CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data

Sage A. Weil Scott A. Brandt Ethan L. Miller Carlos Maltzahn
Storage Systems Research Center
University of California, Santa Cruz
{sage, scott, elm, carlosm}@cs.ucsc.edu

Abstract

Emerging large-scale distributed storage systems are faced with the task of distributing petabytes of data among tens or hundreds of thousands of storage devices. Such systems must evenly distribute data and workload to efficiently utilize available resources and maximize system performance, while facilitating system growth and managing hardware failures. We have developed CRUSH, a scalable pseudo-random data distribution function designed for distributed object-based storage systems that efficiently maps data objects to storage devices without relying on a central directory. Because large systems are inherently dynamic, CRUSH is designed to facilitate the addition and removal of storage while minimizing unnecessary data movement. The algorithm accommodates a wide variety of data replication and reliability mechanisms and distributes data in terms of user-defined policies that enforce separation of replicas across failure domains.

1 Introduction

Object-based storage is an emerging architecture that promises improved manageability, scalability, and performance [Azagury et al. 2003]. Unlike conventional block-based hard drives, object-based storage devices (OSDs) manage disk block allocation internally, exposing an interface that allows others to read and write to variably-sized, named objects. In such a system, each file's data is typically striped across a relatively small number of named objects distributed throughout the storage cluster. Objects are replicated across multiple devices (or employ some other data redundancy scheme) in order to protect against data loss in the presence of failures. Object-based storage systems simplify data layout by replacing large block lists with small object lists and distributing the low-level block allocation problem. Although this vastly improves scalability by reducing file allocation metadata and complexity, the fundamental task of distributing data among thousands of storage devices—typically with varying capacities and performance characteristics—remains.

Most systems simply write new data to underutilized devices. The fundamental problem with this approach is that data is rarely, if ever, moved once it is written. Even a perfect distribution will become imbalanced when the storage system is expanded, because new disks either sit empty or contain only new data. Either old or new disks may be busy, depending on the system workload, but only the rarest of conditions will utilize both equally to take full advantage of available resources.

A robust solution is to distribute all data in a system randomly among available storage devices. This leads to a probabilistically balanced distribution and uniformly mixes old and new data together. When new storage is added, a random sample of existing data is migrated onto new storage devices to restore balance. This approach has the critical advantage that, on average, all devices will be similarly loaded, allowing the system to perform well under any potential workload [Santos et al. 2000]. Furthermore, in a large storage system, a single large file will be randomly distributed across a large set of available devices, providing a high level of parallelism and aggregate bandwidth. However, simple hash-based distribution fails to cope with changes in the number of devices, incurring a massive reshuffling of data. Further, existing randomized distribution schemes that decluster replication by spreading each disk's replicas across many other devices suffer from a high probability of data loss from coincident device failures.

We have developed CRUSH (Controlled Replication Under Scalable Hashing), a pseudo-random data distribution algorithm that efficiently and robustly distributes object replicas across a heterogeneous, structured storage cluster. CRUSH is implemented as a pseudo-random, deterministic function that maps an input value, typically an object or object group identifier, to a list of devices on which to store object replicas. This differs from conventional approaches in that data placement does not rely on any sort of per-file or per-object directory—CRUSH needs only a compact, hierarchical description of the devices comprising the storage cluster and knowledge of the replica placement policy. This approach has two key advantages: first, it is completely distributed such that any party in a large system can independently calculate the location of any object; and second, what

SC2006 November 2006, Tampa, Florida, USA
0-7695-2700-0/06 $20.00 2006 IEEE

little metadata is required is mostly static, changing only when devices are added or removed.

CRUSH is designed to optimally distribute data to utilize available resources, efficiently reorganize data when storage devices are added or removed, and enforce flexible constraints on object replica placement that maximize data safety in the presence of coincident or correlated hardware failures. A wide variety of data safety mechanisms are supported, including *n*-way replication (mirroring), RAID parity schemes or other forms of erasure coding, and hybrid approaches (*e. g.*, RAID-10). These features make CRUSH ideally suited for managing object distribution in extremely large (multi-petabyte) storage systems where scalability, performance, and reliability are critically important.

2 Related Work

Object-based storage has recently garnered significant interest as a mechanism for improving the scalability of storage systems. A number of research and production file systems have adopted an object-based approach, including the seminal NASD file system [Gobioff et al. 1997], the Panasas file system [Nagle et al. 2004], Lustre [Braam 2004], and others [Rodeh and Teperman 2003; Ghemawat et al. 2003]. Other block-based distributed file systems like GPFS [Schmuck and Haskin 2002] and Federated Array of Bricks (FAB) [Saito et al. 2004] face a similar data distribution challenge. In these systems a semi-random or heuristic-based approach is used to allocate new data to storage devices with available capacity, but data is rarely relocated to maintain a balanced distribution over time. More importantly, all of these systems locate data via some sort of metadata directory, while CRUSH relies instead on a compact cluster description and deterministic mapping function. This distinction is most significant when writing data, as systems utilizing CRUSH can calculate any new data's storage target without consulting a central allocator. The Sorrento [Tang et al. 2004] storage system's use of consistent hashing [Karger et al. 1997] most closely resembles CRUSH, but lacks support for controlled weighting of devices, a well-balanced distribution of data, and failure domains for improving data safety.

Although the data migration problem has been studied extensively in the context of systems with explicit allocation maps [Anderson et al. 2001; Anderson et al. 2002], such approaches have heavy metadata requirements that functional approaches like CRUSH avoid. Choy, *et al.* [1996] describe algorithms for distributing data over disks which move an optimal number of objects as disks are added, but do not support weighting, replication, or disk removal. Brinkmann, *et al.* [2000] use hash functions to distribute data to a heterogeneous but static cluster. SCADDAR [Goel et al. 2002] addresses the addition and removal of storage, but only supports a constrained subset of replication strategies. None of these approaches include CRUSH's flexibility or failure domains for improved reliability.

CRUSH most closely resembles the RUSH [Honicky and Miller 2004] family of algorithms upon which it is based. RUSH remains the only existing set of algorithms in the literature that utilizes a mapping function in place of explicit metadata and supports the efficient addition and removal of weighted devices. Despite these basic properties, a number of issues make RUSH an insufficient solution in practice. CRUSH fully generalizes the useful elements of RUSH_P and RUSH_T while resolving previously unaddressed reliability and replication issues, and offering improved performance and flexibility.

