The VLDB Journal manuscript No.
(will be inserted by the editor)

Hash-based labeling techniques for storage scaling*

Shu-Yuen Didi Yao[1], **Cyrus Shahabi**[1], **Per-Åke Larson**[2]

[1] University of Southern California, Computer Science Department, Los Angeles, CA 90089; e-mail: {didiyao,shahabi}@usc.edu
[2] Microsoft Corporation, One Microsoft Way, Redmond, WA 98052; e-mail: palarson@microsoft.com

Received: date / Revised version: date

Abstract Scalable storage architectures allow for the addition or removal of storage devices to increase storage capacity and bandwidth or retire older devices. Assuming random placement of data objects across multiple storage devices of a storage pool, our optimization objective is to redistribute a minimum number of objects after scaling the pool. In addition, a uniform distribution, and hence a balanced load, should be ensured after redistribution. Moreover, the redistributed objects should be retrieved efficiently during the normal mode of operation: in one I/O access and with low complexity computation. To achieve this, we propose an algorithm called Random Disk Labeling (RDL), based on double hashing, where storage can be added or removed without any increase in complexity. We compare RDL with other proposed techniques and demonstrate its effectiveness through experimentation.

Key words Scalable storage systems – Random data placement – Load balancing

1 Introduction

Computer applications typically require ever-increasing storage capacity to meet the demands of their expanding data sets. Examples of these storage applications include file systems, continuous media servers, and Web proxy servers. Because storage requirements often times exhibit varying growth rates, current storage systems may not reserve a great 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.

Our technique to achieve a highly scalable storage system begins by applying a random placement of data objects across a group of storage units (e.g., magnetic disk drives or computer servers). The goal here is that the data object placement allows the storage to be load balanced. Moreover, with a balanced system the aggregate storage capacity and bandwidth can be utilized for concurrently accessing large data objects even after more storage units are added to the system. The challenge is to efficiently redistribute a minimal amount of data to the new storage units and to be able to access them quickly.

1.1 Assumed architecture

In this paper, 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. Given these generalizations, our algorithm can be applied to specific categories which share a common architecture as shown in Figure 1.

The first category is file systems, and more specifically continuous media (CM) servers, where object blocks of large CM file objects (e.g. video or audio objects with 1MB blocks) are striped across the disk array of

* This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0082826 (ITR), IIS-0238560 (CAREER), IIS-0324955 (ITR) and IIS-0307908, 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.

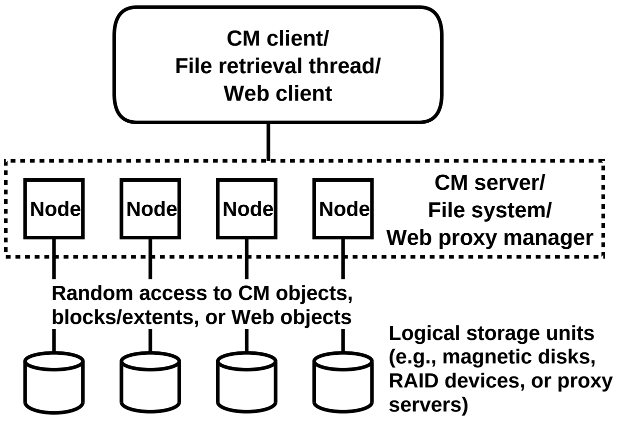

Fig. 1 Common architecture of CM servers, file systems, and Web proxy servers.

a CM server [22]. Blocks are fixed-sized and their striping scheme is based on block-granularity striping. These types of file systems and CM servers are based on shared-nothing database system designs where tuples of relations are declustered across disks [5]. Shared-nothing designs enable parallel reading of large-size relations similar to the large file objects we are dealing with here. These file systems and CM servers also differ from traditional designs such as RAID 3 where individual blocks are declustered with byte-granularity across all disks of the RAID group. Such designs would not scale in throughput when increasing the number of disks [7]. In general, the parity group RAID schemes, such as RAID 3 and 5, do not lend themselves well to scaling since changing the parity group size requires the re-computation of the parity blocks. Moreover, mirroring schemes, such as RAID 1, require twice the amount of storage when scaling. RAID schemes typically are also only applied across a handful of disks. In our scenario, each disk represents a logical storage unit and can potentially be a RAID device itself. The block placement allows load balancing of the CM storage system where the aggregate capacity and bandwidth are achieved when accessing CM files. Disks can be added to the storage system to increase overall capacity or removed due to space conservation or storage reallocation. Note that disk failures, which are masked by fault-tolerance mechanisms such as RAID schemes, differ from disk removals since data can be first moved off a drive before removal. When a disk fails, its data is lost.

The second category is Web proxy servers where Web objects are cached across a group of Web proxy servers. Here, Web objects and Web proxy servers are analogous to data objects and storage units, respectively. As clients make requests for Web objects, their requests may be forwarded to these proxy servers, which may occasionally go off-line. Before these servers go off-line or after they come on-line, their Web objects need to be redistributed to on-line proxy servers to ensure load balancing across proxies and high object availability. This category is also relevant to reverse proxies (i.e., in content distribution networks) where a client calls a reverse proxy server sitting in front of multiple origin servers to retrieve content. The number of origin servers can be scaled-up or down, similar to disks of a storage system, in which case their content needs to be redistributed. This is not the case with forward proxies (i.e., Internet service providers). Forward proxies act as an intermediary between the client and an origin server. Usually the forward proxy also acts as a cache for a pool of clients.

Even though our proposed algorithms are applicable to these specific categories, they are general solutions for placing "data objects" across "storage units". However, for the rest of this paper, we use the application domain of disk storage systems and data files for illustration purposes. Some very large-scale disk storage systems can contain on the order of thousands of disk storage devices [26]. In these systems, data scalability and availability are crucial. So from now on we use the terms "disk" and "file block" (i.e., of a file object) in place of "storage unit" and "data object," respectively.

Note that our use of the term *load balancing* thus far refers to balancing the data across disk drives such that every disk has approximately the same amount of data. However, for our assumed architecture, load balancing based on the storage space also achieves load balancing of data access. Data objects can either be entirely hot or cold, or they can be partially hot or cold. Entirely hot (cold) objects are made up of all hot (cold) blocks, and partially hot (cold) objects have portions of hot (cold) blocks. In all cases, when objects are randomly striped across disks, their blocks are scattered evenly across the disks. Since every disk will contain blocks from every object, the system will exhibit access load balancing across disks when storing a large number of objects (and hence blocks). This means that every disk will contain a similar number of hot and cold blocks. In addition, the storage system may serve a broad range of applications. Access load balancing is also achieved from this mixture of different application access patterns. So, we concentrate on the load balancing of storage space as it also results in the load balancing of data access.

1.2 Problem statement

Pseudo-random placement: File objects are split into fixed-size blocks and distributed over a group of homogeneous disks (i.e., full declustering of objects) such that each disk carries an approximately equal load. Full declustering in shared-nothing database systems, such as striping data across all available disks, has also been shown to achieve high performance [14]. More specifically, we use a *pseudo-random placement* of file object blocks so that a block has roughly equal probabilities of residing on each disk. With pseudo-random distribution, blocks are placed onto disks in a random, but reproducible, sequence. We will show in Section 7 that load balancing is achieved through a uniform distribution.

The placement of a block, i, is determined by its signature X, which is simply an unsigned integer. We use a pseudo-random number generator p_r to compute the signature of a block of an object. p_r is assumed to generate signatures that are practically uniformly distributed between 0 and R. p_r must also produce repeatable sequences for a given seed. One way to derive the seed for block i is from ($\texttt{StrToL}(filename) + i$). This seed is used to initialize the pseudo-random number generator and compute X for block i. Several placement algorithms will be described in subsequent sections.

Scaling operation: Disks can be added to the system to increase overall bandwidth and capacity or removed due space conservation or storage reallocation. We use the notion of *disk group* as a group of n disks that is added or removed. 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, $n/(D+n)$, 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. The number of block movements just described is the minimum needed to maintain an even load. Furthermore, the availability of data can be maintained even during real-time scaling, as described in Section 8.1.

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 each disk is approximately B/D.

Requirement 2 (Minimal Data Movement): During the addition of n disks to a system with D disks storing B blocks, the expected number of blocks to move is $B \times \frac{n}{D+n}$. During the removal of n arbitrary disks, $B \times \frac{n}{D}$ blocks are expected to move.

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 propose Sequential Disk Labeling (SDL) and Random Disk Labeling (RDL), two variations of a hash-based approach, for frequent storage scaling such that the block striping scheme is maintained before and after each scaling operation.

1.3 Organization

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 gives background on two hashing techniques we use to solve our problem. In Section 4, we propose a hash-based solution called SDL. In Section 5, we propose a variation of SDL called RDL that improves the non-uniformity of SDL. Then, in Section 6, we propose a random probing technique that allows the degree of uniformity to be adjusted. Section 7 discusses the performance of our algorithms in terms of uniformity of block distribution and total probes. Section 8 describes some implementation issues. Finally, Section 9 concludes this paper and presents future research directions.

2 Related work

We first present several initial approaches, briefly discussed in [8], for solving block redistribution during storage scaling. Then we discuss related work where scaling techniques are applied to CM servers and proxy servers, shown in Figure 1.

2.1 Initial approaches

A directory structure could be used as a simple bookkeeping solution where block addresses are stored. However, frequent scaling operations would cause frequent updates to a table containing on the order of millions of records. Accessing this directory may also require disk I/O. Also, a distributed server architecture might require distributing the directories resulting in a need for keeping these frequently updated large data structures consistent.

