsc{Remap}_j$ always uses a new source of randomness to compute X_j. AO1 is satisfied since block accesses only require one disk access per block. Block location is determined through computation using inexpensive **mod** and **div** functions instead of disk-resident directories. However, the shrinking random number range bounds the number of scaling operations as shown in Appendix A. The following section describes Randomized SCADDAR which satisfies all of our objectives and allows us to perform scaling operations without bound.

4.3. Randomized SCADDAR approach

Randomized SCADDAR allows an unlimited amount of scaling while still satisfying our requirement of random redistribution and load balancing of blocks across disks. We assume that performing repeated **mods** and pseudo-random function calls is negligible, so a large number of scaling operations can be supported. The difference between Randomized SCADDAR and Bounded SCADDAR is our choice of the new source of randomness after each scaling operation. By using the quotient, q_{j-1}, as the source of randomness for scaling operation j, X_j would eventually shrink to a constant as j increases.

We decide to use $p\_r(X_{j-1})$ as our new source of randomness to guarantee a 32-bit number to be used as the quotient for X_j. We are assuming that the seed used for $p\_r()$ and the random number returned are independent for practical purposes. Basically, we are using the previous X_{j-1} as the seed of the pseudo-random generator number to obtain a fresh X_j. $\textsc{Remap}_j$ for Randomized SCADDAR is constructed in a similar fashion as in Bounded SCADDAR, except $p\_r(X_{j-1})$ is used as the quotient.

Eqs. 7 and 8 define $\textsc{Remap}_j$ for a removal of disks and an addition of disks, respectively:

$$\textsc{Remap}_j = X_j = \begin{cases} p\_r(X_{j-1}) \times N_j + \text{new}(r_{j-1}) \\ \quad \text{if } r_{j-1} \text{ is not removed (a)} \\ p\_r(X_{j-1}) \\ \quad \text{otherwise (b)} \end{cases} \tag{7}$$

$$\textsc{Remap}_j = X_j = \begin{cases} p\_r(X_{j-1}) \times N_j + r_{j-1} \\ \quad \text{if } (p\_r(X_{j-1}) \text{ mod } N_j) < N_{j-1} \text{ (a)} \\ p\_r(X_{j-1}) \times N_j + \\ \quad (p\_r(X_{j-1}) \text{ mod } N_j) \\ \quad \text{otherwise (b)} \end{cases} \tag{8}$$

Let X_j denote the random number for a block after j scaling operations. We say that a pseudo-random number generator is perfect if all the X_j's are independent and they are all uniformly distributed between 0 and R. Given a perfect pseudo-random number generator, Randomized SCADDAR is statistically indistinguishable from complete reorganization[1]. This fact can be easily proven by demonstrating a coupling between the two schemes; due to lack of space, we omit the proofs from this paper version.

However, real life pseudo-random number generators are unlikely to be perfect. Hence, we cannot formally analyze the properties of Randomized SCADDAR. Therefore, in the next section, we will use simulation to verify that Randomized SCADDAR satisfies RO1, RO2, and AO1 for a large number of scaling operations. It is interesting to note the close similarity between Randomized SCADDAR and complete reorganization.

[1]This does not mean that Randomized SCADDAR and complete reorganization will always give identical results; the two processes are identical in *distribution*.

5. Performance evaluation

We can achieve our goal of a load balanced storage system where similar loads are imposed on all the disks on two conditions: *uniform* distribution of blocks and *random* placement of these blocks. A uniform distribution of blocks on disks means that all disks contain the same (or similar) number of blocks. So, we need a metric to compare Bounded and Randomized SCADDAR with complete reorganization and show that all techniques achieve a balanced load. We would like to use the uniformity of the distribution as a metric, hence, the standard deviation of the number of blocks across disks is a suitable choice. However, because the averages of blocks per disk will differ when scaling disks, we normalize the standard deviation and use the coefficient of variation (standard deviation divided by average) instead. One may argue that a uniform distribution may not necessarily result in a *random* distribution since, given 100 blocks, the first 10 blocks may reside on the first disk, the second 10 blocks may reside on the second disk, and so on. However, as discussed in Section 4.3, the distribution of blocks resulting from Randomized SCADDAR can be coupled to that of complete reorganization. Thus, Randomized SCADDAR does result in a satisfactory random distribution. For the rest of this section, we use the uniformity of block distribution as an indication of load balancing.