3 The CRUSH algorithm

The CRUSH algorithm distributes data objects among storage devices according to a per-device weight value, approximating a uniform probability distribution. The distribution is controlled by a hierarchical *cluster map* representing the available storage resources and composed of the logical elements from which it is built. For example, one might describe a large installation in terms of rows of server cabinets, cabinets filled with disk shelves, and shelves filled with storage devices. The data distribution policy is defined in terms of *placement rules* that specify how many replica targets are chosen from the cluster and what restrictions are imposed on replica placement. For example, one might specify that three mirrored replicas are to be placed on devices in different physical cabinets so that they do not share the same electrical circuit.

Given a single integer input value x, CRUSH will output an ordered list $\vec{R}$ of n distinct storage targets. CRUSH utilizes a strong multi-input integer hash function whose inputs include x, making the mapping completely deterministic and independently calculable using only the cluster map, placement rules, and x. The distribution is pseudo-random in that there is no apparent correlation between the resulting output from similar inputs or in the items stored on any storage device. We say that CRUSH generates a *declustered* distribution of replicas in that the set of devices sharing replicas for one item also appears to be independent of all other items.

3.1 Hierarchical Cluster Map

The cluster map is composed of *devices* and *buckets*, both of which have numerical identifiers and weight values associated with them. Buckets can contain any number of devices or other buckets, allowing them to form interior nodes in a storage hierarchy in which devices are always at the leaves. Storage devices are assigned weights by the administrator to control the relative amount of data they are responsible for storing. Although a large system will likely contain devices with a variety of capacity and performance characteristics, randomized data distributions statistically correlate device

utilization with workload, such that device load is on average proportional to the amount of data stored. This results in a one-dimensional placement metric, weight, which should be derived from the device's capabilities. Bucket weights are defined as the sum of the weights of the items they contain.

Buckets can be composed arbitrarily to construct a hierarchy representing available storage. For example, one might create a cluster map with "shelf" buckets at the lowest level to represent sets of identical devices as they are installed, and then combine shelves into "cabinet" buckets to group together shelves that are installed in the same rack. Cabinets might be further grouped into "row" or "room" buckets for a large system. Data is placed in the hierarchy by recursively selecting nested bucket items via a pseudo-random hash-like function. In contrast to conventional hashing techniques, in which any change in the number of target bins (devices) results in a massive reshuffling of bin contents, CRUSH is based on four different bucket types, each with a different selection algorithm to address data movement resulting from the addition or removal of devices and overall computational complexity.

3.2 Replica Placement

CRUSH is designed to distribute data uniformly among weighted devices to maintain a statistically balanced utilization of storage and device bandwidth resources. The placement of replicas on storage devices in the hierarchy can also have a critical effect on data safety. By reflecting the underlying physical organization of the installation, CRUSH can model—and thereby address—potential sources of correlated device failures. Typical sources include physical proximity, a shared power source, and a shared network. By encoding this information into the cluster map, CRUSH placement policies can separate object replicas across different failure domains while still maintaining the desired distribution. For example, to address the possibility of concurrent failures, it may be desirable to ensure that data replicas are on devices in different shelves, racks, power supplies, controllers, and/or physical locations.

In order to accommodate the wide variety of scenarios in which CRUSH might be used, both in terms of data replication strategies and underlying hardware configurations, CRUSH defines *placement rules* for each replication strategy or distribution policy employed that allow the storage system or administrator to specify exactly how object replicas are placed. For example, one might have a rule selecting a pair of targets for 2-way mirroring, one for selecting three targets in two different data centers for 3-way mirroring, one for RAID-4 over six storage devices, and so on[1].

Each rule consists of a sequence of operations applied to the hierarchy in a simple execution environment, presented as pseudocode in Algorithm 1. The integer input to the CRUSH function, x, is typically an object name or other identifier, such as an identifier for a group of objects whose replicas will be placed on the same devices. The *take(a)* operation selects an item (typically a bucket) within the storage hierarchy and assigns it to the vector $\vec{i}$, which serves as an input to subsequent operations. The *select(n,t)* operation iterates over each element $i \in \vec{i}$, and chooses n distinct items of type t in the subtree rooted at that point. Storage devices have a known, fixed type, and each bucket in the system has a

Algorithm 1 CRUSH placement for object x

```
1: procedure TAKE(a)                          ▷ Put item a in working vector i
2:     i ← [a]
3: end procedure

4: procedure SELECT(n,t)                      ▷ Select n items of type t
5:     o ← ∅                                  ▷ Our output, initially empty
6:     for i ∈ i do                           ▷ Loop over input i
7:         f ← 0                              ▷ No failures yet
8:         for r ← 1,n do                     ▷ Loop over n replicas
9:             f_r ← 0                        ▷ No failures on this replica
10:            retry_descent ← false
11:            repeat
12:                b ← bucket(i)              ▷ Start descent at bucket i
13:                retry_bucket ← false
14:                repeat
15:                    if "first n" then      ▷ See Section 3.2.2
16:                        r' ← r + f
17:                    else
18:                        r' ← r + f_r n
19:                    end if
20:                    o ← b.c(r',x)          ▷ See Section 3.4
21:                    if type(o) ≠ t then
22:                        b ← bucket(o)      ▷ Continue descent
23:                        retry_bucket ← true
24:                    else if o ∈ o or failed(o) or overload(o,x)
   then
25:                        f_r ← f_r + 1, f ← f + 1
26:                        if o ∈ o and f_r < 3 then
27:                            retry_bucket ← true      ▷ Retry
   collisions locally (see Section 3.2.1)
28:                        else
29:                            retry_descent ← true     ▷ Otherwise
   retry descent from i
30:                        end if
31:                    end if
32:                until ¬retry_bucket
33:            until ¬retry_descent
34:            o ← [o,o]                      ▷ Add o to output o
35:        end for
36:    end for
37:    i ← o                                  ▷ Copy output back into i
38: end procedure

39: procedure EMIT                            ▷ Append working vector i to result
40:    R ← [R,i]
41: end procedure
```

[1] Although a wide variety of data redundancy mechanisms are possible, for simplicity we will refer to the data objects being stored as *replicas*, without any loss of generality.

