Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution

R. J. Honicky
honicky@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Storage Systems Research Center
Jack Baskin School of Engineering
University of California, Santa Cruz

Abstract

Typical algorithms for decentralized data distribution work best in a system that is fully built before it first used; adding or removing components results in either extensive reorganization of data or load imbalance in the system. We have developed a family of decentralized algorithms, RUSH *(Replication Under Scalable Hashing), that maps replicated objects to a scalable collection of storage servers or disks.* RUSH *algorithms distribute objects to servers according to user-specified server weighting. While all* RUSH *variants support addition of servers to the system, different variants have different characteristics with respect to lookup time in petabyte-scale systems, performance with mirroring (as opposed to redundancy codes), and storage server removal. All* RUSH *variants redistribute as few objects as possible when new servers are added or existing servers are removed, and all variants guarantee that no two replicas of a particular object are ever placed on the same server. Because there is no central directory, clients can compute data locations in parallel, allowing thousands of clients to access objects on thousands of servers simultaneously.*

1. Introduction

Recently, there has been significant interest in using object-based storage as a mechanism for increasing the scalability of storage systems. The storage industry has begun to develop standard protocols and models for object based storage [19], and various other major players in the industry, such as the National Laboratories have also pushed for the development of object-based storage devices (OSDs) to meet their growing demand for storage bandwidth and scalability. The OSD architecture differs from a typical storage area network (SAN) architecture in that block management is handled by the disk as opposed to a dedicated server. Because block management on individual disks requires no inter-disk communication, this redistribution of work comes at little cost in performance or efficiency, and has a huge benefit in scalability, since block layout is completely parallelized.

There are differing opinions about what object-based storage is or should be, and how much intelligence belongs in the disk. In order for a device to be an OSD, however, each disk or disk subsystem must have its own filesystem; an OSD manages its own allocation of blocks and disk layout. To store data on an OSD, a client provides the data and a key for the data. To retrieve the data, a client provides a key. In this way, an OSD has many similarities to an object database.

Large storage systems built from individual OSDs still have a scalability problem, however: where should individual objects be placed? We have developed a family of algorithms, *RUSH* (Replication Under Scalable Hashing) that addresses this problem by facilitating the distribution of multiple replicas of objects among thousands of OSDs. *RUSH* allows individual clients to compute the location of all of the replicas of a particular object in the system algorithmically using just a list of storage servers rather than relying on a directory. Equally important, a simple algorithm for the lookup and placement of data to a function from a key to a particular OSD makes it easy to support complex management functionality such as weighting for disk clusters with different characteristics and online reorganization.

2. *RUSH*: A Family of Algorithms

One important aspect of large scalable storage systems is replication. Qin, *et al.* [21] note that, without replication or another form of data protection such as erasure coding, a two petabyte storage system would have a mean time to

This paper was published in the *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, April 2004.

data loss of around one day. It is therefore important that an object placement algorithm support replication or another method of data protection.

Since our algorithms all support replication and other features necessary for truly scalable OSD-based storage systems, we have named the family of algorithms which we have developed, *Replication Under Scalable Hashing*, or *RUSH*. In fact, *RUSH* variants support something stronger: *adjustable* replication. That is, *RUSH* variants allow the degree of replication of a particular object to be adjusted online, independent of the degree of replication of other objects. Adjustable replication can be used to significantly increase the mean time to data loss in an OSD based system [21].

It is also important to note that in order for replication to be effective, it must guarantee that replicas of the same objects are placed on different disks. While some peer-to-peer systems such as OceanStore [15] do not make such guarantees, but rather use high degrees of replication to make statistical promises about the number of independent replicas, *RUSH* variants all make this guarantee. *RUSH* variants also distribute the replicas of the objects stored on a particular disk throughout the system, so that all of the disks in the system share the burden of servicing requests from a failed disk.

Since many storage systems are upgraded and expanded periodically, *RUSH* variants also support weighting, allowing disks of different vintages to be added to a system. Weighting allows a system to take advantage of the newest technology, and can also allow older technology to be retired.

Another essential characteristic of *RUSH* variants is optimal or near-optimal reorganization. When new disks are added to the system, or old disks are retired, *RUSH* variants minimize the number of objects that need to be moved in order to bring the system back into balance. This is in sharp contrast to pseudo-random placement using a traditional hash function, under which most objects need to be moved in order to bring a system back into balance. Additionally, *RUSH* variants can perform reorganization *online* without locking the filesystem for a long time to relocate data. Near-optimal reorganization is important because completely reorganizing a very large filesystem is very slow, and may take a filesystem offline for many hours. For example, a 1 petabyte file system built from 2000 disks, each with a 500 GB capacity and peak transfer rate of 25 MB/sec would require nearly 12 hours to shuffle all of the data; this would require an aggregate network bandwidth of 50 GB/s. During reorganization, the system would be unavailable. In contrast, a system running *RUSH* can reorganize online because only a small fraction of existing disk bandwidth is needed to copy data to the new disks.

RUSH variants are completely decentralized, so they require no communication except during a reorganization. As a storage system scales to tens or even hundreds of thousands of disks, decentralization of object lookup becomes more and more essential. Moreover,*RUSH* variants require very few resources to run effectively: *RUSH* variants are very fast and require minimal memory. These two features enable the algorithms to be run not only on the clients, but on the OSDs themselves, even under severely constrained memory and processing requirements. Running on the OSD can assist in fast failure recovery.