Another approach is a complete redistribution of data blocks after every scaling operation that would result in an even and random distribution. This is clearly unrealistic because of the large amount of block moves.

To analyze the problem of block reorganization during scaling, we draw an analogy to that of hash tables. The goals of a hash table are to evenly distribute *keys* into buckets and to quickly access these keys. These goals are also desirable properties when storing blocks on a set of disks. Moreover, disks can be treated as hash buckets while blocks are similar to keys. Collisions are also similar in that more disks need to be added to handle disk overflows just as the number of buckets need to be increased during bucket overflows.

Several dynamic hashing techniques, such as extendible and linear hashing, could be applied but they both have their drawbacks. With extendible hashing, an *overflow event* causes bucket splits which might double the number of buckets. We cannot restrict scaling operations to double the number of disks for every disk overflow.

Linear hashing [11] does not address load balancing of keys in buckets after bucket splits. In the worst case, linear hashing will result in half of the buckets being full while the other half are half-full. A generalization of linear hashing, called LH* [12], for distributed servers (buckets) also suffers from this load imbalance problem. Moreover, LH* requires at most *three* server visits when inserting keys and at most *four* server visits when searching keys. Our technique for this scenario will achieve load balancing and require at most *one* server visit for key insertion and searching.

In Section 3, we describe two hashing techniques which we later show to be quite adaptable to our problem of block reorganization during storage scaling.

2.2 Continuous media servers

Traditional shared-nothing database system designs have actually incorporated hash-based placement techniques for load balancing. Early examples of these include Teradata [24], Gamma [4], and Bubba [2]. Still, none of these prior systems efficiently redistributed data while scaling. Since then shared-nothing designs have evolved and been adopted by application-specific servers such as CM servers.

CM servers have been the focus of several past studies. Data placement and retrieval scheduling specifically for CM objects are described in [13,22]. One study has addressed the redistribution of data blocks after storage scaling with round-robin data striping [6]. Inherently such a technique requires that almost all blocks be relocated when adding or removing disks. The overhead of such block movement may be amortized over a certain amount of time but it is, nevertheless, significant and wasteful.

Traditional constrained placement techniques such as round-robin data placement allow for deterministic service guarantees while 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 [1,16,20]. However, they did not consider the rearrangement of data due to storage scaling.

In general, random placement increases the flexibility to support various applications while maintaining a competitive performance [21]. We assume a slight variation of random placement, pseudo-random placement, in order to locate a block quickly at retrieval time, without the overhead of maintaining a directory. This is achieved by the fact that we can regenerate the sequence of numbers, each one a block signature, via a pseudo-random generator function when we use the original seed.

We developed a technique called SCADDAR to enable the redistribution of data blocks after storage scaling in a CM server by mapping the block signatures to a new set of signatures resulting in an even, randomized distribution [8]. Equations 1 and 2 map the signature X^i_{j-1} for a block i to a new signature X^i_j after the j-th scale-up or down operation. We omit the i term and simply use X_j for purposes of clarity. Note that the first signature of a block i is still $X_0 = (\texttt{StrToL}(filename) + i)$ as in Section 1.2. Once X_j is computed for a block, the block location is simply disk $d_j = X_j \bmod D_j$.

$$X_j = \begin{cases} p\_r(X_{j-1}) \times D_j + r_{j-1} & \\ \quad \text{if } (p\_r(X_{j-1}) \bmod D_j) < D_{j-1} & \text{(a)} \\ p\_r(X_{j-1}) \times D_j + (p\_r(X_{j-1}) \bmod D_j) & \\ \quad \text{otherwise} & \text{(b)} \end{cases} \tag{1}$$

When disks are added during scaling operation j a certain percentage of blocks are randomly chosen to be moved to the added disks. Equation 1 returns a new signature X_j for a block to determine whether it should move. p_r is a pseudo-random number generator, D_j is the number of disks after operation j, and $r_{j-1} = (X_{j-1} \bmod D_{j-1})$. Equation 1a computes X_j for blocks that are randomly chosen to stay on their current disks and Equation 1b computes X_j for blocks randomly chosen to be moved.

$$X_j = \begin{cases} p\_r(X_{j-1}) \times D_j + \text{new}(r_{j-1}) & \\ \quad \text{if } r_{j-1} \text{ is not removed} & \text{(a)} \\ p\_r(X_{j-1}) & \\ \quad \text{otherwise} & \text{(b)} \end{cases} \tag{2}$$

Equation 2 is used if scaling operation j is a removal of disks. Only those blocks on the removed disks will be randomly distributed to all other disks. Equation 2a computes X_j for a block if it remains on a non-removed disk and Equation 2b computes X_j if it is on a removed disk and needs to be relocated. The new() function maps from the previous disk numbering scheme to the new disk numbering scheme taking into account of gaps that might occur from disk removals.

SCADDAR adheres to the requirements of Section 1.2 except that computation of block locations become incrementally more expensive. Although each step requires only a pseudo-random number function and a "mod" function call, finding a block's location after scaling operation j requires a series of computing X_0 to X_j for that block. In addition, a history log of operations needs to be maintained. In other words, the number of computations is equal to the number of scaling operations. This may be fine for applications with infrequent scaling operations being performed while the server is online. However, extremely frequent operations will eventually cause the performance of block accesses to suffer. The history log can be reset by a complete block reorganization.

2.3 Web proxy servers and peer-to-peer systems

Several past works have considered mapping Web objects to proxy servers using requirements similar to those described in Section 1.2. Below we describe two relevant techniques called Highest Random Weight (HRW) and consistent hashing along with their drawbacks.

HRW is a technique developed to map Web objects to a group of proxy servers [25]. 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. We show in Section 7 that in some cases HRW is several orders of magnitude slower than our proposed 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. In addition, this optimization has the drawbacks of a directory-based approach discussed at the beginning of this section.

Distributed hash tables (DHTs) are used to locate objects in peer-to-peer systems and Web proxy servers. Consistent hashing is a DHT 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 [23]. 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 metadata [3].

A difference between peer-to-peer systems and storage systems is that nodes are expected to be frequently added and removed in peer-to-peer systems whereas scaling operations are expected to be less frequent in disk storage systems. Indeed a peer-to-peer index, such as Chord, can be used for storage systems by mapping disks to nodes and blocks to hashed keys. However, when scaling infrequently with Chord, the system will be in an unbalanced state longer since a newly added disk will only receive blocks from one neighboring disk. Storage load balancing is important for storage systems, and since the scaling operations are infrequent, the time and resources required to load balance after each operation are justified. Although Chord is quick in redistribution after scaling, the long term unbalanced load it imposes degrades the system until many scaling operations later when the system load is balanced again. Thus, Chord works better for load balancing peer-to-peer systems rather than storage systems because of the frequency of node adds and removes. Other DHTs such as CAN [17], Pastry [19], and Tapestry [27] are similar and pose similar problems when scaling is infrequent.

3 Background: double hashing and random probing

In this section, we briefly describe two hashing techniques used as part of our solution: *double hashing* and *random probing.*

Double hashing (i.e., open addressing with double hashing) scans for available buckets when resolving collisions [10]. When inserting key k, double hashing uses two hash functions, $h_1()$ and $h_2()$, which are based on pseudo-random number generator functions, to determine a probe sequence. $h_1(k)$ determines a bucket address in the range $0 \ldots P-1$, where P is the total number of buckets. This is the initial bucket to be probed. If this bucket is full then we need another hash function, $h_2(k)$, to resolve the collision. $h_2(k)$ produces a value in the range $1 \ldots P-1$, which is the number of buckets that are skipped for all subsequent probes. If this value is relatively prime to P, then after P probes, every bucket will be probed exactly once.

Random probing uses an infinite sequence of independent hash functions, which are also based on pseudo-random number generators, to handle collisions [15]. Each hash function calculates an address for a key in the range $0 \ldots P-1$, so the first hash function computes the first random address to be probed. If a collision occurs with the first address then the second hash function computes the second random address to be probed, and so on. The first available address in this sequence becomes that key's address.

In the following sections, we will apply double hashing and random probing to handle collisions for our block placement algorithms. Table 1 summarizes the terms used repeatedly throughout this paper and their respective definitions.

4 Sequential disk labeling (SDL) algorithm

We adapt the double hashing technique to satisfy our problem of efficient redistribution of data blocks during storage 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.

The main difference between a hash table and our address space is the method in which collisions are handled. In double hashing, a collision occurs when a full bucket is probed, resulting in the probing of other buckets until an available bucket is found. With our address space, a collision occurs when an empty slot is probed. When this happens, other slots are probed until a slot with a disk is probed. In both cases, we use the same probing sequence to decide which buckets (disks) to probe.

Table 1 List of terms used repeatedly throughout this paper and their respective definitions.

