MapX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems

Li Wang, *Didi Chuxing;* Yiming Zhang, *NiceX Lab, NUDT;*
Jiawei Xu and Guangtao Xue, *SJTU*

https://www.usenix.org/conference/fast20/presentation/wang-li

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA
978-1-939133-12-0

Open access to the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
is sponsored by

MAPX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems

Li Wang
laurence.liwang@gmail.com
Didi Chuxing

Yiming Zhang
sdiris@gmail.com (Corresponding)
NiceX Lab, NUDT

Jiawei Xu
titan_xjw@cs.sjtu.edu.cn
SJTU

Guangtao Xue
xue-gt@cs.sjtu.edu.cn
SJTU

Abstract

Data placement is critical for the scalability of decentralized object-based storage systems. The state-of-the-art CRUSH placement method is a decentralized algorithm that deterministically places object replicas onto storage devices without relying on a central directory. While enjoying the benefits of decentralization such as high scalability, robustness, and performance, CRUSH-based storage systems suffer from *uncontrolled* data migration when expanding the clusters, which will cause significant performance degradation when the expansion is nontrivial.

This paper presents MAPX, a novel extension to CRUSH that uses an extra time-dimension mapping (from object creation times to cluster expansion times) for controlled data migration in cluster expansions. Each expansion is viewed as a new layer of the CRUSH map represented by a virtual node beneath the CRUSH root. MAPX controls the mapping from objects onto layers by manipulating the timestamps of the intermediate placement groups (PGs). MAPX is applicable to a large variety of object-based storage scenarios where object timestamps can be maintained as higher-level metadata. For example, we apply MAPX to Ceph-RBD by extending the RBD metadata structure to maintain and retrieve approximate object creation times at the granularity of expansions layers. Experimental results show that the MAPX-based migration-free system outperforms the CRUSH-based system (which is busy in migrating objects after expansions) by up to 4.25× in the tail latency.

1 Introduction

Object-based storage systems have been widely used for various scenarios such as distributed file storage, remote block storage, small object (e.g., profile pictures) storage, blob (e.g., large videos) storage, etc. Compared to filesystem-based storage, object-based storage simplifies data layout by exposing an interface for reading and writing objects via unique object names, and thus reduces management complexity at the backend.

Objects are distributed among a large number of object storage devices (OSDs) possibly with various capacities and characteristics, making data placement critical for the scalability of object-based systems. Decentralized placement methods uniformly distribute objects among OSDs without relying on a central directory, and usually outperform centralized methods because their clients could directly access objects by calculating (instead of retrieving) the responsible OSDs. CRUSH [67] is the state-of-the-art placement algorithm that allows structured mapping from objects onto a hierarchical cluster map comprising nodes representing OSDs, machines, racks, etc. Currently, CRUSH has been widely adopted in large-scale storage systems (like Ceph [66] and Ursa [44]) owing to its simplicity and generality.

While enjoying the benefits of decentralization such as high scalability, robustness, and performance, CRUSH-based storage systems suffer from uncontrolled data migration after expanding the clusters and/or adding more intermediate placement groups (PGs). Although the migration could re-balance the load of the entire system right after the expansion, it also causes significant performance degradation when the expansion is nontrivial (e.g., adding several racks of storage machines).

In practical deployment of distributed storage systems, it is preferred to avoid large-scale data migration after cluster expansions [15], even at the cost of temporary load imbalance. Ceph [66] is a CRUSH-based object storage system which mitigates CRUSH's migration problem via implementation-level optimizations. It limits the migration rate to a relatively-low level, performing writes to the old OSDs if the written object is waiting for migration. However, all object replicas will be *eventually* migrated to the target OSDs calculated by the CRUSH algorithm, making Ceph experience degraded performance for a long period of time.

In contrast, traditional centralized placement methods could easily control data migration for cluster expansions. For example, Haystack [15] and HDFS [9] maintain a central directory recording object positions, so as to keep existing objects unaffected during expansions and place only new

objects onto the newly-added OSDs.

In this paper we present MAPX, a novel extension to CRUSH that uses an extra dimensional mapping (from object creation times to cluster expansion times) for controllable data migration in the expansion of decentralized object-based storage systems. Each expansion is viewed as a new layer of the CRUSH map represented by a virtual node beneath the CRUSH root. MAPX controls the mapping from objects onto layers by manipulating the timestamps of the intermediate PGs.