Action	Resulting $\vec{i}$
take(root)	root
select(1,row)	row2
select(3,cabinet)	cab21 cab23 cab24
select(1,disk)	disk2107 disk2313 disk2437
emit	

Table 1: A simple rule that distributes three replicas across three cabinets in the same row.

type field that is used to distinguish between classes of buckets (*e. g.*, those representing "rows" and those representing "cabinets"). For each $i \in \vec{i}$, the *select(n,t)* call iterates over the $r \in 1,\ldots,n$ items requested and recursively descends through any intermediate buckets, pseudo-randomly selecting a nested item in each bucket using the function $c(r,x)$ (defined for each kind of bucket in Section 3.4), until it finds an item of the requested type t. The resulting $n|\vec{i}|$ distinct items are placed back into the input $\vec{i}$ and either form the input for a subsequent *select(n,t)* or are moved into the result vector with an *emit* operation.

As an example, the rule defined in Table 1 begins at the root of the hierarchy in Figure 1 and with the first *select(1,row)* chooses a single bucket of type "row" (it selects *row*2). The subsequent *select(3,cabinet)* chooses three distinct cabinets nested beneath the previously selected *row*2 ($cab21, cab23, cab24$), while the final *select(1,disk)* iterates over the three cabinet buckets in the input vector and chooses a single disk nested beneath each of them. The final result is three disks spread over three cabinets, but all in the same row. This approach thus allows replicas to be simultaneously separated across and constrained within container types (*e. g.* rows, cabinets, shelves), a useful property for both reliability and performance considerations. Rules consisting of multiple *take*, *emit* blocks allow storage targets to be explicitly drawn from different pools of storage, as might be expected in remote replication scenarios (in which one replica is stored at a remote site) or tiered installations (*e. g.*, fast, near-line storage and slower, higher-capacity arrays).

3.2.1 Collisions, Failure, and Overload

The *select(n,t)* operation may traverse many levels of the storage hierarchy in order to locate n distinct items of the specified type t nested beneath its starting point, a recursive process partially parameterized by $r = 1,\ldots,n$, the replica number being chosen. During this process, CRUSH may reject and reselect items using a modified input r' for three different reasons: if an item has already been selected in the current set (a collision—the *select(n,t)* result must be distinct), if a device is *failed*, or if a device is *overloaded*. Failed or overloaded devices are marked as such in the cluster map, but left in the hierarchy to avoid unnecessary shifting of data. CRUSH's selectively diverts a fraction of an overloaded device's data by pseudo-randomly rejecting with the probability specified in the cluster map—typically related to its reported over-utilization. For failed or overloaded devices, CRUSH uniformly redistributes items across the storage cluster by restarting the recursion at the beginning of the $select(n,t)$ (see Algorithm 1 line 11). In the case of collisions, an alternate r' is used first at inner levels of the recursion to attempt a local search (see Algorithm 1 line 14) and avoid skewing the overall data distribution away from subtrees where collisions are more probable (*e. g.*, where buckets are smaller than n).

3.2.2 Replica Ranks

Parity and erasure coding schemes have slightly different placement requirements than replication. In primary copy replication schemes, it is often desirable after a failure for a previous replica target (that already has a copy of the data) to become the new primary. In such situations, CRUSH can use the "first n" suitable targets by reselecting using $r' = r + f$, where f is the number of failed placement attempts by the current *select(n,t)* (see Algorithm 1 line 16). With parity and erasure coding schemes, however, the rank or position of a storage device in the CRUSH output is critical because each target stores different bits of the data object. In particular, if a storage device fails, it should be replaced in CRUSH's output list $\vec{R}$ *in place*, such that other devices in the list retain the same rank (*i. e.* position in $\vec{R}$, see Figure 2). In such cases, CRUSH reselects using $r' = r + f_r n$, where f_r is the number of failed attempts on r, thus defining a sequence of candidates for each replica rank that are probabilistically independent of others' failures. In contrast, RUSH has no special handling of failed devices; like other existing hashing distribution functions, it implicitly assumes the use of a "first n" approach to skip over failed devices in the result, making it unweildly for parity schemes.

3.3 Map Changes and Data Movement

A critical element of data distribution in a large file system is the response to the addition or removal of storage resources. CRUSH maintains a uniform distribution of data and workload at all times in order to avoid load asymmetries and the related underutilization of available resources. When an individual device fails, CRUSH flags the device but leaves it in the hierarchy, where it will be rejected and its contents uniformly redistributed by the placement algorithm (see Section 3.2.1). Such cluster map changes result in an optimal (minimum) fraction, w_{failed}/W (where W is the total weight of all devices), of total data to be remapped to new storage targets because only data on the failed device is moved.

The situation is more complex when the cluster hierarchy is modified, as with the addition or removal of storage resources. The CRUSH mapping process, which uses the cluster map as a weighted hierarchical decision tree, can result in additional data movement beyond the theoretical optimum

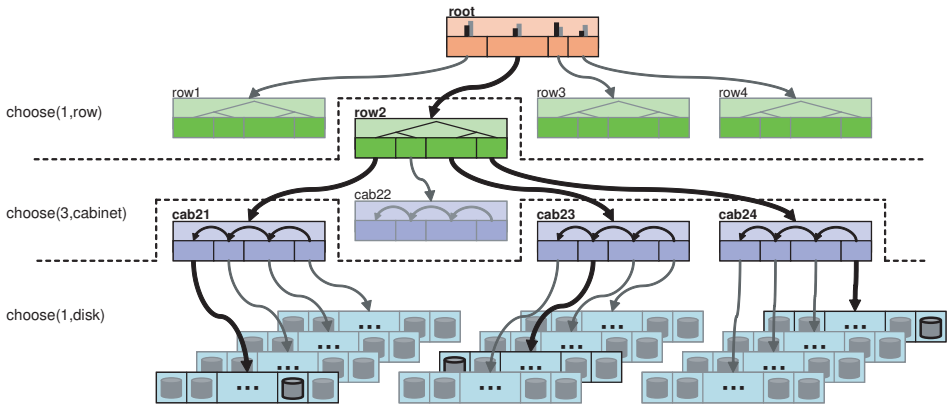

Figure 1: A partial view of a four-level cluster map hierarchy consisting of rows, cabinets, and shelves of disks. Bold lines illustrate items selected by each *select* operation in the placement rule and fictitious mapping described by Table 1.