2.1. Terminology and Symbols

RUSH variants are able to offer such flexibility and performance in part because they make assumptions about the structure of the OSDs and clients. First, we assume that disks and clients are *tightly connected*, *i. e.*, disks and clients are able to communicate with each other directly, with relatively uniform latency, and with relatively high reliability. This is in contrast to *loosely connected* peer-to-peer and WAN networks in which communication latencies are longer and possibly highly varied. Our target environment is a corporate data center or scientific computing network, likely dedicated to storage traffic.

Another important assumption crucial to the functioning of *RUSH* is that disks are added to the system in homogeneous groups. A group of disks, called a *sub-cluster* have the same vintage and therefore share both performance and reliability characteristics. It is possible to add disks with different characteristics at the same time—they must merely be grouped into multiple homogeneous subclusters.

All of the *RUSH* variants are described in pseudo-code later in this section; the symbols used in the pseudo-code and the accompanying explanations are listed in Table 1.

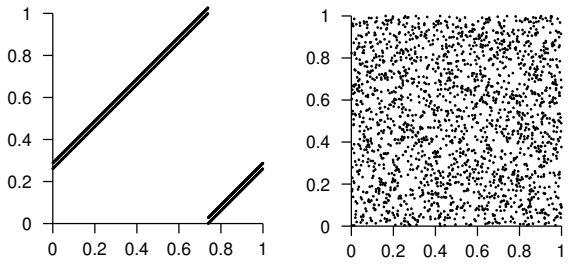
2.2. *RUSH* Concepts

There are several common features of the *RUSH* variants which combine to allow scalability, flexibility and performance. The first of these is the recognition that as large storage systems expand, new capacity is typically added several disks at a time, rather than by adding individual disks. The use of sub-clusters leads naturally to a two-part lookup process: first, determine the sub-cluster in which an object belongs, and then determine which disk in the subcluster holds that object. This two part lookup allows us to mix and match different sub-cluster mapping algorithms and different disk mapping algorithms, in order to provide the best feature set for a particular problem.

All of the *RUSH* variants are structured recursively. This recursive structure arises from the recursive nature of

Symbol	Description
x	The key of the object being located.
r	The replica ID of the object being located
j	The ID of some sub-cluster. Higher id's were added more recently
m_j	The number of disks in sub-cluster j
w_j	The (un-normalized) weight of each disk in sub-cluster j
m'_j	The total amount of weight in sub-cluster j
n'_j	The total amount of weight in all the sub-clusters added previous to sub-cluster j (i.e. sub-clusters $0 \ldots j-1$)
z	A pseudo-random number, or result of a hash function

Table 1. Symbols used in *RUSH* pseudo-code and explanations.



(a) Correlation of hash values of sequential keys (b) Correlation of a pseudo-random stream

Figure 1. Two dimensional correlation of the hash values of sequential keys and the random generator.

adding disks to an existing system: a system naturally divides into the most recently added disks, and the disks that were already in the system when they were added. *RUSH*$_P$ and *RUSH*$_R$ follow this model closely, while *RUSH*$_T$ applies a divide and conquer approach. Although the pseudo-code is recursive, real implementations may be iterative in order to avoid the function call overhead.

Probabilistic placement of objects also seems like a natural choice for *RUSH* because many hash functions offer excellent performance and even distribution of objects, . Probabilistic placement also facilitates weighting of sub-clusters and the distribution of replicas of a particular object evenly throughout the system.

All of the *RUSH* variants use a parametric hash with two additional parameters. The hash function is a simple multiplicative hash which yields values in the range $[0,1)$: $h(k) = Ak \mod 1$ where $A \in [0,1)$.

Unfortunately, a simple multiplicative hash function generates a highly correlated sequence of randomly-distributed hash values when its parameters are sequential, as shown in Figure 1(a). Since *RUSH* requires that hash values be well-distributed regardless of parameter sequentiality, we use a simple hash function on one parameter to seed a random number generator and then use the stream of resulting values as random numbers. The output of the generator is shown in Figure 1(b).

2.3. *RUSH*$_P$: Placement using Prime Numbers

RUSH$_P$ is described in detail in our previous work [10] so we will only briefly discuss this algorithm. The algorithm for *RUSH*$_P$, using symbols described in Table 1 is shown in Figure 2.

RUSH$_P$ takes a key and replica id and returns a disk id. It first decides in which sub-cluster an object replica belongs by computing the parametric hash of an object. It compares the hash value, which is in the range $[0,1)$, to the ratio of the amount of weight in the most recently added sub-cluster to the total weight in the system. If the hash value is less than the ratio of weights, the object replica belongs in the most recently added sub-cluster. Otherwise, the object must belong in one of the previously added sub-clusters, so *RUSH*$_P$ discards the most recently added sub-cluster and repeats the process.

Once *RUSH*$_P$ decides on the sub-cluster in which an object replica belongs, it places the object replica in some disk in the sub-cluster using the function $f(x,r) = z + rp(\mod m_j)$[1], where z is essentially a hash of the key x, p is a randomly chosen prime number, and m_j is the number of disks in the sub-cluster.