Term	Definition
X	Signature of any data block
$h_1(k)$, $h_2(k)$	Hash functions with key k
$p\_r1(s)$, $p\_r2(s)$	Pseudo-random number generators with seed s
B	Total number of data blocks
D	Total number of storage units (i.e., magnetic disks or proxy servers)
P	Total number of slots where $D \leq P$ (i.e., size of address space)
n	Size of a disk group to be added or removed
sp	Start position of probe sequence
sl	Step length of probe sequence
$p(\ell)$	ℓ-th slot probed of probe sequence
$d_{i,j}$	Address of data block i after the j-th scaling operation
K	Maximum random probes
E	Expected number of data blocks per disk
x_d	Actual number of data blocks per disk
β	Percentage of confidence
μ	Population mean
$\overline{X}$	Sample mean
ς	Sample standard deviation
n_s	Sample size
$z(\alpha)$	z-score of a particular α value

We now examine what will happen when a disk is added to an empty slot in our address space. When new blocks are added, collisions will not occur here if this slot is probed. However, *every* block that previously collided with this slot must be moved here so that searching for these blocks succeeds. Equivalently with a hash table, collisions would no longer occur on a full bucket if all its keys were suddenly removed. But if this happens, then the keys which previously collided at this bucket need to be moved here from other buckets until it is filled. If these keys are not moved here, a search for them will not be successful. We can anticipate when the current disks will overflow with blocks and resolve this by adding disks to slots in our address space at any time. To anticipate when to add more disks, we draw an upper threshold for each disk. If the amount of data on any disk exceeds this threshold as more data is added, the system is considered full and more disks should be added. Determining how close this threshold is to the full capacity of the disk depends on how much data is being stored. The more data being stored at once, the lower the threshold should be in order to catch any single disk aberration of high data occupancy. However, calculating an exact upper threshold level is beyond the scope of this paper.

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 have D disks that occupy slots $0, \ldots, D-1$. We can simply think of D disks which are *labeled* 0 through $D-1$, but we use the concept of disk occupying slots to help visualize our algorithm. We call this the Sequential Disk Labeling (SDL) algorithm.

As explained in Chapter 1.2, each block has a signature, X, generated by a pseudo-random number generator function, p_r1[1]. To determine the initial placement for each block, we use the block's signature, X, to compute its random start position, $sp = X \bmod P$, and we use X as the seed to a second function, p_r2, to compute its random step length, $sl = p\_r2(X) \bmod (P-1) + 1$. Because some slots contain disks and some do not, we want to probe slots until 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. sl should never be 0; avoiding repeated probes into the same slot. We probe by the same amount, sl, in order to guarantee that we search all slots in at most P probes. The first slot in the probe sequence that contains a disk becomes the address for that block. Note that we use the Linux file system in our current design to determine where a block is placed on a disk. For improved data throughput, we plan to design our own file system and place blocks directly onto the raw disk. Thus, sp and sl combine to make up the probe sequence where $p(\ell)$ is the slot address at the ℓ-th probe iteration as defined in Equation 3.

$$p(\ell) = (sp + \ell \times sl) \bmod P, \text{ where } \ell = 0, 1, 2, \ldots, P-1 \quad (3)$$

Example 1 As shown in Figure 2, assume we have 10 disks and 101 slots ($D = 10, P = 101$). We want to com-

[1] The standard C library functions srand() and rand() perform well as a pseudo-random number generator and its resulting distribution of blocks is indistinguishable from that of other generators such as Combined Tausworthe generators.

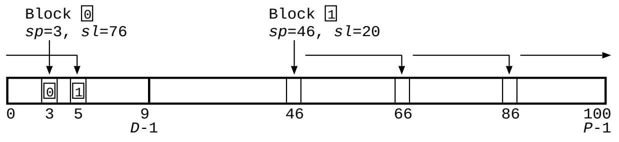

Fig. 2 Placement of two blocks. Block 0 initially hits and block 1 initially misses ($D = 10, P = 101$).

pute the signature X, sp, and sl for blocks 0 and 1 of a file. Using the blocks' filename (converted to an integer) added to the block number as the seed, the signature $X = p\_r1(\texttt{StrToL}(filename)+0) = 4851$ for block 0. To compute block 0's probe sequence, we use 4851 where $sp = 4851 \bmod P = 3$ and $sl = p\_r2(4851) \bmod (P-1)+1 = 76$. Recall that sl is in the range $1 \dots P-1$ which is why 1 is added in the computation of sl. Slot 3 contains a disk so we have found the address for block 0. For block 1, $X = p\_r1(\texttt{StrToL}(filename)+1) = 29934$. Similarly, we find $sp = 46$ and $sl = 20$. Slot 46 does not contain a disk so we traverse block 1's probe sequence, probing by 20 slots, until we arrive at a disk in slot 5. In general, finding disks for any given block may require multiple wrap-arounds. ∎

Because we set P to a prime number, the probe sequences are guaranteed to be a permutation of $0, 1, \dots, P-1$, that is, every slot is probed exactly once in the worst case. As long as P is relatively prime to sl, this holds true [10].

Next, we want to be able to add or remove disks to our address space. A scaling operation is the addition or removal of n disks. For block i, $d_{i,j-1}$ is its address *after* scaling operation $j-1$ and $d_{i,j}$ is the address *after* operation j. $d_{i,j-1}$ may or may not equal $d_{i,j}$ depending on if block i moves after operation j. For addition operations, the n disks will be added sequentially to slots $D, \dots, (D+n-1)$ as long as $(D+n) \leq P$. For disk removals, when the overall amount of data shrinks, the n disks are removed from slots $(D-1-n), \dots, (D-1)$. When disks are added to and removed from slots in this manner, the memory requirement is constant since only the number of disks, D, needs to be stored.

To perform an addition operation, first we add disks to the appropriate slots. Then we iterate through each block, i, in sequence ($i = 0, \dots, B-1$) and, without actually accessing these blocks, we compute X, sp, sl, and $d_{i,j}$ for block i. If $d_{i,j}$ is an old disk ($d_{i,j} = d_{i,j-1}$), 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 $d_{i,j}$ is a new disk ($d_{i,j} \neq d_{i,j-1}$), then this should be the new location for the block. We continue with the probe sequence to find $d_{i,j-1}$, which is the current address, and move the block to $d_{i,j}$. In this case, the probe length becomes shorter since, before this add operation, the slot, $d_{i,j}$, was also probed but was empty so probing continued.

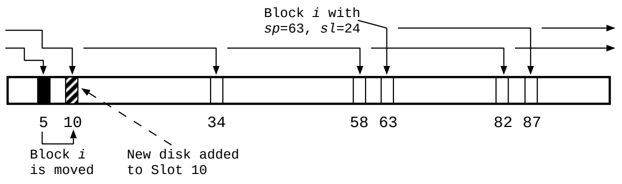

Fig. 3 Probe sequence of block i before and after a disk add operation j. Block i moves from disk 5 to disk 10 after disk 10 is added ($D = 10, P = 101$).

Example 2 Figure 3 shows an example of adding a new disk to slot 10. Before scaling operation j, block i's probe sequence is $sp, (sp+sl) \bmod P, (sp+2\times sl) \bmod P, \dots, (sp+6\times sl) \bmod P$. Here, $d_{i,j-1} = (sp+6\times sl) \bmod P$. For this example, $sp = 63$ 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 $63, 87, 10$ and $d_{i,j} = (sp+2\times sl) \bmod P$. ∎

For removal operations, we first mark the disks which will be removed. Then, for each block stored on these disks, we continue with the probe sequence until we hit an unmarked disk, $d_{i,j}$, 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.

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. So the scaling operation and the presence 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.

We were surprised to find that the SDL algorithm does not lead to an even distribution of blocks across the disks. As shown in the next section, the distribution appears "bowl" shaped where the first few and last few disks contain more blocks than the center disks. In Section 5, we provide a more superior algorithm called Random Disk Labeling.

4.1 Non-uniformity of the sequential disk labeling algorithm

To understand why a bowl-shaped distribution is observed, we intuitively determine how likely blocks will fall on certain disks. We examine four cases, which in sum will result in a bowl-shaped distribution of blocks. The bowl-shape is more pronounced when D is much smaller than P so we use this assumption in our analysis.

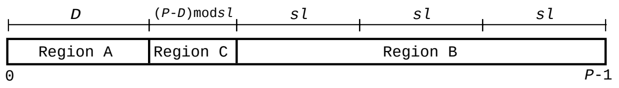

Fig. 4 Three regions of the address space.

We split our address space into Regions A, B, and C in Figure 4. Region A contains slots $0, \ldots, D-1$. Region B contains slots $(D+(P-D) \bmod sl), \ldots, P-1$. Region C contains slots $D, \ldots, (D + (P - D) \bmod sl - 1)$. For Case 1, probing is successful on the first try, meaning that *sp* is in Region A. The remaining cases involve an initial probe miss where *sp* falls in Regions B or C. The only difference in these cases is the range of *sl*. For Case 2, $1 \leq sl \leq D$. For Case 3, $(P - D) \leq sl \leq (P - 1)$. Finally for Case 4, $D < sl < (P - D)$.

Case 1: If *sp* falls in Region A, each disk has an equal chance of receiving the block. These blocks are uniformly distributed among the disks. The value of *sl* is irrelevant since no hopping is required. This case does not contribute to the bowl-shaped distribution.

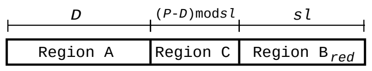

Fig. 5 Three regions of the address space with Region B reduced.

Case 2: If *sp* falls in Regions B or C and $1 \leq sl \leq D$ then the distribution will be skewed to the left creating the left edge of the bowl. To illustrate, we first reduce Region B in Figure 4 to the length of one *sl* and call it Region B$_{red}$ as in Figure 5. This is possible since there is an equal probability that *sp* will land in any slot in Region B and will eventually hop by length *sl* to the rightmost *sl* group.