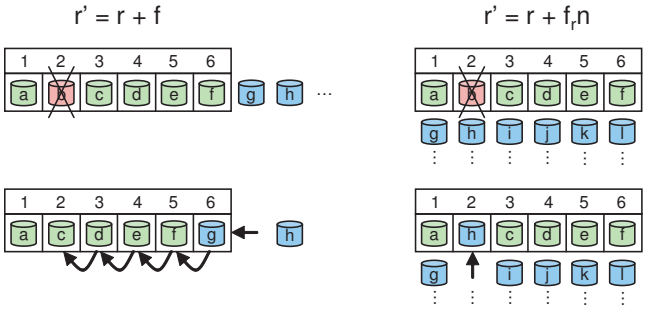

Figure 2: Reselection behavior of *select(6,disk)* when device $r = 2$ (b) is rejected, where the boxes contain the CRUSH output $\vec{R}$ of $n = 6$ devices numbered by rank. The left shows the "first n" approach in which device ranks of existing devices (c, d, e, f) may shift. On the right, each rank has a probabilistically independent sequence of potential targets; here $f_r = 1$, and $r' = r + f_r n = 8$ (device h).

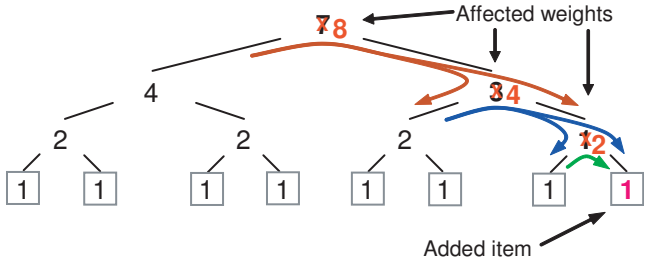

Figure 3: Data movement in a binary hierarchy due to a node addition and the subsequent weight changes.

of $\frac{\Delta w}{W}$. At each level of the hierarchy, when a shift in relative subtree weights alters the distribution, some data objects must move from from subtrees with decreased weight to those with increased weight. Because the pseudo-random placement decision at each node in the hierarchy is statistically independent, data moving into a subtree is uniformly redistributed beneath that point, and does not necessarily get remapped to the leaf item ultimately responsible for the weight change. Only at subsequent (deeper) levels of the placement process does (often different) data get shifted to maintain the correct overall relative distributions. This general effect is illustrated in the case of a binary hierarchy in Figure 3.

The amount of data movement in a hierarchy has a lower bound of $\frac{\Delta w}{W}$, the fraction of data that would reside on a newly added device with weight Δw. Data movement increases with the height h of the hierarchy, with a conservative asymptotic upper bound of $h\frac{\Delta w}{W}$. The amount of movement approaches this upper bound when Δw is small relative to W, because data objects moving into a subtree at each step of the recursion have a very low probability of being mapped to an item with a small relative weight.

3.4 Bucket Types

Generally speaking, CRUSH is designed to reconcile two competing goals: efficiency and scalability of the mapping algorithm, and minimal data migration to restore a balanced distribution when the cluster changes due to the addition or removal of devices. To this end, CRUSH defines four different kinds of buckets to represent internal (non-leaf) nodes in the cluster hierarchy: *uniform buckets*, *list buckets*, *tree buckets*, and *straw buckets*. Each bucket type is based on a different internal data structure and utilizes a different function $c(r,x)$ for pseudo-randomly choosing nested items during the replica placement process, representing a different tradeoff between computation and reorganization efficiency. Uniform buckets are restricted in that they must contain items that are all of the same weight (much like a conventional hash-based distribution function), while the other bucket types can contain a mix of items with any combination of weights. These differences are summarized in

Action	Uniform	List	Tree	Straw
Speed	O(1)	O(n)	O(log n)	O(n)
Additions	poor	optimal	good	optimal
Removals	poor	poor	good	optimal

Table 2: Summary of mapping speed and data reorganization efficiency of different bucket types when items are added to or removed from a bucket.

Table 2.

3.4.1 Uniform Buckets

Devices are rarely added individually in a large system. Instead, new storage is typically deployed in blocks of identical devices, often as an additional shelf in a server rack or perhaps an entire cabinet. Devices reaching their end of life are often similarly decommissioned as a set (individual failures aside), making it natural to treat them as a unit. CRUSH uniform buckets are used to represent an identical set of devices in such circumstances. The key advantage in doing so is performance related: CRUSH can map replicas into uniform buckets in constant time. In cases where the uniformity restrictions are not appropriate, other bucket types can be used.

Given a CRUSH input value of x and a replica number r, we choose an item from a uniform bucket of size m using the function $c(r,x) = (\text{hash}(x) + rp) \mod m$, where p is a randomly (but deterministically) chosen prime number greater than m. For any $r \leq m$ we can show that we will always select a distinct item using a few simple number theory lemmas.[2] For $r > m$ this guarantee no longer holds, meaning two different replicas r with the same input x may resolve to the same item. In practice, this means nothing more than a non-zero probability of collisions and subsequent backtracking by the placement algorithm (see Section 3.2.1).

If the size of a uniform bucket changes, there is a complete reshuffling of data between devices, much like conventional hash-based distribution strategies.

3.4.2 List Buckets

List buckets structure their contents as a linked list, and can contain items with arbitrary weights. To place a replica, CRUSH begins at the head of the list with the most recently added item and compares its weight to the sum of all remaining items' weights. Depending on the value of $\text{hash}(x,r,item)$, either the current item is chosen with the appropriate probability, or the process continues recursively down the list. This approach, derived from RUSH$_P$, recasts the placement question into that of "most recently added item, or older items?" This is a natural and intuitive choice for an expanding cluster: either an object is relocated to the newest device with some appropriate probability, or it remains on the older devices as before. The result is optimal data migration when items are added to the bucket. Items removed from the middle or tail of the list, however, can result in a significant amount of unnecessary movement, making list buckets most suitable for circumstances in which they never (or very rarely) shrink.

[2]The Prime Number Theorem for Arithmetic Progressions [Granville 1993] can be used to further show that this function will distribute replicas of object x in $m\phi(m)$ different arrangements, and that each arrangement is equally likely. $\phi(\cdot)$ is the Euler Totient function.