The time-dimension mapping cannot support *general* object storage where the maintenance overhead of per-object timestamps might be overwhelming. However, MAPX is applicable to a large variety of object-based storage scenarios (such as block storage and file storage), where the object creation timestamps can be maintained as higher-level storage metadata. We apply MAPX to Ceph-RBD (Reliable-autonomic-distributed-object-store Block Device) [3] and CephFS (Ceph File System) [4] with minimum modifications to the original CRUSH algorithm in Ceph (Luminous) [5]. For Ceph-RBD, we extend the *rbd_header* metadata structure to maintain and retrieve approximate object creation times at the granularity of expansion layers; while for CephFS, we extend the *inode* metadata structure to take the files' creation times, which could also be maintained at the granularity of layers, as the creation times of the files' objects. More complex applications of MAPX could be built based on block storage (Ceph-RBD) or file storage (CephFS). Experimental results show that the MAPX-based migration-free system outperforms the CRUSH-based system (which is busy in migrating objects after expansions) by up to 4.25× in the tail latency.

The rest of this paper is organized as follows. Section 2 introduces the background and problem of CRUSH. Section 3 presents the design of MAPX. Section 4 evaluates the performance of MAPX and compares it with CRUSH. Section 5 introduces related work. And finally Section 6 concludes the paper and discusses future work.

2 Background

2.1 CRUSH Overview

CRUSH uses a logical cluster map to abstract the storage cluster's hierarchical structure. Fig. 1 illustrates a three-level storage hierarchy, where the entire cluster (root) is composed of cabinets (representing racks), which are filled with shelves (representing storage machines) each installing many OSDs (disks). The internal nodes (root, cabinet, and shelf) in the hierarchy are referred to as buckets (the types of which are *straw2* throughout this paper as discussed in detail in Section 5.1). The hierarchy is flexible for extension. For example, cabinets might be further grouped into "row" buckets for larger clusters.

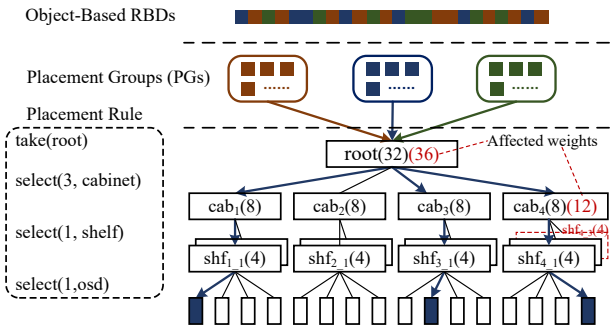

Figure 1: Example of CRUSH placement algorithm. An RBD is mapped to a PG which is subsequently mapped to a list of OSDs. The second operation (`select(3,cabinet)`) realizes three-way replication with three different cabinets. For simplicity each leaf OSD has the same weight of one.

Each OSD has a *weight* assigned by the administrator to control the OSD's relative amount of stored data, so that the load of an OSD is on average proportional to its weight. The weight of an internal bucket is (recursively) calculated as the sum of the weights of its child items. There are mainly two steps for CRUSH to place object replicas onto OSDs, which are briefly introduced below and will be discussed in more details in Section 5.1.

First, the objects are categorized into PGs by computing the modulo of the hashing of object names, i.e., *pgid* = `HASH(`*name*`) mod PG_NUM`. Second, the objects in a PG are mapped to a list of OSDs following the CRUSH algorithm. The first step is similar to traditional hashing and in the rest of this section we will briefly introduce the second step.

The CRUSH algorithm supports flexible constraints for reliable replica placement by (i) encoding the information of failure domains (like shared power source or network) into the cluster map, and (ii) letting the administrator define the *placement rules* that specify how replicas are placed by recursively selecting bucket items.