Now if *sp* falls in Region B$_{red}$, it is clear that the block will land among slots $0, \ldots, sl - 1$ after only *one* hop of *sl*. This hop will wrap-around from the end of the table to the beginning. Moreover, since $sl \leq D$, the block will definitely hit a disk after only one wrap-around. To compute the probability of how likely a block is to land on a specific disk, we look at different sizes of *sl*. If $sl = 1$ then the block will hit Disk 0 with a 100% probability. If $sl = 2$ then there is a 50% chance of hitting Disk 0 and a 50% chance of hitting Disk 1. If $sl = 3$ then 33% each on Disks 0, 1, and 2. In general, we can find the probability of a block initially landing in Region B$_{red}$ hitting any disk using Equation 4 where Pr_d is the probability of hitting disk d among D total disks.

$$Pr_d = \frac{1}{D} \sum_{i=d+1}^{D} \frac{1}{i} \tag{4}$$

Equation 4 shows that with smaller d values, a block will have a higher probability of landing on Disk d. Since the disks with smaller d values are located towards the left of Region A, more blocks would be assigned to these disks. Thus the left edge of the bowl is formed.

Since Region C is always smaller than *sl* and usually much smaller than Region B, few blocks will initially hit here. The contribution of blocks landing in this region do not greatly affect the bowl distribution so we ignore this case.

Case 3: If *sp* falls in Regions B or C from Figure 4 and $(P-D) \leq sl \leq (P-1)$ then the distribution will be skewed to the right causing the right edge of the bowl. Because $sl \geq (P - D)$, one hop of length *sl* will wrap around the table and the new slot will be less than or equal to D slots to the left of the original slot. Essentially in this case we are hopping to the left by D or less slots each time. This behavior is the mirror image of Case 2 and the probability of hitting any disk in this case can be computed using Equation 5.

$$Pr_d = \frac{1}{D} \sum_{i=P-d}^{D} \frac{1}{i} \tag{5}$$

Similar to Equation 4, Equation 5 shows that with larger d values, a block will have a higher probability of landing on Disk d. The right edge of the bowl is formed since disks with larger d values exist towards the right of Region A.

Case 4: If *sp* falls in Regions B or C and $D < sl < (P - D)$ then the distribution will have a slight bowl effect where the bowl is shallow. It appears that this case can be further broken into sub-cases where left or right edge contributions can be isolated. However, we have shown that Cases 2 and 3 already demonstrate a major contribution to the left and right edges. Therefore, we do not further investigate this case.

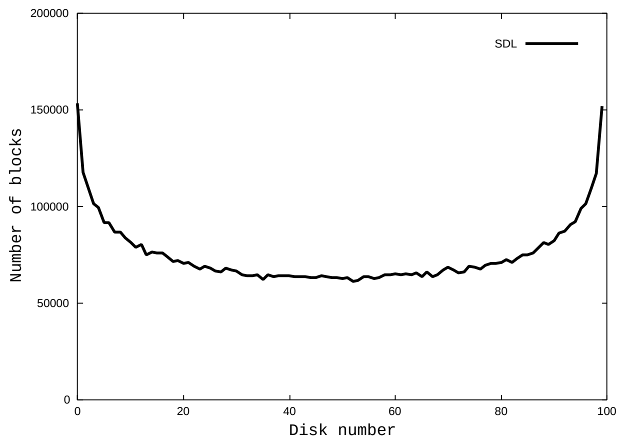

Fig. 6 Number of blocks on disks using SDL ($D = 100, P = 10,007$).

The total probabilities of a block landing on a disk are the sum of the three cases above. Initial misses, blocks with *sp*'s in Regions B and C, cause the bowl-shaped distribution. Figure 6 shows the distribution of blocks resulting from the SDL algorithm with $D = 100$, $P = 10,007$, and approximately 7.5 million blocks.

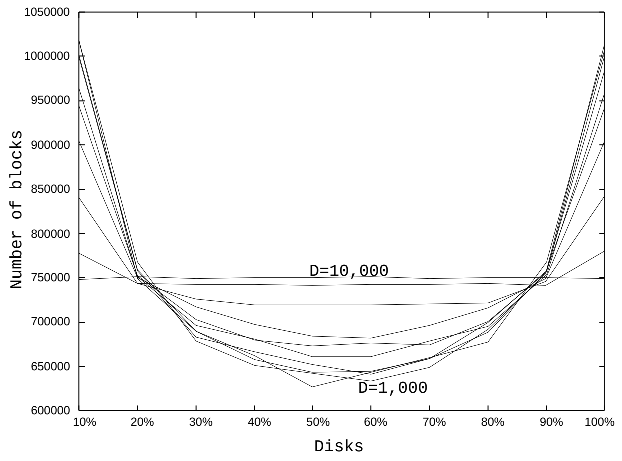

Fig. 7 Normalized distribution as # of disks increases ($P = 10,007$).

In Figure 7, we observe the distribution when we add disks from $1,000$ to $10,000$ at $1,000$-disk increments. The y-axis shows the number of blocks that are loaded on the first 10% of the disks, the second 10% of the disks, and so on. The distribution becomes more uniform as we increase the number of disks because, as D approaches P, the probability that sp hits a disk, (D/P), approaches 1. However, we want to initially set D to be less than P to allow for disk additions.

5 Random disk labeling (RDL) algorithm

Our Random Disk Labeling (RDL) algorithm is similar to the SDL algorithm except that we use a random allocation of disks where we randomly place D disks among the P slots instead of placing them sequentially in slots $0, \ldots, D-1$ as with SDL. Likewise, during storage scaling, we add a disk to an empty slot that is randomly chosen or remove a randomly selected disk in order to maintain the overall random allocation scheme. Essentially, we use double hashing on disks *labeled* with random slots in the range $0, \ldots, P-1$. The memory requirement for RDL is larger than that for SDL. Here, a table is needed to store the slot locations of each disk, so the requirement is on the order of $O(P)$. Recall that the memory requirement for SDL is constant since only the number of disks is stored.

In practice, since the sequence of where disks are added or removed does not have to be reproduced, we can use any random number generator (i.e., non-pseudo-random). However, for the sake of simplicity, we use a pseudo-random number generator in our design and implementation.

The RDL algorithm results in a much more even distribution of blocks. Figure 8 shows that each disk has an approximately equal number of blocks where $D = 100$ and $P = 10,007$. The x-axis indicates the disk number ranging from the 0-th disk to the 99-th disk. Although each disk is assigned to a random slot (e.g., Disk 0 in Slot 423, Disk 1 in Slot 29, etc.) we only show the disk number in Figure 8. Because the disks are not sequentially clustered together, the placement of blocks will not favor any particular disk. Here, by using a fixed step length, sl, to probe along disks that are randomly spaced apart, we are in essence probing sequentially arranged disks using a sequence of random sl's. Probing by random sl's would not cause any disks to be favored and thus would lead to an even distribution. Here, sp still has a $\frac{D}{P}$ chance of hitting any disk, but on initial sp misses, probing by a fixed sl will not contribute to a bowl shaped distribution because of the random labelings. Also, termination of probing when finding a block is still guaranteed to be at most P probes since probe sequences are still permutations of $0, 1, \ldots, P-1$. Later in Section 7, we will give a direct comparison of the improvement of RDL over SDL.

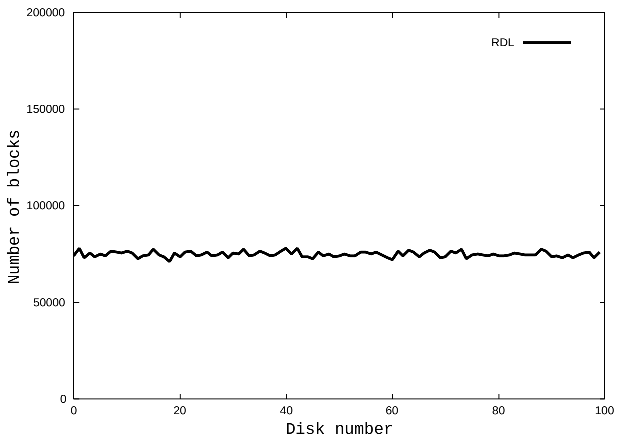

Fig. 8 Number of blocks on disks using RDL ($D = 100, P = 10,007$).

6 Random probing

We applied double hashing to Sequential Disk Labeling (SDL) and Random Disk Labeling (RDL). We now show that *random probing* can further improve the uniformity of block distribution. In this section, we will present a two-phase algorithm consisting of a random probing phase (Phase 1) and a double hashing phase (Phase 2). Phase 2 is initiated only if Phase 1 fails to find a non-empty slot within K probes. K is the *maximum* number of probes that Phase 1 performs before switching to Phase 2. This two-phase algorithm can be applied to disks which are both sequentially labeled and randomly labeled which we refer to as SDL-RP and RDL-RP, respectively, for the remainder of this paper. We also refer to the RDL and RDL-RP class of algorithms as RDL*. Likewise SDL and SDL-RP are referred to as SDL*.

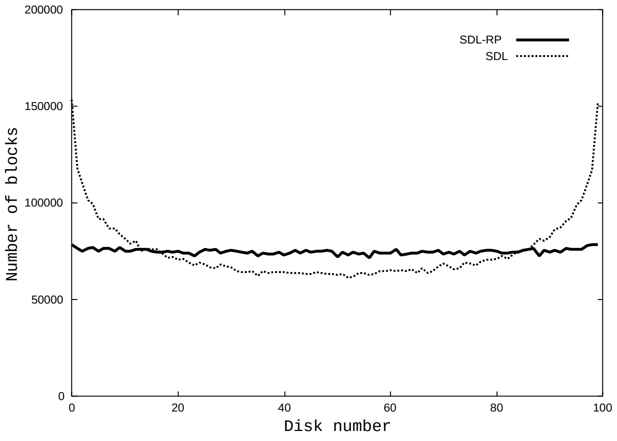

Figure 9a: SDL-RP ($K = 285, P = 10,007$).