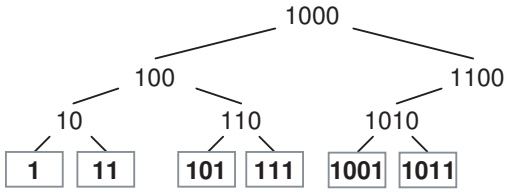

Figure 4: Node labeling strategy used for the binary tree comprising each tree bucket.

The RUSH$_P$ algorithm is approximately equivalent to a two-level CRUSH hierarchy consisting of a single list bucket containing many uniform buckets. Its fixed cluster representation precludes the use for placement rules or CRUSH failure domains for controlling data placement for enhanced reliability.

3.4.3 Tree Buckets

Like any linked list data structure, list buckets are efficient for small sets of items but may not be appropriate for large sets, where their $O(n)$ running time may be excessive. Tree buckets, derived from RUSH$_T$, address this problem by storing their items in a binary tree. This reduces the placement time to $O(\log n)$, making them suitable for managing much larger sets of devices or nested buckets. RUSH$_T$ is equivalent to a two-level CRUSH hierarchy consisting of a single tree bucket containing many uniform buckets.

Tree buckets are structured as a weighted binary search tree with items at the leaves. Each interior node knows the total weight of its left and right subtrees and is labeled according to a fixed strategy (described below). In order to select an item within a bucket, CRUSH starts at the root of the tree and calculates the hash of the input key x, replica number r, the bucket identifier, and the label at the current tree node (initially the root). The result is compared to the weight ratio of the left and right subtrees to decide which child node to visit next. This process is repeated until a leaf node is reached, at which point the associated item in the bucket is chosen. Only $\log n$ hashes and node comparisons are needed to locate an item.

The bucket's binary tree nodes are labeled with binary values using a simple, fixed strategy designed to avoid label changes when the tree grows or shrinks. The leftmost leaf in the tree is always labeled "1." Each time the tree is expanded, the old root becomes the new root's left child, and

the new root node is labeled with the old root's label shifted one bit to the left (1, 10, 100, etc.). The labels for the right side of the tree mirror those on the left side except with a "1" prepended to each value. A labeled binary tree with six leaves is shown in Figure 4. This strategy ensures that as new items are added to (or removed from) the bucket and the tree grows (or shrinks), the path taken through the binary tree for any existing leaf item only changes by adding (or removing) additional nodes at the root, at the beginning of the placement decision tree. Once an object is placed in a particular subtree, its final mapping will depend only on the weights and node labels within that subtree and will not change as long as that subtree's items remain fixed. Although the hierarchical decision tree introduces some additional data migration between nested items, this strategy keeps movement to a reasonable level, while offering efficient mapping even for very large buckets.

3.4.4 Straw Buckets

List and tree buckets are structured such that a limited number of hash values need to be calculated and compared to weights in order to select a bucket item. In doing so, they divide and conquer in a way that either gives certain items precedence (*e. g.*, those at the beginning of a list) or obviates the need to consider entire subtrees of items at all. That improves the performance of the replica placement process, but can also introduce suboptimal reorganization behavior when the contents of a bucket change due an addition, removal, or re-weighting of an item.

The straw bucket type allows all items to fairly "compete" against each other for replica placement through a process analogous to a draw of straws. To place a replica, a straw of random length is drawn for each item in the bucket. The item with the longest straw wins. The length of each straw is initially a value in a fixed range, based on a hash of the CRUSH input x, replica number r, and bucket item i. Each straw length is scaled by a factor $f(w_i)$ based on the item's weight[3] so that heavily weighted items are more likely to win the draw, *i. e.* $c(r,x) = \max_i(f(w_i)\text{hash}(x,r,i))$. Although this process is almost twice as slow (on average) than a list bucket and even slower than a tree bucket (which scales logarithmically), straw buckets result in optimal data movement between nested items when modified.

The choice of bucket type can be guided based on expected cluster growth patterns to trade mapping function computation for data movement efficiency where it is appropriate to do so. When buckets are expected to be fixed (*e. g.*, a shelf of identical disks), uniform buckets are fastest. If a bucket is only expected to expand, list buckets provide optimal data movement when new items are added at the head of the list. This allows CRUSH to divert exactly as much data to the new device as is appropriate, without any shuffle between other bucket items. The downside is $O(n)$ mapping speed and extra data movement when older items are removed or reweighted. In circumstances where removal is expected and reorganization efficiency is critical (*e. g.*, near the root of the storage hierarchy), straw buckets provide optimal migration behavior between subtrees. Tree buckets are an all around compromise, providing excellent performance and decent reorganization efficiency.

[3] Although a simple closed form for $f(w_i)$ is not known, it is relatively straightforward to calculate the weight factors procedurally (source code is available). This calculation need only be performed each time the bucket is modified.

4 Evaluation

CRUSH is based on a wide variety of design goals including a balanced, weighted distribution among heterogeneous storage devices, minimal data movement due to the addition or removal of storage (including individual disk failures), improved system reliability through the separation of replicas across failure domains, and a flexible cluster description and rule system for describing available storage and distributing data. We evaluate each of these behaviors under expected CRUSH configurations relative to RUSH$_P$- and RUSH$_T$-style clusters by simulating the allocation of objects to devices and examining the resulting distribution. RUSH$_P$ and RUSH$_T$ are generalized by a two-level CRUSH hierarchy with a single list or tree bucket (respectively) containing many uniform buckets. Although RUSH's fixed cluster representation precludes the use of placement rules or the separation of replicas across failure domains (which CRUSH uses to improve data safety), we consider its performance and data migration behavior.

4.1 Data Distribution

CRUSH's data distribution should appear random—uncorrelated to object identifiers x or storage targets—and result in a balanced distribution across devices with equal weight. We empirically measured the distribution of objects across devices contained in a variety of bucket types and compared the variance in device utilization to the binomial probability distribution, the theoretical behavior we would expect from a perfectly uniform random process. When distributing n objects with probability $p_i = \frac{w_i}{W}$ of placing each object on a given device i, the expected device utilization predicted by the corresponding binomial $b(n,p)$ is $\mu = np$ with a standard deviation of $\sigma = \sqrt{np(1-p)}$. In a large system with many devices, we can approximate $1-p \simeq 1$ such that the standard deviation is $\sigma \simeq \sqrt{\mu}$—that is, utilizations are most even when the number of data objects is large.[4] As expected, we found that the CRUSH distribution consistently matched the mean and variance of a binomial for both homogeneous clusters and clusters with mixed device weights.