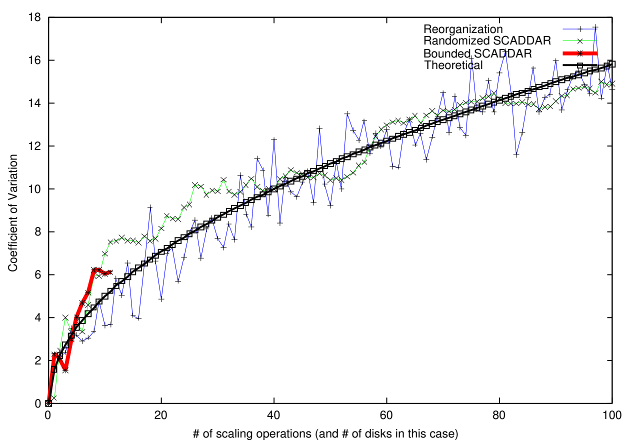


Figure 2. Scaling to 100 disks using: 1) complete reorganization after each scale operation, 2) Randomized SCADDAR, 3) Bounded SCADDAR, and 4) theoretical computation using Def. 5.1.

For each set of experiments, we performed 100 disk scaling operations and used 4000 data blocks. Fig. 2 shows the coefficient of variations of four methods when adding disks one-by-one starting with one disk and ending with 100 disks. The first curve shows complete block reorganization after adding each disk. Although complete reorganization requires a great amount of block movement, the uniform distribution of the blocks is ideal. The second curve shows Randomized SCADDAR. This follows the trend of the first curve suggesting that the uniform distribution of blocks is maintained at a near-ideal level while minimizing block movement. The third curve shows Bounded SCADDAR. The limitations of this algorithm are clear since no more than 11 addition operations can be performed even though block movement is minimized and a uniform distribution is maintained across these 11 operations. This is due to the shrinking range of random numbers after each operation. Beyond the 11^{th} operation, block movement actually ceases and the added disks remain empty. The fourth curve shows the theoretical coefficient of variation as defined in Def. 5.1. The theoretical coefficient of variation is derived from the theoretical standard deviation of Bernoulli trials. The Randomized SCADDAR curve also follows a similar trend as the theoretical curve.

Definition 5.1: Theoretical coefficient of variation is defined as $\sqrt{\frac{D-1}{B}} \times 100.$ ∎

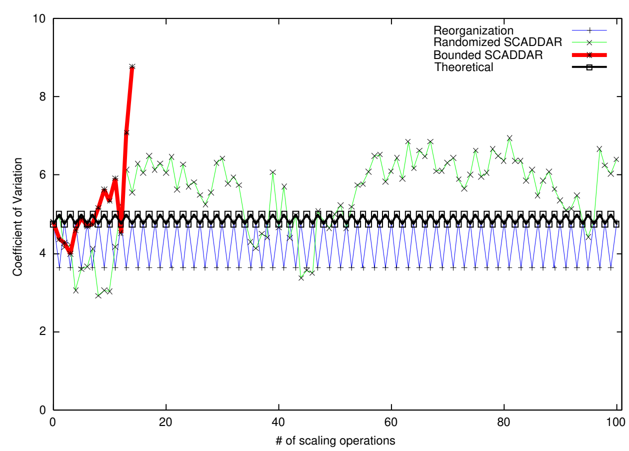


Figure 3. Alternately adding 1 disk and removing 1 disk using: 1) complete reorganization after each scale operation, 2) Randomized SCADDAR, 3) Bounded SCADDAR, and 4) theoretical computation using Def. 5.1. (10 initial disks)