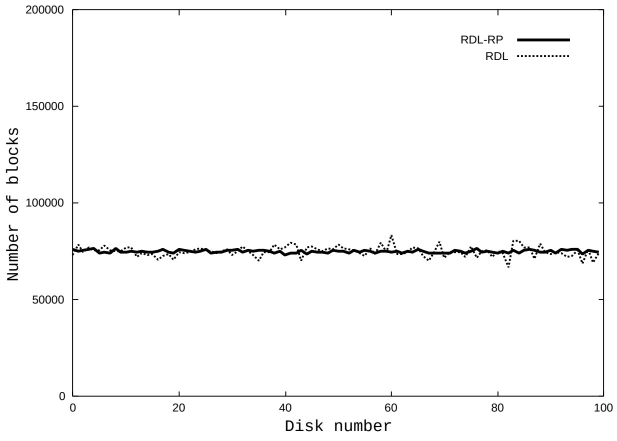

Figure 9b: RDL-RP ($K = 150, P = 1,009$).

Fig. 9 Number of blocks on disks using SDL-RP and RDL-RP.

To find a block's address using random probing, we repeatedly call a pseudo-random number generator, with block i's signature as the seed, to produce a sequence of values in the range $0 \ldots P-1$. The first slot that is occupied by a disk in this probe sequence becomes the address for block i. We are, of course, only probing the slots of our memory-resident address space.

Each probe simply involves selecting a random value, sp, using a pseudo-random number generator in the range $0, \ldots, P-1$ until a disk is hit. Intuitively, it is clear that this will lead to a uniform distribution of blocks assuming a well-performing pseudo-random number generator. However, there is no guarantee of termination and the probability of continuously hitting empty slots exists, though very small. Our two-phased algorithm solves this problem. In Phase 1, we perform a maximum of K random probing trials. If no disk is hit during Phase 1, we enter Phase 2 and perform double hashing where, again, termination within P trials is guaranteed.

We show the number of blocks on 100 disks using this approach in Figure 9. With SDL-RP using a maximum of $K = 285$ random probing trials, we see in Figure 9a that a much more even distribution is achieved as compared to SDL. Some cases of RDL can also benefit from random probing. Although the case where RDL with $10,007$ slots already exhibits good uniformity as shown in Figure 8, the case where RDL with $1,009$ slots results in worse uniformity. In Figure 9b, with RDL-RP where $D = 100$ and $P = 1,009$, using a maximum of $K = 150$ random probes will improve the level of uniformity. Since RDL in Figure 8 shows better uniformity than SDL in Figure 6, RDL-RP will require fewer random probes than SDL-RP to achieve similar levels of uniformity. In Section 7, we compare the levels of uniformity achieved by SDL-RP and RDL-RP and show that RDL-RP never requires more random probes than SDL-RP. Even though Phase 2 of SDL-RP still introduces a small amount of the bowl effect, Figure 9a shows that the combination of the two phases virtually eliminates this. Using K random probes in either SDL-RP or RDL-RP, we can now find a block in at most $K + P$ trials.

A question that remains is when to switch from Phase 1 to Phase 2 (i.e., what should be the value of K)? If we know the desired level of uniformity then we can calculate K for SDL-RP using a closed-form formula given as Equation 11 in Appendix A. However, we were unable to derive a similar formula to compute K for RDL-RP. Instead, we provide some insight on potential K values in our experiments in Section 7.

7 Performance evaluation

In this section, we first compare the computation time of RDL with a prior work called Highest Random Weight (HRW). Then, we compare the load uniformity of RDL and SDL. Finally, we compare the amount of probing of RDL-RP and SDL-RP when varying the maximum probes, K, and the number of slots, P.

All of our performance measurements were conducted through simulation. However, the simulated results (i.e., block locations and their retrieval methods) are identical to the results of an actual implementation. We use our simulation to measure the quantity of blocks on disks and the cost of locating them, regardless of block size or content.

7.1 RDL vs. HRW

HRW, described in Section 2, attempts to redistribute Web objects (data blocks in our case) residing on a scalable group of proxy servers (disks). HRW does in fact satisfy our first two requirements (even load and minimal data movement) described in Section 1.2, however it fails in Requirement 3 (fast access). The time to redistribute all the blocks is the summation of the access times of every block. HRW's time to redistribute up to D disks is actually similar to SCADDAR's time to redistribute up to D disks for the case that SCADDAR's previous $D-1$ operations were all 1-disk adds. In this case, the number of disks equals the number of scaling operations. Recalling that HRW requires $B \times D$ pseudo-random function calls, for B blocks and D disks, during scaling, we show that this is significantly more time consuming than RDL. In some cases, the computation time of HRW is several orders of magnitude higher than RDL's time. Note that the time includes only CPU computation time, not block transfer time, which would be similar for both RDL and HRW since they are transferring similar numbers of blocks.

We simulated the initial placement of 1MB-data blocks using RDL and HRW onto a group of 75GB-disks and measured the running time. A Pentium III 933Mhz PC with 256MB of memory was used to run the simulation written in C. Since the block layout on D disks from an initial placement is identical to the block layout after scaling up to D disks, we are really measuring the cost of a scaling operation.

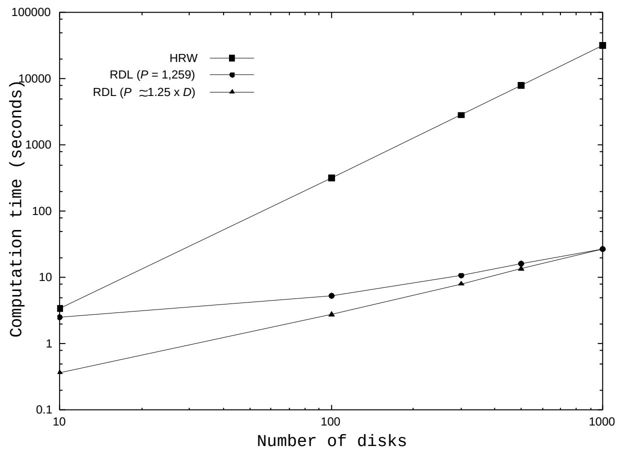

Fig. 10 Computation time of HRW and RDL with $P = 1,259$ and $P \approx 1.25 \times D$.

Figure 10 shows the computation time needed to scale from any arbitrary number of disks to D disks. Note that for a more realistic simulation we are increasing the number of blocks as we increase the disks instead of keeping the blocks constant. Thus, about 75,000 blocks reside on each disk as they are scaled. However, if we kept the total number of blocks constant, the percentage of improvement of RDL over HRW would still be the same. We compare the computation time of HRW with RDL when P is fixed to 1,259 and when P is ˜25% higher than D. The x-axis is logarithmic and represents the number of disks ranging from 10 to 1,000. The y-axis, also logarithmic, is the computation time in seconds. HRW requires much more time to compute all the block locations than RDL. The computation time may be insignificant compared to the data redistribution time when scaling a small set of disks, but can be a large factor of the total scaling time when dealing with a large set of disks. So, choosing the right algorithm for computing block locations becomes important for these large disk sets. For example, from Figure 10, the computation times of HRW and RDL are both less than 5 seconds when scaling from 9 to 10 75GB-disks. This is clearly insignificant compared to the ˜6,000 seconds needed to redistribute the 75GB of data for this scaling operation (with a 100Mbps inter-disk connection). However, when scaling, say, 999 to 1,000 disks, HRW requires 31,492 seconds of computation time whereas RDL only requires 27 seconds. Thus, the total computation time plus the 6,000 seconds of data redistribution time is 37,492 seconds (HRW) compared to 6,027 seconds (RDL). Here, HRW imposes a much higher computational overhead than RDL. The large time difference is because RDL performs only B pseudo-random function calls and some probing for each block as compared to $B \times D$ calls for HRW[2].

7.2 Load balancing

Next, we conducted several experiments to compare how well our SDL* (i.e., SDL and SDL-RP) and RDL* (i.e., RDL and RDL-RP) algorithms reorganized data blocks after successive scaling operations. We need a metric to measure the uniformity of the block distribution in order to gauge the load balancing of the set of disks after scaling. We use a "goodness of fit" statistic, χ^2, as the metric for comparing how well the distributions from the SDL* and RDL* algorithms match up with a perfectly even distribution (i.e., equal number of blocks on all disks). The χ^2 equation is defined as

$$\chi^2 = \sum_{d=0}^{D-1} \frac{(x_d - E)^2}{E} \tag{6}$$

where E is the expected value of the number of blocks per disk and x_d is the number of blocks on disk d [18].

For each set of experiments, we simulated scaling from 10 disks to 10,007 disks using roughly 7.5 million data blocks. Each scaling operation is an addition of 10 disks. For SDL*, the first 10 disks are placed in slots $0 \ldots 9$ and subsequent 10-disk adds are placed in the next 10 available slots (i.e., $10 \ldots 19$, $20 \ldots 29$, etc.). For RDL*, the first 10 disks are placed in 10 randomly chosen empty slots and subsequent 10-disk adds are placed into 10 randomly chosen slots from the remaining empty slots. Here, $P = 10,007$ and we observe the trend of the χ^2 values as D approaches P. Figure 11 shows the χ^2 values of SDL and RDL as we scale the number of

[2] Each pseudo-random function call in HRW's algorithm actually requires two srand() and rand() calls whereas RDL's algorithm only requires one call of each.

Table 2 Estimation of the χ^2 means and standard deviations.