[4] The binomial distribution is approximately Gaussian when there are many objects (*i. e.* when n is large).

4.1.1 Overload Protection

Although CRUSH achieves good balancing (a low variance in device utilization) for large numbers of objects, as in any stochastic process this translates into a non-zero probability that the allocation on any particular device will be significantly larger than the mean. Unlike existing probabilistic mapping algorithms (including RUSH), CRUSH includes a per-device overload correction mechanism that can redistribute any fraction of a device's data. This can be used to scale back a device's allocation proportional to its overutilization when it is in danger of overfilling, selectively "leveling off" overfilled devices. When distributing data over a 1000-device cluster at 99% capacity, we found that CRUSH mapping execution times increase by less than 20% despite overload adjustments on 47% of the devices, and that the variance decreased by a factor of four (as expected).

4.1.2 Variance and Partial Failure

Prior research [Santos et al. 2000] has shown that randomized data distribution offers real-world system performance comparable to (but slightly slower than) that of careful data striping. In our own performance tests of CRUSH as part of a distributed object-based storage system [Weil et al. 2006], we found that randomizing object placement resulted in an approximately 5% penalty in write performance due to variance in the OSD workloads, related in turn to the level of variation in OSD utilizations. In practice, however, such variance is primarily only relevant for homogeneous workloads (usually writes) where a careful striping strategy is effective. More often, workloads are mixed and already appear random when they reach the disk (or at least uncorrelated to on-disk layout), resulting in a similar variance in device workloads and performance (despite careful layout), and similarly reduced aggregate throughput. We find that CRUSH's lack of metadata and robust distribution in the face of any potential workload far outweigh the small performance penalty under a small set of workloads.

This analysis assumes that device capabilities are more or less static over time. Experience with real systems suggests, however, that performance in distributed storage systems is often dragged down by a small number of slow, overloaded, fragmented, or otherwise poorly performing devices. Traditional, explicit allocation schemes can manually avoid such problem devices, while hash-like distribution functions typically cannot. CRUSH allows degenerate devices to be treated as a "partial failure" using the existing overload correction mechanism, diverting an appropriate amount of data and workload to avoiding such performance bottlenecks and correct workload imbalance over time.

Fine-grained load balancing by the storage system can further mitigate device workload variance by distributing the read workload over data replicas, as demonstrated by the D-SPTF algorithm [Lumb et al. 2004]; such approaches, although complementary, fall outside the scope of the CRUSH mapping function and this paper.

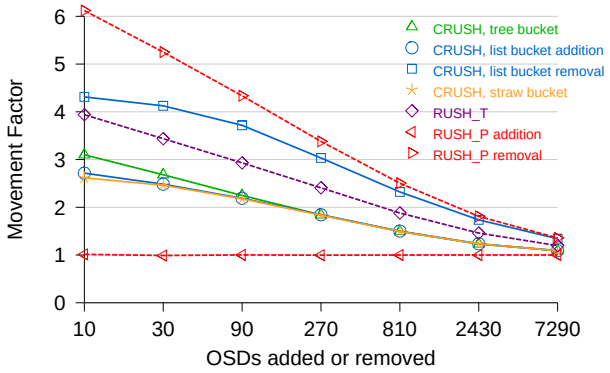

Figure 5: Efficiency of reorganization after adding or removing storage devices two levels deep into a four level, 7290 device CRUSH cluster hierarchy, versus RUSH$_P$ and RUSH$_T$. 1 is optimal.

4.2 Reorganization and Data Movement

We evaluate the data movement caused by the addition or removal of storage when using both CRUSH and RUSH on a cluster of 7290 devices. The CRUSH clusters are four levels deep: nine rows of nine cabinets of nine shelves of ten storage devices, for a total of 7290 devices. RUSH$_T$ and RUSH$_P$ are equivalent to a two-level CRUSH map consisting of a single tree or list bucket (respectively) containing 729 uniform buckets with 10 devices each. The results are compared to the theoretically optimal amount of movement $m_{optimal} = \frac{\Delta w}{W}$, where Δw is the combined weight of the storage devices added or removed and W is the total weight of the system. Doubling system capacity, for instance, would require exactly half of the existing data to move to new devices under an optimal reorganization.

Figure 5 shows the relative reorganization efficiency in terms of the *movement factor* $m_{actual}/m_{optimal}$, where 1 represents an optimal number of objects moved and larger values mean additional movement. The X axis is the number of OSDs added or removed and the Y axis is the movement factor plotted on a log scale. In all cases, larger weight changes (relative to the total system) result in a more efficient reorganization. RUSH$_P$ (a single, large list bucket) dominated the extremes, with the least movement (optimal) for additions and most movement for removals (at a heavy performance penalty, see Section 4.3 below). A CRUSH multi-level hierarchy of list (for additions only) or straw buckets had the next least movement. CRUSH with tree buckets was slightly less efficient, but did almost 25% better than plain RUSH$_T$ (due to the slightly imbalanced 9-item binary trees in each tree bucket). Removals from a CRUSH hierarchy built with list buckets did poorly, as expected (see Section 3.3).

Figure 6 shows the reorganization efficiency of different bucket types (in isolation) when nested items are added or

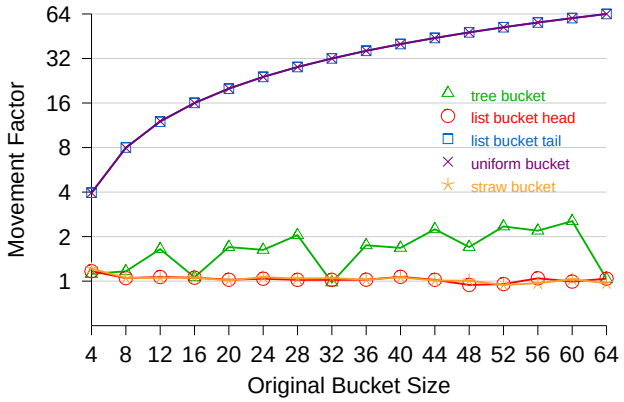

Figure 6: Efficiency of reorganization after adding items to different bucket types. 1 is optimal. Straw and list buckets are normally optimal, although removing items from the tail of a list bucket induces worst case behavior. Tree bucket changes are bounded by the logarithm of the bucket size.