Fig. 1 demonstrates a typical placement procedure of CRUSH (for the dark blue PG) beginning at the root, where the values in the buckets' parentheses represent the weights. The first operation (`take(root)`) of the rule selects the root of the storage hierarchy and uses it as an input to subsequent operations. The second operation (`select(3,cabinet)`) repeatedly computes the following Eq. (1) to choose $x = 3$ items (cabinets at this level) for three-way replication, from totally $|\vec{i}| = 4$ items $\in \vec{i}$ beneath the root:

$$C(pgid, \vec{i}, r) = \underset{i \in \vec{i}}{\operatorname{argmax}} HASH(pgid, r, ID(i)) \times W(i), \quad (1)$$

where *pgid* is the ID of the input PG, $r = 1, 2, \cdots$ is a parameter for the argmax computation, *HASH* is a three-input hash function, and $ID(i)$ and $W(i)$ are the ID and weight of an item $i \in \vec{i}$, respectively. To choose x distinct items, it is

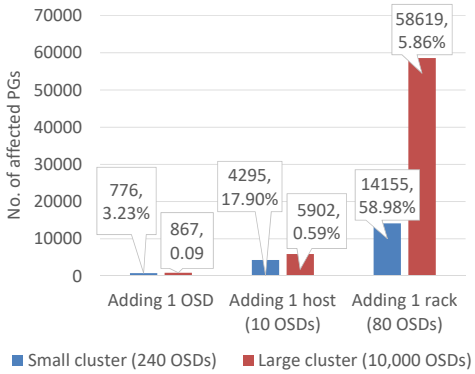

Figure 2: Data migration of two simulated CRUSH clusters during expansions.

possible to perform Eq. (1) more than x times because the output of Eq. (1) may have already been chosen in previous computation or the chosen item may be failed/overloaded.

Similarly, the subsequent operations (`select(1,shelf)` and `select(1,osd)`) follow Eq. (1) to choose $x = 1$ shelf and OSD beneath each of the three cabinets. The final result of the placement rule is the three darkblue OSDs in Fig. 1.

2.2 The Main Drawback of CRUSH

CRUSH achieves statistical load balancing without a central directory, and could automatically re-balance the load when the storage cluster map changes. On the downside, however, it also causes *uncontrollable* data migration in cluster expansions. For instance, adding a new shelf (shf_{4_3}) with 4 OSDs beneath a cabinet (cab_4) in Fig. 1 will affect the weights (labeled in the second red parentheses) of all items along the path from the newly-added shelf up to the root, and thus will lead to data movement not only from other shelves in cab_4 to the newly-added shf_{4_3} but also from other cabinets to cab_4. The amount of data migration can be as high as $h\frac{\Delta w}{W}$ if Δw is small relative to W [67], where h is the number of levels in the hierarchy, and Δw and W are the increased weight of the expansion and the total weight of all OSDs, respectively.

To demonstrate the severity of the problem, we measure the amount of data movement in two simulated CRUSH-based three-level Ceph clusters, which adopt three-way replication taking a rack as a failure domain. One rack consists of 8 hosts each containing 10 OSDs. The first small cluster has a total of 3 racks, 24 hosts, and 240 OSDs, and stores 24,000 PGs; while the second large cluster has 125 racks, 1000 hosts, and 10,000 OSDs, and stores 1,000,000 PGs. We respectively add one OSD, one machine, and one rack to the two clusters. The result (Fig. 2) shows that the migration is significant when the expansion is nontrivial, e.g., almost 60% of the PGs will be affected when adding one rack to the small cluster, which will inevitably cause performance degradation during the entire migration period.

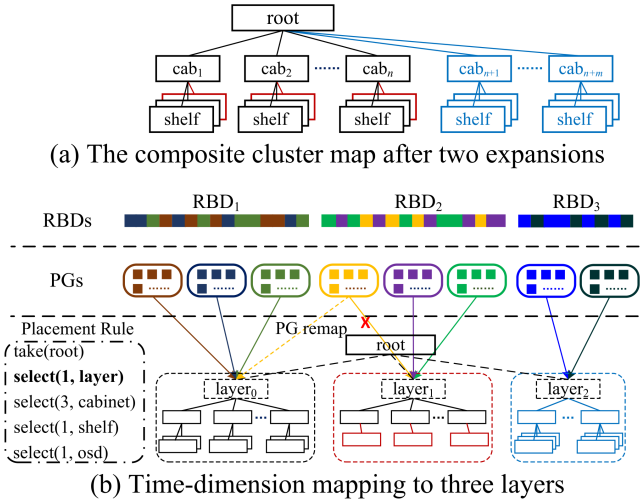

Figure 3: MAPX records each expansion as a layer. MAPX implicitly adds a *select* operation (`select(1,layer)`) to the placement rule.