As a comparison of the stability of the uniform distribution of data blocks, we show the coefficient of variation in Fig. 3 using a steady average number of disks upon successive scaling operations. In other words, we begin with 10 initial disks and perform an add disk, a delete disk, an add disk, etc. for 100 total operations (50 adds, 50 deletes). We observe that the Randomized SCADDAR curve follows roughly the same trend as the theoretical curve. Also, as expected, the Bounded SCADDAR curve does not scale beyond 14 operations. From the rule-of-thumb equation at the end of Appendix A, we find $k \leq 7$ where $\epsilon \leq 5\%$, $\mu_k = 10$ and $b = 32$. Using simulation, we observe an ability to support more than 14 operations but we believe the uniform distribution of blocks begins to degrade after 7 operations.

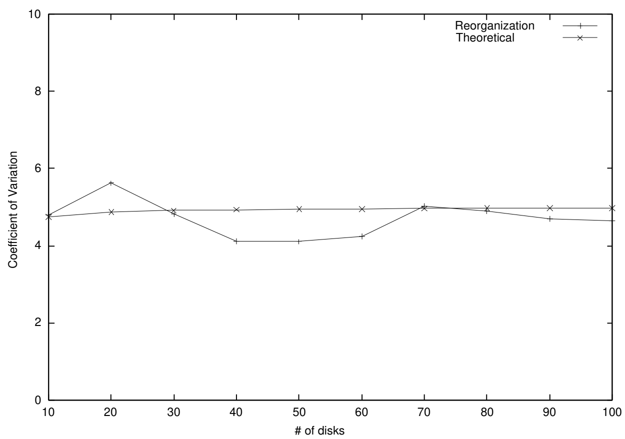


Figure 4. Increasing the number of blocks proportionally to the number of disks starting with 4000 blocks and 10 disks using: 1) complete reorganization and 2) theoretical computation using Def. 5.1.

Finally, we observe that the coefficient of variation seems to increase without bound as the number of disks increases due to successive add operations in Fig. 2. Intuitively, for a fixed number of blocks and an increasing number of disks, coefficient of variation becomes more sensitive to block fluctuation since the average number of blocks per disk is lower. This point is shown by increasing the number of blocks proportionally with the number of disks. Fig. 4 shows the coefficient of variation starting with 10 disks and 4000 blocks and ending with 100 disks and 40000 blocks. We show the uniformity of the distribution when performing complete reorganization of blocks as both the number of blocks and disks are increased. As expected, the coefficient of variation is fairly level and follows the theoretical coefficient of variation as defined in Def. 5.1. We assume Randomized SCADDAR to behave similarly as discussed in Section 4.3.

6. Conclusions and future research directions

The Randomized SCADDAR approach meets all the objectives required in redistributing randomly placed blocks. The objectives of minimizing block movement, maintaining randomness, and few disks accesses holds true for up to k scaling operations for Bounded SCADDAR, where the upper-bound of k can be computed. However, using Randomized SCADDAR, we show (through simulation) that there is no upper-bound for the number of supportable disk scaling operations. The quality of random block distribution and minimization of block movement are both maintained, resulting in load balancing and low disk access.

We would like to apply SCADDAR to redistribute randomly placed blocks on hetergeneous disk arrays. Currently, SCADDAR is applicable to both homogeneous physical disks and logical disks. By applying previous work of mapping heterogeneous physical disks to homogeneous logical disks [18], SCADDAR may naturally evolve to allow block redistribution on hetergeneous physical disks. As for fault tolerance, data mirroring may be a simple solution with SCADDAR. Mirrored blocks could be placed at a fixed offset determined by a function f(N_j). For example, f(N_j) could return $N_j/2$ as an offset. We also plan to investigate using data parity bits to handle faults with less required storage space. SCADDAR can also be implemented as an online disk scaling technique where scalings can be performed during a normal mode of operation. One solution is to perform SCADDAR with block copying instead of block moving. After all blocks have been copied, the system would instantly switch from using the previous set of disks to using the new set. The unused blocks can then be deleted during idle times.