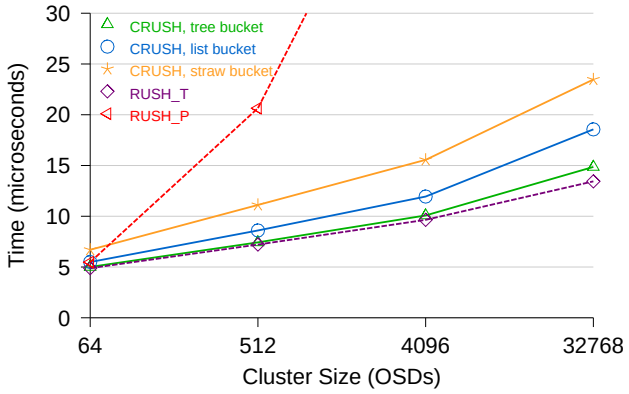

Figure 7: CRUSH and RUSH$_T$ computation times scale logarithmically relative to hierarchy size, while RUSH$_P$ scales linearly.

removed. The movement factor in a modified tree bucket is bounded by $\log n$, the depth of its binary tree. Adding items to straw and list buckets is approximately optimal. Uniform bucket modifications result in a total reshuffle of data. Modifications to the tail of a list (*e. g.*, removal of the oldest storage) similarly induce data movement proportional to the bucket size. Despite certain limitations, list buckets may be appropriate in places within an overall storage hierarchy where removals are rare and at a scale where the performance impact will be minimal. A hybrid approach combining uniform, list, tree, and straw buckets can minimize data movement under the most common reorganization scenarios while still maintaining good mapping performance.

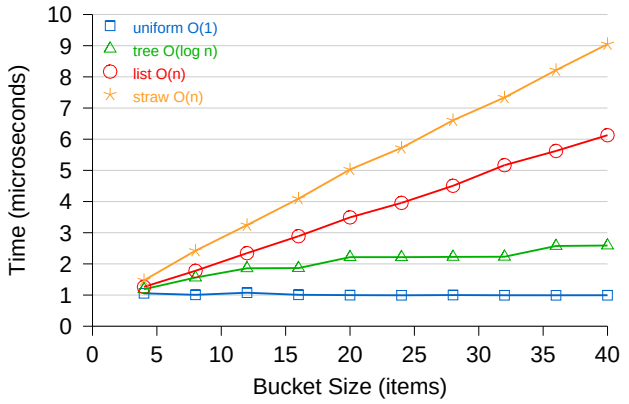

Figure 8: Low-level speed of mapping replicas into individual CRUSH buckets versus bucket size. Uniform buckets take constant time, tree buckets take logarithmic time, and list and straw buckets take linear time.

4.3 Algorithm Performance

Calculating a CRUSH mapping is designed to be fast—$O(\log n)$ for a cluster with n OSDs—so that devices can quickly locate any object or reevaluate the proper storage targets for the objects that they already store after a cluster map change. We examine CRUSH's performance relative to RUSH$_P$ and RUSH$_T$ over a million mappings into clusters of different sizes. Figure 7 shows the average time (in microseconds) to map a set of replicas into a CRUSH cluster composed entirely of 8-item tree and uniform buckets (the depth of the hierarchy is varied) versus RUSH's fixed two-level hierarchy. The X axis is the number of devices in the system, and is plotted on a log scale such that it corresponds to the depth of the storage hierarchy. CRUSH performance is logarithmic with respect to the number of devices. RUSH$_T$ edges out CRUSH with tree buckets due to slightly simpler code complexity, followed closely by list and straw buckets. RUSH$_P$ scales linearly in this test (taking more than 25 times longer than CRUSH for 32768 devices), although in practical situations where the size of newly deployed disks increases exponentially over time one can expect slightly improved sub-linear scaling [Honicky and Miller 2004]. These tests were conducted with a 2.8 GHz Pentium 4, with overall mapping times in the tens of microseconds.

The efficiency of CRUSH depends upon the depth of the storage hierarchy and on the types of buckets from which it is built. Figure 8 compares the time (Y) required for $c(r,x)$ to select a single replica from each bucket type as a function of the size of the bucket (X). At a high level, CRUSH scales as $O(\log n)$—linearly with the hierarchy depth—provided individual buckets that may be $O(n)$ (list and straw buckets scale linearly) do not exceed a fixed maximum size. When and where individual bucket types should be used depends on the expected number of additions, removals, or re-weightings. List buckets offer a slight performance advantage over straw buckets, although when removals are possible one can ex-

pect excessive data shuffling. Tree buckets are a good choice for very large or commonly modified buckets, with decent computation and reorganization costs.

Central to CRUSH's performance—both the execution time and the quality of the results—is the integer hash function used. Pseudo-random values are calculated using a multiple input integer hash function based on Jenkin's 32-bit hash *mix* [Jenkins 1997]. In its present form, approximately 45% of the time spent in the CRUSH mapping function is spent hashing values, making the hash key to both overall speed and distribution quality and a ripe target for optimization.

4.3.1 Negligent Aging

CRUSH leaves failed devices in place in the storage hierarchy both because failure is typically a temporary condition (failed disks are usually replaced) and because it avoids inefficient data reorganization. If a storage system ages in neglect, the number of devices that are failed but not replaced may become significant. Although CRUSH will redistribute data to non-failed devices, it does so at a small performance penalty due to a higher probability of backtracking in the placement algorithm. We evaluated the mapping speed for a 1,000 device cluster while varying the percentage of devices marked as failed. For the relatively extreme failure scenario in which half of all devices are dead, the mapping calculation time increases by 71%. (Such a situation would likely be overshadowed by heavily degraded I/O performance as each devices' workload doubles.)

4.4 Reliability

Data safety is of critical importance in large storage systems, where the large number of devices makes hardware failure the rule rather than the exception. Randomized distribution strategies like CRUSH that decluster replication are of particular interest because they expand the number of peers with which any given device shares data. This has two competing and (generally speaking) opposing effects. First, recovery after a failure can proceed in parallel because smaller bits of replicated data are spread across a larger set of peers, reducing recovery times and shrinking the window of vulnerability to additional failures. Second, a larger peer group means an increased probability of a coincident second failure losing shared data. With 2-way mirroring these two factors cancel each other out, while overall data safety with more than two replicas increases with declustering [Xin et al. 2004].