D	Sample size (n_s)	95% confidence interval of population mean	Sample standard deviation (ς)	Interval size $\times$ $100/\overline{X}$ (%)
1,000	100	(10,317.3, 10,503.6)	475.0	1.79%
5,000	100	(29,700.5, 29,976.8)	704.8	0.93%
8,000	100	(32,700.6, 32,957.7)	656.0	0.78%
10,007	1	23,509.5 (actual average)	0	n/a

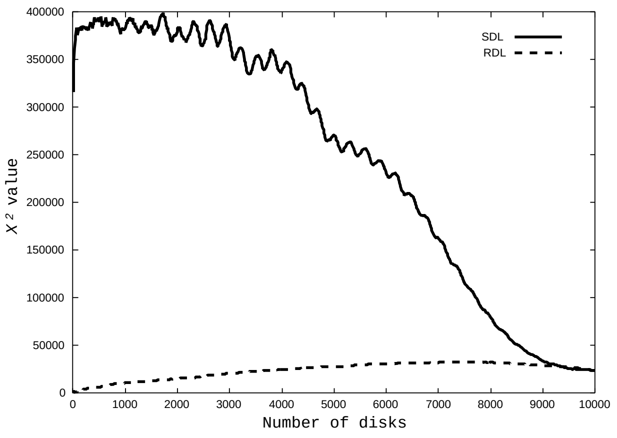

Fig. 11 χ^2 values of SDL and RDL ($P = 10,007$).

disks to P. The χ^2 values for the SDL algorithm actually decrease as disks are added since the bowl-shaped distribution becomes wider, causing the nonuniformity to become less apparent. Here, RDL shows significantly lower χ^2 values than SDL and thus has a more uniform distribution. Only as D approaches P do the χ^2 values become similar. The curves actually converge to the same χ^2 value when $D = P$ because when there are no empty slots, the block distribution will be identical for SDL and RDL. The χ^2 results for SDL-RP and RDL-RP are very similar to each other and, furthermore, are negligible to (though just slightly lower than) the RDL curve. This demonstrates that random probing improves load balancing for both SDL and RDL, although the degree of improvement for SDL is much greater. Thus, we omit the SDL-RP and RDL-RP curves from Figure 11.

For SDL, when adding one disk there is only one possible slot to add that disk; the next available slot. Thus, for every D value only one χ^2 value is possible. However, for RDL, there are $\binom{P}{D}$ ("P choose D") possible combinations of which slots the disks can reside in since they are chosen at random, so there are $\binom{P}{D}$ possible χ^2 values. Because Figure 11 shows the χ^2 value for just one combination of each D value, we want to find the mean and standard deviation of the χ^2 values for all the $\binom{P}{D}$ combinations of each D value. However, as we scale D to P, $\binom{P}{D}$ becomes very large until $D = P/2$, then decreases until $D = P$. Even the smallest $\binom{P}{D}$ values are very large. When $P = 10,007$ there are roughly 50 million combinations of disk placements for 2 disks (and also for $10,005$ disks) so, finding a χ^2 value for every combination is computationally intractable. Therefore, we use a random sampling technique to approximate the mean and the standard deviation of the χ^2 values of all the possible combinations for a specified D and P.

Given a D and a P, we take a random sample from a population of size $\binom{P}{D}$ to estimate the population mean, μ. The estimate of the mean is represented as a *confidence interval.* A 95% confidence interval says that μ will fall in this interval with a 95% probability [18]. A confidence interval can be computed from parameters of the random sample by using Equation 7:

$$\text{confidence interval} = \overline{X} \pm z\left(\frac{1-\frac{\beta}{100}}{2}\right) \times \frac{\varsigma}{\sqrt{n_s}} \quad (7)$$

where β is the percentage of confidence, $\overline{X}$ is the sample mean, ς is the sample standard deviation, n_s is the sample size, and $z(\alpha)$ is a z-score value. A standard z-score table can be used to look up the z-score for a particular α [18].

We do not display our confidence interval results graphically since the intervals are too small. Instead we provide the interval values in a tabular format. Table 2 shows 95% confidence intervals of random samples taken for $D = 1,000,\ 5,000,$ and $8,000$. Each row in the table is a separate random sample. Table 2 also shows the sample standard deviations, which are good estimators of the population standard deviations [18]. When $D = 10,007$ there is only one possible χ^2 value; where every slot contains a disk. We set $n_s = 100$ for each sample, in other words, 100 χ^2 values, or *sample members*, were randomly selected for each random sample from the $\binom{P}{D}$ possible combinations. A sample size where n_s is greater than 25 or 30 is usually large enough [18]. For each sample member, we choose D unique slots to which we assigned D disks. From Table 2, we illustrate that the confidence interval size is insignificant and a very small percentage of the sample mean. Therefore, Figure 11 is a good representation for RDL.

7.3 Probe lengths

The distribution uniformity of SDL and RDL can be improved with random probing, but these additional probes increase the overall number of probes. We empirically

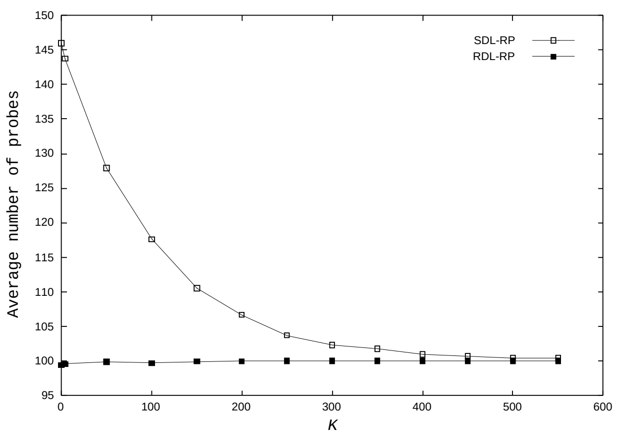

Figure 12a: Average number of total probes.

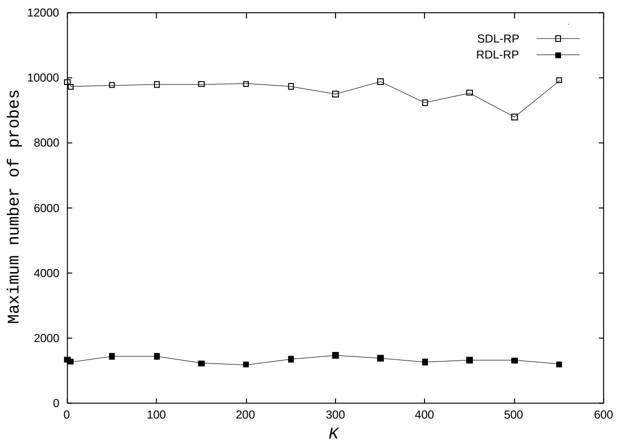

Figure 12b: Maximum number of total probes.

Fig. 12 Varying the maximum number of random probes, K $(D = 100, P = 10,007)$.

measured the amount of probing required by SDL-RP and RDL-RP to find blocks. Recall that for these algorithms, we perform K maximum random probes in Phase 1 and double hashing in Phase 2. When $K = 0$, SDL-RP is equivalent to SDL and RDL-RP is equivalent to RDL. We found that varying the K value will cause the average number of probes required by SDL-RP to vary while the average probes for RDL-RP remains relatively constant. We set the number of disks, D, to 100 and the number of slots, P, to $10,007$ for these experiments. Figure 12a shows that with a low K, SDL-RP requires almost 1.5 times more probes on average than RDL-RP and there is a higher chance that termination of probing will occur in Phase 2 (double hashing) for both algorithms. Termination in Phase 2 for SDL-RP will lead to more probing than termination in Phase 2 for RDL-RP. This results from the large gap of consecutive empty slots in SDL-RP since all the disks are clustered together. Using a higher K causes SDL-RP's average probe count to converge with that of RDL-RP and Phase 1 termination is more likely, thus probabilistically leading to P/D average probes for both algorithms.

Clearly, setting K to a lower value results in a better probing performance for RDL-RP than SDL-RP. Although the amount of probing required by both algorithms in Figure 12a seem similar when using large K's, the *maximum* probe length of RDL-RP is shorter than that of SDL-RP. We show in Figure 12b that when setting $K = 550$, SDL-RP had a maximum probe length of $9,930$ while RDL-RP's maximum probe length was $1,206$. In general, for either algorithm, setting K to a large value (e.g., on the order of P) should be avoided since there is a small chance that the maximum total probe lengths could be very large as well. Moreover, a very large K value will not result in significantly better load balancing.

The choice in the number of slots, P, affects the average number of probes for SDL-RP and RDL-RP. We set the number of disks, D, to 100 and use $K = 50$ random probes for Phase 1 to show the average number of probes as P varies. Figure 13a shows that, for both algorithms, the number of average probes increases as P increases since there are more empty slots to probe. The maximum probe lengths also increase as shown in Figure 13b. However, as P increases, SDL-RP requires more probes than RDL-RP since the size of the gap of consecutive empty slots increases.

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, using a large P value requires more probing so, ideally, the growth of the storage system should be gauged beforehand to determine a good P since it cannot be altered later without causing a complete reorganization of data blocks. An advantage of RDL-RP here is that even with a large P value, RDL-RP requires fewer average and maximum probes than SDL-RP, as shown in Figure 13.

RDL produces a more uniform distribution of blocks than SDL as seen in Figure 11. When using a low maximum number of random probes, K, Figure 12 shows that RDL-RP requires fewer probes than SDL-RP when locating blocks. Figure 13 shows that with a large number of slots, P, RDL-RP results in fewer probes. Moreover, a large P value allows for a more scalable disk set. In all cases, RDL-RP performs the same as or better than SDL-RP.