We are currently implementing SCADDAR on our own continuous media server, *Yima*, to test the practicality of its load balancing, low-complexity, and preservation of block randomness.

7. Acknowledgements

The authors would like to thank Rolfe Schmidt and Hong Zhu for their assistance in determining the theoretical coefficient of variation and their suggestions on particular pseudo-random number generators.

References

[1] W. Aref, I. Kamel, Niranjan, and S. Ghandeharizadeh. Disk Scheduling for Displaying and Recording Video in Non-Linear News Editing Systems. In *Multimedia Computing and Networking Conference*, February 1997.

[2] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. Staggered Striping in Multimedia Information Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[3] S. Berson, R. R. Muntz, and W. R. Wong. Randomized Data Allocation for Real-Time Disk I/O. In *COMPCON*, pages 286–290, 1996.

[4] J. Buford, editor. *Multimedia Systems*. Addison-Wesley, 1994.

[5] S. Chung, editor. *Multimedia Information Storage and Management*. Kluwer Academic Publishers, 1996.

[6] A. Ghafoor. Special Issue on Multimedia Database Systems. *ACM Multimedia Systems*, 3(5-6), November 1995.

[7] S. Ghandeharizadeh and D. Kim. On-line Reorganization of Data in Scalable Continuous Media Servers. In 7^{th} *International Conference and Workshop on Database and Expert Systems Applications (DEXA'96)*, September 1996.
[8] S. Ghandeharizadeh and C. Shahabi. Management of Physical Replicas in Parallel Multimedia Information Systems. In *Proceedings of the Foundations of Data Organization and Algorithms (FODO) Conference*, October 1993.
[9] S. Ghandeharizadeh and C. Shahabi. Distributed Multimedia Systems. In J. G. Webster, editor, *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley and Sons Ltd., New York, 1999.
[10] W. Grosky, R. Jain, and R. Mehrotra, editors. *The Handbook of Multimedia Information Management*. Prentice-Hall, 1997.
[11] S. H. Kim and S. Ghandeharizadeh. Design of Multi-user Editing Servers for Continuous Media. In *IEEE International Workshop on Research Issues in Data Engineering (RIDE'98)*, Orlando, Florida, February 1998.
[12] R. Muntz, J. Santos, and S. Berson. RIO: A Real-time Multimedia Object Server. In *ACM Sigmetrics Performance Evaluation Review*, volume 25, September 1997.
[13] K. Nwosu, B. Thuraisingham, and P. Berra. Multimedia Database Systems-A New Frontier. *IEEE Multimedia*, 4(3):21–23, Jul.-Sep. 1997.
[14] J. R. Santos and R. R. Muntz. Performance Analysis of the RIO Multimedia Storage System with Heterogeneous Disk Configurations. In *ACM Multimedia*, pages 303–308, 1998.
[15] J. R. Santos, R. R. Muntz, and B. Ribeiro-Neto. Comparing Random Data Allocation and Data Striping in Multimedia Servers. In *SIGMETRICS*, Santa Clara, California, June 17-21 2000.
[16] C. Shahabi, S. Ghandeharizadeh, and S. Chaudhuri. On Scheduling Atomic and Composite Continuous Media Objects. *To appear in IEEE TKDE*, 2001.
[17] F. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming RAID-A Disk Array Management System for Video Files. In *First ACM Conference on Multimedia*, August 1993.
[18] R. Zimmermann and S. Ghandeharizadeh. Continuous Display Using Heterogeneous Disk-Subsystems. In *Proceedings of the Fifth ACM Multimedia Conference*, pages 227–236, Seattle, Washington, November 9-13, 1997.