However, a critical issue with multiple failures is that, in general, one cannot expect them to be independent—in many cases a single event like a power failure or a physical disturbance will affect multiple devices, and the larger peer groups associated with declustered replication greatly increase the risk of data loss. CRUSH's separation of replicas across user-defined failure domains (which does not exist with RUSH or existing hash-based schemes) is specifically designed to prevent concurrent, correlated failures from causing data loss. Although it is clear that the risk is reduced, it is difficult to quantify the magnitude of the improvement in overall system reliability in the absence of a specific storage cluster configuration and associated historical failure data to study. Although we hope to perform such a study in the future, it is beyond the scope of this paper.

5 Future Work

CRUSH is being developed as part of Ceph, a multi-petabyte distributed file system [Weil et al. 2006]. Current research includes an intelligent and reliable distributed object store based largely on the unique features of CRUSH. The primitive rule structure currently used by CRUSH is just complex enough to support the data distribution policies we currently envision. Some systems will have specific needs that can be met with a more powerful rule structure.

Although data safety concerns related to coincident failures were the primary motivation for designing CRUSH, study of real system failures is needed to determine their character and frequency before Markov or other quantitative models can used to evaluate their precise effect on a system's mean time to data loss (MTTDL).

CRUSH's performance is highly dependent on a suitably strong multi-input integer hash function. Because it simultaneously affects both algorithmic correctness—the quality of the resulting distribution—and speed, investigation into faster hashing techniques that are sufficiently strong for CRUSH is warranted.

6 Conclusions

Distributed storage systems present a distinct set of scalability challenges for data placement. CRUSH meets these challenges by casting data placement as a pseudo-random mapping function, eliminating the conventional need for allocation metadata and instead distributing data based on a weighted hierarchy describing available storage. The structure of the cluster map hierarchy can reflect the underlying physical organization and infrastructure of an installation, such as the composition of storage devices into shelves, cabinets, and rows in a data center, enabling custom placement rules that define a broad class of policies to separate object replicas into different user-defined failure domains (with, say, independent power and network infrastructure). In doing so, CRUSH can mitigate the vulnerability to correlated device failures typical of existing pseudo-random systems with declustered replication. CRUSH also addresses the risk of device overfilling inherent in stochastic approaches by selectively diverting data from overfilled devices, with minimal computational cost.

CRUSH accomplishes all of this in an exceedingly efficient fashion, both in terms of the computational effi-

ciency and the required metadata. Mapping calculations have $O(\log n)$ running time, requiring only tens of microseconds to execute with thousands of devices. This robust combination of efficiency, reliability and flexibility makes CRUSH an appealing choice for large-scale distributed storage systems.

7 Acknowledgements

R. J. Honicky's excellent work on RUSH inspired the development of CRUSH. Discussions with Richard Golding, Theodore Wong, and the students and faculty of the Storage Systems Research Center were most helpful in motivating and refining the algorithm. This work was supported in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714. Sage Weil was supported in part by a fellowship from Lawrence Livermore National Laboratory. We would also like to thank the industrial sponsors of the SSRC, including Hewlett Packard Laboratories, IBM, Intel, Microsoft Research, Network Appliance, Onstor, Rocksoft, Symantec, and Yahoo.

8 Availability

The CRUSH source code is licensed under the LGPL, and is available at:

`http://www.cs.ucsc.edu/~sage/crush`

References

ANDERSON, E., HALL, J., HARTLINE, J., HOBBS, M., KARLIN, A. R., SAIA, J., SWAMINATHAN, R., AND WILKES, J. 2001. An experimental study of data migration algorithms. In *Proceedings of the 5th International Workshop on Algorithm Engineering*, Springer-Verlag, London, UK, 145–158.

ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. 2002. Hippodrome: running circles around storage administration. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*.

AZAGURY, A., DREIZIN, V., FACTOR, M., HENIS, E., NAOR, D., RINETZKY, N., RODEH, O., SATRAN, J., TAVORY, A., AND YERUSHALMI, L. 2003. Towards an object store. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, 165–176.

BRAAM, P. J. 2004. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug.

BRINKMANN, A., SALZWEDEL, K., AND SCHEIDELER, C. 2000. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, ACM Press, 119–128. Extended Abstract.

CHOY, D. M., FAGIN, R., AND STOCKMEYER, L. 1996. Efficiently extendible mappings for balanced data distribution. *Algorithmica 16*, 215–232.

GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, ACM.

GOBIOFF, H., GIBSON, G., AND TYGAR, D. 1997. Security for network attached storage devices. Tech. Rep. TR CMU-CS-97-185, Carniege Mellon, Oct.

GOEL, A., SHAHABI, C., YAO, D. S.-Y., AND ZIMMERMAN, R. 2002. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*, 473–482.

GRANVILLE, A. 1993. On elementary proofs of the Prime Number Theorem for Arithmetic Progressions, without characters. In *Proceedings of the 1993 Amalfi Conference on Analytic Number Theory*, 157–194.

HONICKY, R. J., AND MILLER, E. L. 2004. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, IEEE.

JENKINS, R. J., 1997. Hash functions for hash table lookup. http://burtleburtle.net/bob/hash/evahash.html.

KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *ACM Symposium on Theory of Computing*, 654–663.

LUMB, C. R., GANGER, G. R., AND GOLDING, R. 2004. D-SPTF: Decentralized request distribution in brick-based storage systems. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 37–47.

NAGLE, D., SERENYI, D., AND MATTHEWS, A. 2004. The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*.

RODEH, O., AND TEPERMAN, A. 2003. zFS—a scalable distributed file system using object disks. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, 207–218.

SAITO, Y., FRØLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. 2004. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 48–58.

SANTOS, J. R., MUNTZ, R. R., AND RIBEIRO-NETO, B. 2000. Comparing random data allocation and data striping in multimedia servers. In *Proceedings of the 2000 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM Press, Santa Clara, CA, 44–55.

SCHMUCK, F., AND HASKIN, R. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, USENIX, 231–244.

TANG, H., GULBEDEN, A., ZHOU, J., STRATHEARN, W., YANG, T., AND CHU, L. 2004. A self-organizing storage cluster for parallel data-intensive applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*.

WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*.

XIN, Q., MILLER, E. L., AND SCHWARZ, T. J. E. 2004. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 172–181.