With a few simple number theory lemmas we can show that as long as r is less than or equal to m_j, $f(x,r_i) \neq f(x,r_k)$ for $i \neq k$. Using a more advanced analytic number theory result called the Prime Number Theorem for Arithmetic Progressions [9], we can show that this function will distribute the replicas of object x in $m_j\phi(m_j)$ different arrangements[2], and each arrangement is equally likely.

Note that, in most cases, a single sub-cluster will not contain all of the replicas of an object. However, there is a finite probability of this occurring, and it indeed will always happen if there is only one sub-cluster, so the algorithm must allow for this possibility. *RUSH*$_P$ uses a trick to allow all of the sub-clusters except for the original sub-cluster to have fewer disks than the object replication factor, as described in our original paper on *RUSH*$_P$ [10].

[1] We use the group theoretic notation for modulus, where mod has lower operator precedence that arithmetic operations and other functions, so that the modulus binds with the whole algebraic expression to the left.

[2] $\phi(\cdot)$ is the Euler Totient function.

```
def RUSH_P (x, r, j):
  m'_j ← m_j w_j
  n'_j ← Σ_{i=0}^{j-1} m'_i
  z = hash(x, j, 0) · (n'_j + m'_j)
  choose a prime number p ≥ m'_j based on hash(x, j, 1)
  v ← x + z + r × p
  z' ← (z + r × p) mod (n'_j + m'_j)
  if m_j ≥ R and z' < m'_j
    map the object to server n_j + (v mod m_j)
  else if m_j < R and z' < R · w_j and v mod R < m_j
    map the object to server n_j + (v mod R).
  else
    RUSH_P (x, r, j − 1)
```

Figure 2. Pseudo-code for $RUSH_P$.

2.4. $RUSH_R$: Support for Removal

$RUSH_R$ differs from $RUSH_P$ in that it locates all of the replicas of an object simultaneously, allowing sub-clusters to be removed or reweighted without reorganizing the entire system. $RUSH_R$ uses the same ratio that $RUSH_P$ uses in order to determine which objects go where: the ratio between the number of servers in the most recently added sub-cluster and the number of servers in the whole system. These values, however are passed separately as parameters to a draw from the hypergeometric distribution[3]. The result of the draw is the number of replicas of the object that belong in the most recently added sub-cluster, as shown in Figure 3(a).

Once the number of replicas that belong in the most recently added sub-cluster has been determined, we use a simple technique, shown in Figure 3(b), to randomly draw the appropriate number of disk identifiers.

2.5. $RUSH_T$: A Tree-Based Approach

$RUSH_T$ was devised to increase the scalability of *RUSH* by allowing computation time to scale logarithmically with the number of sub-clusters in the system. It also offers more flexibility in the ways in which a system can be reorganized. It accomplishes this with some sacrifice in the competitiveness of its reorganizations, as described in Section 3.4.

$RUSH_T$ is similar to $RUSH_P$, except that it uses a binary tree data structure rather than a list. Each node in the tree knows the total weight to the left and right of the node, and each nodes has a unique identifier which is used as a parameter to the hash function.

As with the other *RUSH* variants, $RUSH_T$ first looks up the sub-cluster in which an object replica belongs. The $RUSH_T$ algorithm, shown in Figure 4(b) accomplishes this by calculating the hash of the key of the object, parameterized by the unique index of the current node (starting with the root). $RUSH_T$ then compares this hash value to the ratio between the amount of weight to the left of the current node and the total amount of weight to the left and right of the node. If the hash value is less than this ratio, then the object belongs to the left of the current node, otherwise, it belongs to the right. This process is repeated until a sub-cluster leaf node is reached.

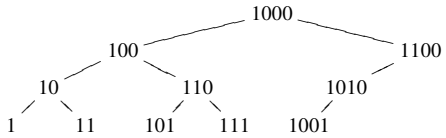
[3]We add weighting to the hypergeometric distribution to support weighting of sub-clusters.

```
def getSub − Cluster(node, x, r):
  if node.flavor = SUB − CLUSTER:
    return node
  if hash(x, node.index, r, 0) < node.left.totalWeight:
    return getSub − Cluster(node.left, x, r)
  else:
    return getSub − Cluster(node.right, x, r)
```

(a) Locating the sub-cluster in which replica r of object x is stored

```
def RUSH_T (x, r)
  node ← getSub − Cluster(root, x, r)
  j ← node.index
  v ← hash(x, j, 0, 1) · m_j
  choose a prime number p ≥ m'_j based on hash(x, j, 0, 2)
  s ← x + v + r · p( mod m_j)
  return (j, s)
```

(b) The $RUSH_T$ main algorithm



(c) A tree for a system with five sub-clusters, with nodes labeled by their binary index

Figure 4. Pseudo-code for the $RUSH_T$ algorithm.

A tree illustrating the result of this process for a system with five subclusters is show in Figure 4(c).

Once the replica's sub-cluster has been found, $RUSH_T$ then determines the specific server in the sub-cluster on which to place the object replica in the same way that $RUSH_P$ does, described in Section 2.3. The technique used in $RUSH_P$ for allowing sub-clusters to be smaller than the replication factor does not work, however, for $RUSH_T$; this is perhaps the main drawback of $RUSH_T$.