8 Implementation issues

In this section we explore several implementation issues with the storage scaling techniques described in this paper. These issues include continuous file availability during real-time block reorganization and providing block redundancy to improve load balancing.

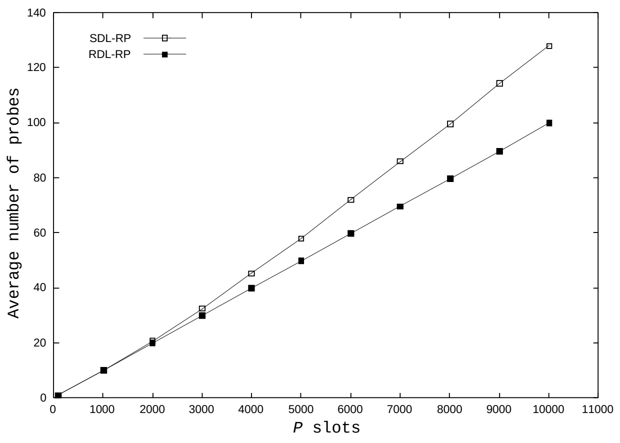

Figure 13a: Average number of total probes.

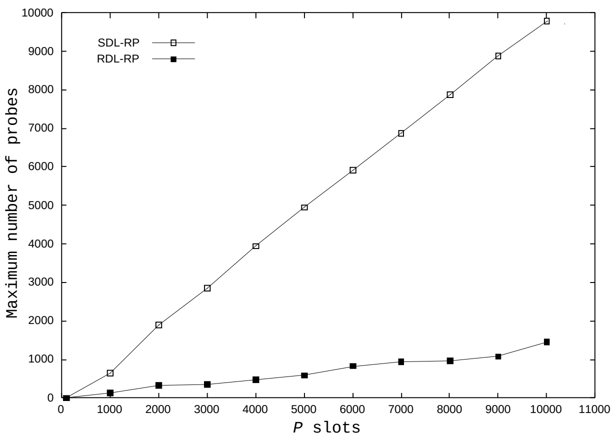

Figure 13b: Maximum number of total probes.

Fig. 13 Varying the number of slots, P $(D = 100, K = 50)$.

8.1 Availability

When real-time storage scaling is performed and block reorganization is occurring, the availability of all files should be maintained. In general, if a block is identified and designated to move from a source disk to a target disk, a copy of the block should first be placed on the target disk. Then eventually, depending on the type of reorganization approach described below, the block copy becomes visible and the original block is deleted. To identify the sequence of blocks to be moved, two approaches are described below: a file-by-file approach and a disk-by-disk approach.

With file-by-file reorganization, we reorganize an entire file before updating system configurations to indicate that this file is now in accordance with the new disk layout. Once the entire file is reorganized, the original blocks are deleted. The new block layout now becomes visible to all file requests. We repeat this for every file on the storage system. The downside of this approach is that disks will be accessed multiple times during reorganization since each disk will most likely contain blocks from every file.

Another approach is reorganizing blocks disk-by-disk. This approach will avoid repeated disk accesses and reduce seek overhead since we are reorganizing an entire disk at a time. Here, we evaluate every block from a list of blocks on a disk without actually accessing the blocks until they are copied to a target disk. For every block on the block list, we can use a block's name to determine whether the block should be copied or not. Since we are storing each block as a file, the block's name actually contains the name of the file object that this block belongs to and the block number. For example, a block named "TopGun_311.blk" indicates this block is the 311^{rd} block of file object **TopGun**. Now, we can use (`StrToL`("**TopGun**") + 311) as the seed to the pseudo-random number generator to determine if it needs to be copied out. Once all the disks are reorganized, the system is updated so all requests switch to the new disk layout. The original blocks are then deleted. The drawback here is that the new disk layout will be realized only after all the blocks are reorganized. Also, when deciding whether blocks should move, every block in the list will cause repeated iterations through the files, which adds to the computational complexity.

8.2 Block replication

With random data placement, short term load imbalance is statistically possible. Some disks could be temporarily overloaded while other disks are idle. In [21], the authors suggest the use of block replication to improve the load balancing in these cases. Even with 25% of the blocks replicated, load balancing is significantly improved.

To apply a 25% block replication scheme to our disk labeling technique, we consider each block and decide with a 25% probability whether to replicate the block or not. If a block is to be replicated, we decide where the block copy should reside based on the probe sequence. Recall that the original block resides on the first disk along the probe sequence. Similarly, the block copy should reside on the *next* disk along the probe sequence. Note that the block copy will never reside on the original block's disk since the probe sequence does not contain any duplicate disks. During reorganization due to disk addition, if an original block is moved, then its copy should move to the disk of the original block. Likewise, reorganization due to disk removals will cause the original block to move to the block copy location and the block copy will be moved to the next disk along the probe sequence. Some optimizations can be applied here but they are beyond the scope of this paper.

9 Concluding discussions and future work

The three requirements from Section 1.2 are satisfied by the RDL* algorithms (i.e., RDL and RDL-RP). SDL-RP also satisfies these requirements with enough random probes.

For the SDL-RP and RDL* algorithms, after every scaling operation, the placement of the blocks is identical to the placement that would have been achieved if we had initially placed the blocks on those disks. Therefore, no history information of scaling operations needs to be maintained to find block locations. This is because for a specific block, the probe sequence will always remain the same and scaling operations only affect the point along the probe sequence where the block will reside. Because scaling operations always bring the block distribution back to an initial state, we know that uniformity is achieved since all initial states are uniform. This satisfies load balancing of disks and Requirement 1.

The amount of block movement during scaling operations is minimized so Requirement 2 is satisfied. Blocks will only move to new disks during addition operations and never from an old disk to another old disk. Also, blocks to be moved are randomly chosen from the old disks. Similarly for removal operations, blocks are moved off of removed disks and randomly redistributed across the remaining disks.

Finally, a maximum of $K + P$ probes is guaranteed for SDL-RP and RDL-RP to find any particular block. This satisfies Requirement 3 since the access complexity is measured by the number of probes needed to locate a block. Again, probing is only performed on the slots of our memory-resident address space. In practice, even though we rarely observe the worst case maximum probe lengths, we must take measures to reduce them by using as few random probes as possible.

The SDL* algorithms are simpler with less metadata than RDL*. SDL* stores only the number of disks since disks are added to the next available slots (i.e., slots $0, \ldots, D-1$ will be occupied). However, for disk removals, if disks are not removed from the end then the empty slots need to be stored as well. As discussed in Section 4, when n disks are removed (to conserve space), they are removed from the end slots $(D-1-n), \ldots, (D-1)$. So if simplicity is desired, then SDL* are better choices than RDL* even though the load balancing performance is worse. If better load balancing is desired and added probing costs can be incurred, then SDL-RP is a better choice than SDL.

With RDL*, every disk's position must be randomly generated and then stored in a table. This table is updated on disk additions and removals. Thus, RDL* can achieve better load balancing, with some added complexity costs. If even better load balancing is desired and extra probing can be incurred, then RDL-RP should be used instead of RDL.

We will continue to refine and extend our algorithms. Scalability would be improved if we could grow beyond P disks without requiring a complete organization of all the data blocks. One possibility is to use another scaling technique such as SCADDAR [8] to continue scale-up once the bound of P is reached. Since SCADDAR incurs a slight accumulation of performance penalties as the system is scaled, a complete reorganization may eventually be necessary to eliminate these penalties. However, this complete reorganization can be executed gradually as a background process only when resources are available. We are also planning to incorporate RDL-RP into the storage subsystem of our actual real-time server system, namely *Yima* [22], as well as non-real-time servers.

We would like to modify our algorithms to allow for more frequent and incremental scaling as expected in peer-to-peer systems. Instead of load balancing after every scaling step, we could batch a set of scaling steps and load balance after this batch operation. This would result in a more universal solution for unknown data growth rates and would be able to balance high-growth and low-growth data requiring frequent and infrequent scaling steps, respectively.

Throughout this paper, we have assumed our scaling techniques to operate on a set of homogeneous physical disks. However, often times when retiring an old disk, the new replacement disk might have improved characteristics since the older disk models are no longer available. So we need to extend our techniques to operate on heterogeneous physical disks. Without modifying our algorithms, we can accomplish this by using a technique called *Disk Merging* described in [28] to build a layer of homogeneous *logical* disks above the heterogeneous physical disks. Our disk labeling techniques can then operate on these logical disks as usual.

Data mirroring may be a solution for fault-tolerance with RDL. Mirrored blocks can be placed at a fixed offset determined by a function $f(D)$. For example, $f(D)$ would return $D/2$ as an offset. Furthermore, each RDL slot is a storage unit which we assume to be a single disk drive. However, this storage unit can also represent a RAID device. When scaling the storage with redundancy, an entire RAID device can be either added or removed from a slot.

Acknowledgements. We would like to thank Rolfe R. Schmidt for his suggestions on minimizing random probing trials, and Larry Goldstein for his advice on finding confidence intervals. We would also like to thank the anonymous referees for their invaluable comments.