A. Analysis: Bounding the reduction in randomness

Define the *unfairness coefficient* of a random load distribution scheme as

$$\frac{\text{The largest expected load on any machine}}{\text{The smallest expected load on any machine}} - 1.$$

If we pick an integer x uniformly at random from the range $[0 \ldots R-1]$ and then use $x \bmod N_k$ to compute the disk to which the block must be assigned after k scaling operations, then the unfairness coefficient is given by

$$f(R; N_k) = \frac{1}{R \text{ div } N_k}.$$

We will pretend in this analysis that the pseudo-random number generator in fact generates a truly random number in the range $0 \ldots 2^b - 1$ (i.e. we have b truly random bits).

Let R_i denote the range of the random number space that we have after i operations; further let $\epsilon > 0$ be the largest unfairness coefficient we are willing to tolerate. Then $f(R_k; N_k)$ must be at most ϵ.

Lemma A.1: *R_k div $N_k \geq R_0$ div $(N_0 N_1 N_2 \ldots N_k)$.*

Proof: We first observe that after the i^{th} addition/removal operation, the random number range is at least R_{i-1} div N_{i-1}. Thus, after k operations, the random number range is at least $(((R_0 \text{ div } N_0) \text{ div } N_1) \ldots \text{ div } N_{k-1})$. Thus, $R_k \text{ div } N_k \geq (((R_0 \text{ div } N_0) \text{ div } N_1) \ldots \text{ div } N_k)$. Given any three positive integers x, a, and b, it is easy to verify that $(x \text{ div } a) \text{ div } b = x \text{ div } (ab)$. Hence, $(((R_0 \text{ div } N_0) \text{ div } N_1) \ldots \text{ div } N_k) = R_0 \text{ div } (N_0 N_1 N_2 \ldots N_k)$, and therefore, $R_k \text{ div } N_k \geq R_0 \text{ div } (N_0 N_1 N_2 \ldots N_k)$. ∎

Let π_k represent the product $N_0 N_1 N_2 \ldots N_k$.

Lemma A.2: *If $\pi_k \leq R_0 \cdot (\epsilon/(1+\epsilon))$ then $f(R_k; N_k) < \epsilon$.*

Proof: By Lemma A.1, $R_k \text{ div } N_k \geq R_0 \text{ div } \pi_k$. But $R_0 \text{ div } \pi_k > R_0/\pi_k - 1$. Therefore, $f(R_k; N_k) < 1/(R_0/\pi_k - 1)$. By the precondition in the lemma, $R_0/\pi_k \geq (1+\epsilon)/\epsilon$. This implies that $R_0/\pi_k - 1 \geq 1/\epsilon$. Now, $R_k \text{ div } N_k > 1/\epsilon$, or $f(R_k; N_k) < \epsilon$. ∎

Let μ_k represent the average number of disks during the first k scaling operations. Then $\pi_k \leq \mu_k^{k+1}$ (the geometric mean of a set of positive numbers can never be larger than the arithmetic mean). Hence, the above condition results in the following approximate rule-of-thumb:

> We can continue to use the same random sequence without redistributing the load as long as R_0 is larger than μ_k^{k+1}/ϵ.

Taking logarithm (base two) on both sides, the rule-of-thumb translates into

$$k + 1 \leq (b - \log(1/\epsilon))/(\log \mu_k).$$

For example, if we have an average of sixteen disks, desire $\epsilon \leq 1\%$, and are using a 64-bit random number generator, then $k+1 \leq (64 - \log 100)/4$ i.e. $k+1 \leq 57/4$ i.e. $k \leq 13$. Therefore, a total of 13 disk addition/removal operations can be supported.

The above rule-of-thumb should only be used for obtaining a good *a priori* estimate on how well the system will perform. In an implementation of this scheme, we can keep track of the quantity π_k explicitly and find out whether the next operation will lead to a violation of the precondition in Lemma A.2.