When a system gains additional sub-clusters or is otherwise reorganized, $RUSH_T$ must ensure that nodes in the tree always have the same unique identifier. This is done by allocating binary identifiers starting with the leftmost leaf (the original sub-cluster in the system), allocating that leaf the identifier "1." To add a new root node, shift the root's identifier one bit to the left, copy the identifiers from the left subtree to the right subtree, and append a "1" on the left hand side of every identifier in the right hand subtree, pruning off any unused branches.[4] A tree illustrating the

[4]We can optimize this so that only identifiers that are actually needed

```
def init()
  for j = 0 to c − 1:
    m'_j ← m_j w_j
    n'_j ← Σ_{i=0}^{j−1} m'_i
  end for

def RUSH_R (x, R, j, l)
  seed the random number generator with hash(x, j, 0)
  t ← max(0, R − n_j)
  // H is a draw from the weighted hypergeometric distribution
  u ← t + H(R − t, n'_j − t, m'_j + n'_j − t, w_j)
  if u > 0 then
    seed the random number generator with hash(x, j, 1)
    y ← choose(u, m_j)
    reset()
    add n_j to each element in y
    append y to l
    R ← R − u
  end if
  if R = 0:
    return l:
  else:
    return RUSH_R (x, R, j − 1, l)
```

(a) An algorithm for mapping all replicas of an object to corresponding servers.

```
def initChoose (n)
  a_{0...n−1} ← {0, ..., n − 1}

def choose ( k, n )
  for i = 0 to k − 1
    generate a random integer 0 ≤ z < (n − i)
    // swap a_z and a_{n−i−1}
    r_i ← a_z
    a_z ← a_{n−i−1}
    a_{n−i−1} ← r_i
  end for
  return r

def reset
  for i ∈ {0, ..., k − 1}
    c ← a_{n−k+i}
    if c < n − k:
      a_c ← c
    a_{n−k+i} ← n − k + i
  end for
```

(b) Algorithm for choosing k integers out of $\{0,\ldots,n-1\}$ without replacement.

Figure 3. Pseudo-code for the $RUSH_R$ algorithm.

result of this process for a system with five sub-clusters is show in Figure 4(c).

3. RUSH in Practice

Since the *RUSH* family of algorithms was designed to allocate data in petabyte-scale storage systems, we measured its behavior under different situations to show that *RUSH* does, indeed, meet its design goals. In this section we describe a series of experiments that illustrate various characteristics of each of the *RUSH* variants. All measurements were performed on a 2.8 GHz Pentium 4 machine.

3.1. Object Lookup Performance

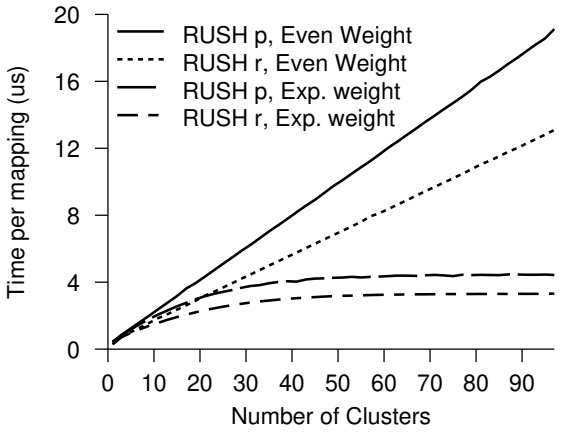
In our first experiment, we examine the performance characteristics of *RUSH*. Informally, in the worst case, $RUSH_P$ and $RUSH_R$ must iterate over every sub-cluster in the system, doing a constant amount of work in each sub-cluster, so the lookup time of $RUSH_P$ and $RUSH_R$ grows linearly with the number of sub-clusters in the system. $RUSH_T$ on the other hand, uses a binary tree to locate the correct sub-cluster, again doing constant work at each node, so lookup time is logarithmic in the number of sub-clusters in the system.

The lookup time for $RUSH_P$ and $RUSH_R$, however, also varies depending on the weighting in the system. If the most recently added disks have higher weight than previously added disks, then the lookup process will tend to stop after only examining the most recently added sub-clusters. Since newer disks will tend to have greater capacity and and throughput, we expect newer disks to have higher weight, and thus we expect the performance of $RUSH_P$ and $RUSH_R$ to be sub-linear.

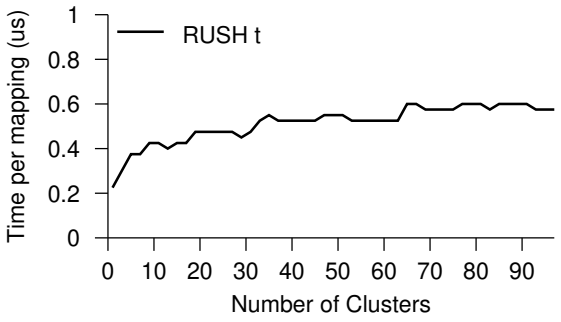
Since $RUSH_T$ must always traverse a tree regardless of the weightings of the sub-clusters in the system, weighting does not affect the performance of $RUSH_T$. This may be advantageous in systems where quality of service guarantees are important.