References

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

2. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, 1990.
3. J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.
4. D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.
5. D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.
6. 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.
7. S. Ghandeharizadeh and S. H. Kim. Striping in Multi-disk Video Servers. In *Proceedings of the SPIE High-Density Data Recording and Retrieval Technologies Conference*, pages 88–102, October 1995.
8. A. Goel, C. Shahabi, S.-Y. D. Yao, and R. Zimmermann. SCADDAR: An Efficient Randomized Technique to Reorganize Continuous Media Blocks. In *Proceedings of the 18th International Conference on Data Engineering*, pages 473–482, February 2002.
9. D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663, May 1997.
10. D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1998.
11. P.-Å. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4), April 1988.
12. W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* — A Scalable, Distributed Data Structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.
13. C. Martin, P. S. Narayan, B. Özden, R. Rastogi, and A. Silberschatz. The Fellini Multimedia Storage Server. In S. M. Chung, editor, *Multimedia Information Storage and Management*, chapter 5. Kluwer Academic Publishers, Boston, August 1996. ISBN: 0-7923-9764-9.
14. M. Mehta and D. J. DeWitt. Data Placement in Shared-Nothing Parallel Database Systems. *The VLDB Journal*, 6(1):53–72, 1997.
15. R. Morris. Scatter Storage Techniques. *Communications of the ACM*, 11(1):38–44, January 1968.
16. R. Muntz, J. Santos, and S. Berson. RIO: A Real-time Multimedia Object Server. In *ACM Sigmetrics Performance Evaluation Review*, volume 25, September 1997.
17. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM '01*, pages 161–172, August 2001.
18. J. A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, 1995.
19. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 329–350, November 2001.
20. J. R. Santos and R. R. Muntz. Performance Analysis of the RIO Multimedia Storage System with Heterogeneous Disk Configurations. In *ACM Multimedia Conference*, Bristol, UK, 1998.
21. 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.
22. C. Shahabi, R. Zimmermann, K. Fu, and S.-Y. D. Yao. Yima: A Second Generation Continuous Media Server. *IEEE Computer*, pages 56–64, June 2002.
23. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, May 2001.
24. Teradata Corp. *DBC/1012 Data Base Computer System Manual. Teradata Corp. Document No. C10-0001-02, Release 2.0*, November 1985.
25. D. G. Thaler and C. V. Ravishankar. Using Name-Based Mappings to Increase Hit Rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, February 1998.
26. Q. Xin, E. L. Miller, T. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability Mechanisms for Very Large Storage Systems. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, pages 146–156, April 2003.
27. B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Tech. Rep. UCB/CSD-01-1141, UC Berkeley, April 2001.
28. 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 Minimizing the random probing trials for SDL-RP

We can control the amount of uniformity resulting from the SDL-RP algorithm by varying the maximum number of random probing trials before switching to Phase 2. Different levels of uniformity may be better suited for different applications. An application that is not as sensitive to a uniform block distribution may benefit from fewer probes when finding block addresses.

A perfectly uniform block distribution will result in a $1/D$ percentage of all the blocks on every disk. The worst-case distribution, that of a bowl-shaped one, has more blocks on the first and last disks and the fewest blocks on the center disks. We use the bowl bottom, b, to measure the shape of a bowl resulting from SDL with D disks. More specifically, the bowl bottom is the load percentage of the $\lfloor \frac{D}{2} \rfloor$-th disk. We will show that the maximum number of random probing trials, K, can be estimated given a *desired* bowl bottom, b_{des}, where $b \leq b_{des} \leq 1/D$.

The random probing phase of the SDL-RP algorithm will produce a uniform distribution where, by the law of large numbers, b_{des} is expected to be $1/D$, which is ideal. This is observed using SDL-RP when $K = \infty$. On the other hand, the SDL phase of SDL-RP results in a bowl-shaped distribution with an expected bowl bottom, b. This happens with $K = 0$ using SDL-RP. So increasing (decreasing) K will increase (decrease) b_{des}. A high K value results in a more uniform distribution, but requires more probes. Given a K value, the SDL-RP algorithm is as follows:

```
0) Allocate disks sequentially.
1) Phase 1: Perform random probing trials until:
     disk is found.  END.
     K trials have been performed.  Goto 2).
2) Phase 2: Perform double hashing until:
     disk is found.  END. (A disk will be found
     in under P probes.)
```

We can achieve the best and worst-case block distributions using $K = \infty$ and $K = 0$ respectively, but how do we compute an exact K value to achieve some given b_{des} in the range $b \ldots 1/D$? To do this, we find the probability, α, of termination (finding one disk) in the random probing phase, Phase 1, after performing K trials. If K unsuccessful random probing trials have occurred then termination will occur in Phase 2, which is equivalent to performing SDL alone. The probability of Phase 2 termination is then $1-\alpha$. Since each of these phases results in either a $1/D$ load percentage, for Phase 1, or a b load percentage, for Phase 2, we can find the expected load of the bowl bottom, $E(b)$, in Equation 8.

$$E(b) = \alpha \times \frac{1}{D} + (1 - \alpha) \times b \tag{8}$$

We substitute b_{des} for $E(b)$, since the expected bowl bottom value is our desired value, and simplify this equation into Equation 9 in order to solve for α.

$$\alpha = \frac{b_{des} - b}{\frac{1}{D} - b} \tag{9}$$

The distribution of the number of disk hits we make within K trials is approximated by a normal distribution where we expect to hit a disk on average $\mu = K \times \frac{D}{P}$ times with a standard deviation of $\sigma = \sqrt{K \times \frac{D}{P} \times (1 - \frac{D}{P})}$. This approximation is dictated by the Central Limit Theorem and becomes more accurate with a large number of iterations of the SDL-RP algorithm. In other words, we can assume a normal distribution and use normal distribution tables when trying to find home disks for a large number of blocks.

By using α from Equation 9, we can lookup its z-score using a standard z-score table for normal distributions [18]. We denote this $z(\alpha)$. The $z(\alpha)$ is the number of standard deviations between the mean value, μ, and a certain value, x. K can be found using this $z(\alpha)$ and the z-score formula in Equation 10.

$$z(\alpha) = \frac{\mu - x}{\sigma} \Longrightarrow z(\alpha) = \frac{K \times \frac{D}{P} - x}{\sqrt{K \times \frac{D}{P} \times (1 - \frac{D}{P})}} \tag{10}$$

Now we can easily solve for K since we have μ and σ. We set $x = 1$ since we want the probability, α, of 1 disk hit occurring. We can solve for K by using Equation 11.

$$K = \left(\frac{z(\alpha)\sqrt{\frac{D}{P} \times (1 - \frac{D}{P})} \pm \sqrt{z(\alpha)^2 \times \frac{D}{P} \times (1 - \frac{D}{P}) - 4 \times \frac{D}{P}}}{2 \times \frac{D}{P}} \right)^2 \tag{11}$$

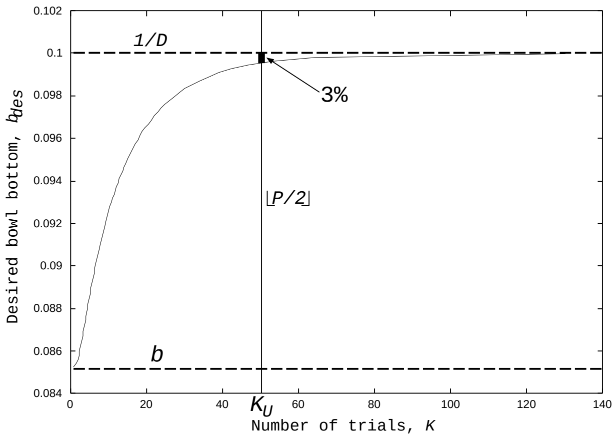

Fig. 14 The minimum K random probing trials needed to reach the desired bowl bottom, b_{des} $(D = 10, P = 101)$.

Figure 14 shows the number of trials needed to achieve a bowl bottom of b_{des}. A greater number of trials will produce a more uniform distribution. The maximum, or most uniform, b_{des} is $\frac{1}{D}$ while the minimum b_{des} is the expected value of the bowl bottom, b. In this case, we can achieve good uniformity by using a K value of $\lfloor P/2 \rfloor$ trials. We can set K to the number of trials that gives us a fairly high b_{des} which is, for example, within 3% from $1/D$ as shown in Figure 14. In other words, by fixing $b_{des} = 1/D - (0.03 \times (1/D - b))$ we can easily calculate K and we call this $K_{3\%}$. However, $K_{3\%}$ depends on D and P since these values affect the curve in Figure 14.

We show that $K_{3\%}$ decreases as we increase D and fix P to 101 in Figure 15. This is because the chance of hitting a disk, D/P, becomes greater as D increases, so fewer random probing trials are needed. Using the SDL-RP algorithm, we are not able to change the value of $K_{3\%}$ as we scale-up disks since blocks which require $K_{3\%}$ probes during the random probing phase will not

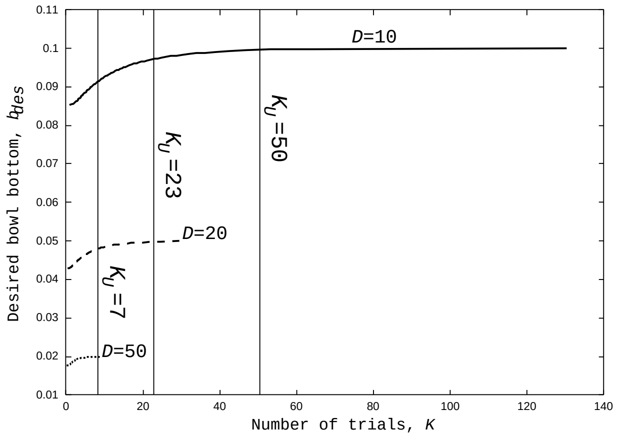

Fig. 15 The upper-bound, K_U, for $D = 10, 20$, and 50 ($P = 101$).

be found if we later use a smaller $K_{3\%}$. So, to perform disk scale-ups, we fix $K_{3\%}$ to a high value according to a low $D : P$ ratio since a high $K_{3\%}$ value can still be used as D increases.