To perform the experiments in this section, we started with a system with only a single sub-cluster. We then looked up four million object replicas, and took the average time for a lookup. Two more sub-clusters were added, and the process repeated; this was done until we reached 100 sub-clusters. Figure 5(a) shows the performance of $RUSH_P$ and $RUSH_R$ where each sub-cluster has the same weight, and where the weight in each sub-cluster increases exponentially, by a factor of 1.1. Figure 5(b) shows the shape of the performance curve for $RUSH_T$. As expected, lookup times for $RUSH_T$ increased logarithmically with the number of sub-clusters in the system. The timing is so much faster than $RUSH_P$ and $RUSH_R$ that it does not show up on Figure 5—lookups required less than one microsecond even after 100 clusters had been added. The unevenness of the $RUSH_T$ line is due to the limited resolution of the timer on our system. If performance is an issue, mappings done by $RUSH_P$ and $RUSH_R$ can be cached because mappings

are allocated. We chose to describe this as pruning for the sake of simplicity.



(a) Lookup time for $RUSH_P$ and $RUSH_R$ under different sub-cluster weighting assumptions



(b) Lookup times for $RUSH_T$

Figure 5. Per-object-replica lookup times as the number of sub-clusters in the system varies, under two weighting scenarios.

of object replicas to disks do not change unless the system configuration is changed.

3.2. Object Distribution

In this experiment, we tested the ability of *RUSH* to distribute objects according to a distribution of weights over sub-clusters. We inserted 10,000 objects with 4 replicas each into 3 sub-clusters of 5 disks each. Each sub-cluster has twice the weight of the sub-cluster to its left.

Figure 6 shows the results of this experiment. Even with only 40,000 object replicas, all of the *RUSH* variants place objects almost exactly according to the appropriate weight. The leftmost bar in the figure shows the ideal value; none of the *RUSH* variants differ greatly from this ideal distribution.

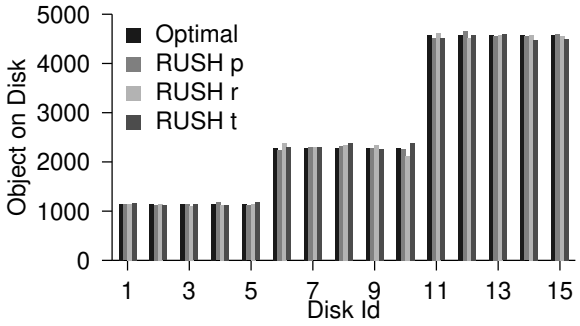


Figure 6. The distribution of object replicas over three sub-clusters with five disks each for $RUSH_P$, $RUSH_R$ and $RUSH_T$.

3.3. Failure Resilience

In this experiment, we studied the behavior of the system when a disk fails. When a disk fails, the disks which hold replicas of the objects stored on the failed disk must service requests which would have been serviced by the failed disk. In addition, these disks must devote some percentage of their resources to failure recovery once the failed disk has been replaced. We call this extra workload *failure load*.

The more evenly distributed the replicas of objects from the failed disk, the more evenly distributed the failure load. In order to minimize service degradation and cascading failures, *RUSH* distributes the replicas of objects on a particular disk over the entire system. These replicas are distributed according the weights of each sub-cluster. We call the distribution of replicas of objects stored on the failed disk the *failure distribution*.

Figure 7 shows the failure distribution for disk 8 in a system with 3 sub-clusters of 5 disks each, all evenly weighted. Both $RUSH_P$ and $RUSH_T$ show a slight tendency to favor disks that are close to the failed disk, while $RUSH_R$ shows no such tendency. For comparison, we also examined a system in which the all the replicas of an object are allocated to four adjacent disks. For example, replicas of some object might be distributed to disks 8, 9, 10, and 11. In such a system, when disk 8 fails, replicas of the objects on disk 8 could be located on disks 5, 6, 7, 9, 10 or 11. In comparison to the system that places replicas on adjacent disks, the "favoritism" showed by $RUSH_P$ and $RUSH_T$ is almost negligible. In systems with more disks and more objects, the favoritism is even less pronounced, especially when compared to a system that distributes replicas to adjacent disks regardless of how many servers are in the system.

Unfortunately, the deviation from the optimal value depends on several complex factors and is therefore difficult to quantify. In $RUSH_P$ and $RUSH_T$, the amount of the devi-

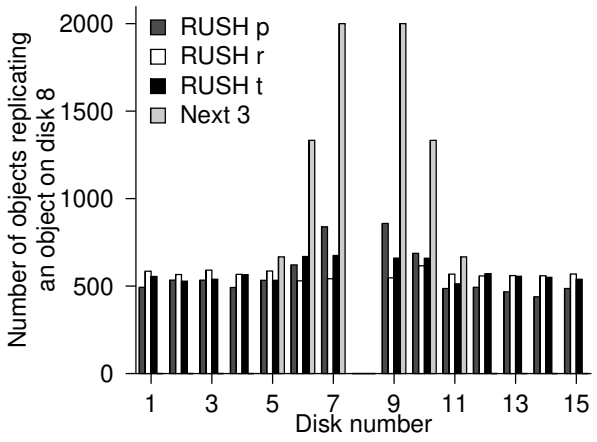


Figure 7. The distribution of replicas of objects on disk 8, which has just failed. *RUSH* **variants distribute the load under failure; Next-3 uses adjacent disks for replicas causing imbalanced load under failure.**

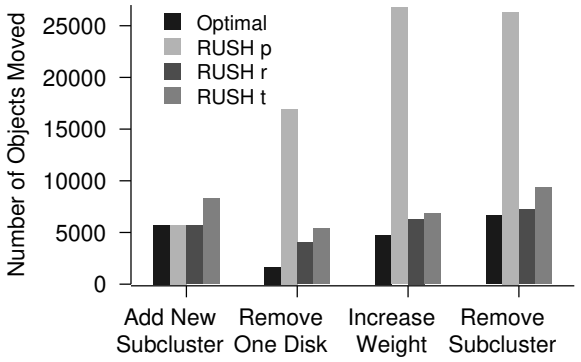


Figure 8. The number of objects which move under various reorganizations.

ation depends partly on the number unique coprimes of the size the sub-cluster in which the failure occurred, known as the Euler Totient Function. Despite this difficulty in quantifying the variance, we can see empirically that $RUSH_R$ gives the most faithful failure distribution, but that $RUSH_P$ and $RUSH_T$ are both extremely accurate. Figure 7 is representative of our findings for other system configurations.

3.4. Reorganization

The final experiments described in this paper examined the number of objects that move during four different reorganization scenarios. In the first scenario, a sub-cluster is added to the system. In the second, a single disk is removed from the system, causing the number of disks in a sub-cluster to be adjusted. In the third scenario, the weight of a sub-cluster is increased. In the fourth scenario, an entire sub-cluster is removed from the system. In each of the experiments, the system starts with six sub-clusters with 4 disks each. We added 10,000 objects, each having 4 replicas, and then reorganized the system, counting the number of objects which moved during the reorganization.

As shown in Figure 8, $RUSH_P$ and $RUSH_R$ both perform optimally when a sub-cluster is added to the system; in fact, they are theoretically optimal, so any variation is due to small variations in probabilistic distribution. $RUSH_T$ is slightly sub-optimal, but is consistently within a small constant factor of the optimal case.

RUSH was not designed for a system in which individual disks are removed, as is apparent from the second column in Figure 8. While $RUSH_R$ and $RUSH_T$ move about three times as many objects as the optimal number, $RUSH_P$ moves around ten times as many—over a third of the objects in the system! If removing a disk is necessary, and the number of objects moved is unacceptable, we can take advantage of *RUSH*'s adjustable replication factors by maintain a list of removed disks. If an object maps to any removed disk, increase its replication factor by one and look up the new replica instead. This process may be repeated until an existing disk is found. While this mechanism clearly causes an optimal number of objects to be moved, it has some limitations. For example, $RUSH_P$ and $RUSH_T$ both have a maximum replication factor. Also, worst case lookup time increases linearly with the number of removed disks. A more in depth examination of this mechanism is the subject of future research.

The third column of Figure 8 shows that $RUSH_T$ and $RUSH_R$ perform well when the weight of an entire subcluster is increased, but $RUSH_P$ again performs poorly. In both adding (column 3) and removing weight (column 4) from a sub-cluster, $RUSH_T$ and $RUSH_R$ both perform well, whereas $RUSH_P$ does not. This behavior is expected, since $RUSH_T$ and $RUSH_R$ were designed to allow changing subcluster weights, whereas $RUSH_P$ was not.

4. Applications of *RUSH*

Each *RUSH* variant excels in a different environment. While $RUSH_T$ offers the best performance and the best overall flexibility in reconfiguration, it does not allow subclusters smaller than the maximum number of a replicas that a single object may have. It also moves more than the optimal number of objects during reorganization, though the number is within a small factor of the optimal number of objects. $RUSH_T$ is thus best suited for very large systems in which disks will be added in large numbers. However, for systems which the designers intend to build by adding a very small number of disks at a time (small scale systems)

	RUSH					
	P	R	T	LH*	SH	Tab
Replication	X	X	X	X		-
Adjustable replication	X	X	X			-
Weighting	X	X	X			-
Arbitrary sub-cluster size	-	X			X	-
Optimal reorganization	-	-				-
Near-optimal reorganization	-	X	X	X		-
Configurable disk / sub-cluster removal		X	X	X		-
Erasure coding	X		X	X		-
Decentralized	X	X	X	X	X	
Minimize correlated failure	X	X	X			-
Allows sub-cluster resizing		X	X			-
Small memory footprint	X	X	X		X	

Table 2. Features of *RUSH* and three other related algorithms/distribution mechanisms. *SH* refers to simple hashing, and *Tab* is a simple tabular approach.

or systems that will be reorganized continuously, one of the other *RUSH* variants may be better.

$RUSH_R$ also provides significant flexibility, but at the cost of performance. Since $RUSH_R$ performance degrades linearly with the number of sub-clusters in the system, it is not well suited to a system where sub-clusters are added or removed frequently. It also cannot support systems where each "replica" in a replica group is actually a distinct entity rather than an exact copy, as would be the case if "replicas" were used as part of an erasure coding scheme. On the other hand, because $RUSH_R$ does not rely on the number theory behind $RUSH_P$ and $RUSH_T$, it can allow any replication factor up to the total number of disks in the system. This means that a system using $RUSH_R$ has more flexibility in the number of disks that can be removed from the system individually using the mechanism discussed in Section 3.4. It also has more flexibility to use a "fast mirroring copy" technique [21] to increase the mean time to data loss in a system.

$RUSH_P$ offers similar behavior to $RUSH_R$, except that it supports erasure coding. This ability comes at the expense of not being able to remove sub-clusters once they are added (without significant reorganization costs). $RUSH_P$ also requires that the replication factor in the system be less than or equal to the number of disks in the first sub-cluster added to the system, but subsequent sub-clusters need not have sufficient disks to hold all of the replicas of a single object on unique disks. In that respect, it is more flexible than $RUSH_T$, but less flexible than $RUSH_R$.

Because $RUSH_T$'s performance does not depend on the weighting of the sub-clusters, and objects on every sub-cluster can be located in the same amount of time, $RUSH_T$ can make better quality of service guarantees.

Table 2 gives a side-by-side comparison of the features of the *RUSH* variants and Linear Hashing variants [14], simple hashing, and a tabular, or "directory" approach. An "X" indicates that the algorithm supports the feature and "-" indicates that it supports the feature under some circumstances. The "tabular" approach keeps a centralized table, possibly cached in part on the clients, containing pointers to all of the object replicas. It has "-" instead of "X" because the approach neither rules out nor ensures any of these features. In fact, *RUSH* could be used to generate and maintain the table in such a system. Any such system has the same constraints to *RUSH*: it must guarantee that no two replicas of the same object are on the same disk and support weighting.

We conclude this section by giving a brief example of a system best suited to each of the *RUSH* variants. In a small system such as a small storage server or a replicated object database, the user does not necessarily want to add several disks or servers at once. In that case $RUSH_P$ or $RUSH_R$ is more appropriate. In the case of a small storage server, storage space is typically a more important issue that in an object database, where transaction speed is often more important. For that reason, $RUSH_P$, which supports erasure coding, is probably more suited to a small storage server, whereas $RUSH_R$ is probably more suited to an object database because of the added flexibility in configuration. Large storage systems, on the other hand, typically add many disks at once, and therefore are not constrained by the need to add a minimum numbers of disks to the system at once. Very large storage clusters are also typically very performance sensitive. Because $RUSH_T$ provides the best performance both in terms of lookup times and reorganizations, and because of its flexibility in configuration and reconfiguration, $RUSH_T$ is most appropriate for very large storage systems, such as the one we are designing in the Storage Systems Research Center at the University of California, Santa Cruz.

5. Related Work

The topic of Scalable Distributed Data Structures (SDDS) has received considerable attention. Litwin, *et al.* have developed many distributed variants of Linear Hashing which incorporate features such as erasure coding and security. The original LH* paper provides an excellent introduction to the LH* variants[14]. LH* unfortunately does not use disk space optimally [2], and results in a "hot spot" of disk and network activity during reorganizations. More importantly, LH* does not support weighting, and distributes data in such a way that increases the likelihood of correlated failures or performance degradations. Other data structures such as DDH [7] suffer from similar issues in utilizing space efficiently. Kröll and Widmayer [12] pro-

pose tree-based SDDSs called Distributed Random Trees which allow for complex queries such as range queries. DRTs do not support data replication (although the authors discuss metadata replication), and their worst case performance is linear in the number of *disks* in the system. Litwin *et al.* also propose a B-Tree based family called RP* [13], which suffers from problems similar to LH*.

Choy, *et al.* [5] describe algorithms for distributing data over disks which move an optimal number of objects as disks are added. These algorithms do not support weighting, replication, or disk removal. Brinkmann, *et al.* [3] propose a method to distribute data by partitioning the unit range. Their algorithm features 2-competitive reorganizations and supports weighting but not replication; it was extended by Wu and Burns [20] to map file sets, rather than files, to the unit range using a method similar to our technique of mapping objects to replica groups. SCADDAR [8] explored algorithms for assigning media blocks to disks in a system in which disks are added and removed. SCADDAR uses remapping functions similar in flavor to those in *RUSH*, but does not support replication beyond simple offset-based replication as discussed in Section 3.3. Consistent hashing [11] has many of the qualities of *RUSH*, but has a high migration overhead and is less well-suited to read-write file systems; Tang and Yang [18] use consistent hashing as part of a scheme to distribute data in large-scale storage clusters.

Chau and Fu discuss algorithms to support graceful degradation of performance during failures in declustered RAID systems [4]. As discussed in Section 3.3, our algorithms also feature graceful degradation of performance during failures.

Peer-to-peer systems such as CFS [6], PAST [17], and Gnutella [16] assume that storage nodes are extremely unreliable. Consequently, data has a very high degree of replication. Furthermore, most of these systems make no attempt to guarantee long term persistence of stored objects. In some cases, objects may be "garbage collected" at any time by users who no longer want to store particular objects on their node, and in others, objects which are seldom used are automatically discarded. Because of the unreliability of individual nodes, these systems use replication for high availability, and are less concerned with maintaining balanced performance across the entire system. Other large scale persistent storage systems such as Farsite [1] and OceanStore [15] provide more file system-like semantics. Objects placed in the file system are guaranteed, within some probability of failure, to remain in the file system until they are explicitly removed. The inefficiencies introduced by the peer-to-peer and wide area storage systems address security, reliability in the face of highly unstable nodes, and client mobility, among other things. However, these features require too much overhead for a tightly coupled high-performance object storage system.

6. Future Work

One problem with *RUSH* is that we do not yet know how to calculate the inverse—we can not directly answer the question, "which objects are stored on a given a disk?" We must instead iterate through all possible object identifiers and replica numbers and calculate the disk on which it belongs. We then simply discard all objects that do belong on the disk in question. Some preliminary research, however, suggests that it may be possible to invert *RUSH*, and enumerate the objects assigned to a particular server. This process involves solving a system of n linear equations, where n represents the number of comparisons necessary to locate the correct subcluster for an object.

We also would like to place theoretical bounds on the the number of objects which can move during a reorganization, and quantify the standard error in the number of objects stored on a particular disk.

We are currently exploring different mechanisms for reducing the impact of removing a single disk either temporarily or permanently from the system.

Finally, we are examining the utility of these algorithms for a broader class of applications including an object database and a searchable distributed web cache.

7. Conclusions

This paper has provided an overview of a family of algorithms we have developed to distribute objects over disks in a heterogeneous object based storage device. We describe three algorithms: $RUSH_P$, $RUSH_R$ and $RUSH_T$. These three algorithms all support weighting of disks, object replication, and near-optimal reorganization in may common scenarios.

Our experiments show that while all three algorithms can perform lookups very quickly, $RUSH_T$ performs an order of magnitude faster in systems which have been reorganized several times. $RUSH_T$ also provides the best reorganization behavior under many conditions. This increased flexibility comes at some expense to the range of configurations which are possible for $RUSH_T$. In particular, every subcluster in a system managed by $RUSH_T$ must have at least as many disks as an object has replicas. Since small systems will typically have small replication factors, this may or may not be an impediment. Clearly, however, $RUSH_T$ is the best algorithms for distributing data over very large clusters of disks.

$RUSH_P$ and $RUSH_R$ both provide alternatives to $RUSH_T$ for smaller systems. Since it has the greatest flexibility in

configurations $RUSH_R$ may be the best option for systems which need to remove disks one at a time from the system. Because it supports erasure coding, $RUSH_P$ may be the best option for smaller systems where storage space is at a premium.

RUSH algorithms operate well across a wide range of scalable storage systems. By providing support for replication, system growth and disk obsolescence, and totally decentralized lookup, *RUSH* enables the construction of high-performance, highly parallel object-based storage systems.

Acknowledgments

The research in this paper was supported in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714. We also thank the industrial sponsors of the Storage Systems Research Center, including Hewlett Packard, IBM, Intel, LSI Logic, Microsoft, ONStor, Overland Storage, and Veritas.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] Y. Breitbart, R. Vingralek, and G. Weikum. Load control in scalable distributed file structures. *Distributed and Parallel Databases*, 4(4):319–354, 1996.

[3] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform capacities. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 53–62, Winnipeg, Manitoba, Canada, Aug. 2002.

[4] S.-C. Chau and A. W.-C. Fu. A gracefully degradable declustered RAID architecture. *Cluster Computing Journal*, 5(1):97–105, 2002.

[5] D. M. Choy, R. Fagin, and L. Stockmeyer. Efficiently extendible mappings for balanced data distribution. *Algorithmica*, 16:215–232, 1996.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Banff, Canada, Oct. 2001. ACM.

[7] R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, pages 101–114, 1993.

[8] A. Goel, C. Shahabi, D. S.-Y. Yao, and R. Zimmerman. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*, pages 473–482, Feb. 2002.

[9] A. Granville. On elementary proofs of the Prime Number Theorem for Arithmetic Progressions, without characters. In *Proceedings of the 1993 Amalfi Conference on Analytic Number Theory*, pages 157–194, Salerno, Italy, 1993.

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

[11] 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 *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[12] B. Kröll and P. Widmayer. Distributing a search tree among a growing number of processors. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 265–276. ACM Press, 1994.

[13] W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A family of order-preserving scalable distributed data structures. In *Proceedings of the 20th Conference on Very Large Databases (VLDB)*, pages 342–353, Santiago, Chile, 1994.

[14] 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.

[15] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2003 Conference on File and Storage Technologies (FAST)*, pages 1–14, Mar. 2003.

[16] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the Gnutella network. *IEEE Internet Computing*, 6(1):50–57, Aug. 2002.

[17] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Banff, Canada, Oct. 2001. ACM.

[18] H. Tang and T. Yang. An efficient data location protocol for self-organizing storage clusters. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03)*, Phoenix, AZ, Nov. 2003.

[19] R. O. Weber. Information technology—SCSI object-based storage device commands (OSD). Technical Council Proposal Document T10/1355-D, Technical Committee T10, Aug. 2002.

[20] C. Wu and R. Burns. Handling heterogeneity in shared-disk file systems. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03)*, Phoenix, AZ, Nov. 2003.

[21] Q. Xin, E. L. Miller, D. D. E. Long, S. A. Brandt, T. Schwarz, 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*, pages 146–156, Apr. 2003.