3 MAPX Design

Compared to moderate load imbalance, large-scale data migration often has much more negative impact on I/O performance in the expansion of distributed storage systems. The CRUSH placement algorithm suffers from data migration after each cluster expansion because it "crushes" the differences between the new and the old objects/OSDs. To address this problem, MAPX extends the original CRUSH algorithm with an extra time-dimension mapping.

3.1 Migration-Free Expansion

Storage systems usually prefer to avoid data migration after cluster expansion even at the cost of temporary load imbalance. For instance, Haystack and HDFS leverage a central directory to keep existing objects unaffected during cluster expansions. As new objects are stored onto the new OSDs, the available capacity of them decreases over time and thus eventually the entire system will achieve approximate load balancing. Data migration can be performed (with metadata modification) at any time as needed.

Inspired by the centralized placement methods, our goal is to achieve controlled data migration for cluster expansions. To achieve this, we design MAPX on top of CRUSH by introducing an extra time-dimension mapping to distinguish the new and the old objects/OSDs, while still preserving the benefits of randomness and uniformness of CRUSH.

Fig. 3(a) depicts an example of two expansions to the original cluster which consists of n cabinets each having two shelves. The first expansion adds a shelf (represented by a red rectangle) to each of the n cabinets and the second expansion adds m cabinets (represented by blue rectangles).

Algorithm 1 Extended `select` Procedure of MAPX

```
1:  procedure SELECT(number, type)
2:      if type ≠"layer" then
3:          return CRUSH_SELECT(number, type)
4:      end if
5:      layers ← layers beneath currently-processing bucket
        ▷ each layer represents an expansion
6:      num_layers ← number of layers in layers
7:      pg ← current Placement Group
8:      o⃗ ← Φ                                          ▷ output list
9:      for (i = num_layers − 1; i ≥ 0; i − −) do
10:         layer ← layers[i]
11:         if layer.timestamp ≤ pg.timestamp then
12:             if layer was chosen by previous select then
13:                 continue
14:             end if
15:             o⃗ ← o⃗ + {layer}
16:             number ← number − 1
17:             if number == 0 then
18:                 break
19:             end if
20:         end if
21:     end for
22:     return o⃗
23: end procedure
```

Unlike CRUSH which *monolithically* updates the cluster map, MAPX views each expansion, as well as the original cluster, as a separate *layer* which contains not only the new leaf OSDs but also all the internal buckets (shelves, cabinets, etc.) from the leaf OSDs up to the root.

To support the time-dimension mapping with minimum modifications to CRUSH, we insert a virtual level beneath the common CRUSH root (Fig. 3(b)), where each virtual node represents a layer of expansion. The virtual level enables MAPX to realize migration-free expansion by mapping new objects to the new layer before further processing of the CRUSH algorithm. Since the new layer will not affect the weights of the old ones, the placement of old objects within old layers will not change.

Mapping objects to PGs. In each expansion, the new layer is assigned with a certain number of newly-created PGs each having a timestamp (t_{pgs}) equal to the layer's expansion time (t_l). When writing/reading an object O (with creation timestamp t_o), we first compute the ID ($pgid$) of O's PG by

$$pgid = Hash(name) \bmod \text{INIT\_PG\_NUM}[j] + \sum_{i=0}^{j-1} \text{INIT\_PG\_NUM[i]}, \qquad (2)$$

where $name$ is the object name, INIT_PG_NUM[i] is the initial number of PGs of the i^{th} layer, and the j^{th} layer has the latest timestamp $t_l \le t_o$ among all layers. Note that although PGs might be remapped to other layers for, e.g., load rebalancing (Section 3.2), `INIT_PG_NUM` is a layer's constant and thus the mapping from objects to PGs is *immutable*. Consequently, each object is mapped to a responsible PG during creation, which has the latest timestamp $t_{pgs} \le t_o$ among all PGs. For instance, suppose that the three RBD_1, RBD_2, and RBD_3 in Fig. 3(b) are created respectively after the expansions of layer_0, layer_1, and layer_2. The objects of RBD_1, RBD_2, and RBD_3 will use the three layers' `INIT_PG_NUM` to calculate their PGs respectively within layer_0, layer_1, and layer_2.

Mapping PGs to OSDs. Similar to CRUSH, MAPX maps a PG onto a list of OSDs following a sequence of operations in a user-defined placement rule. As shown in Fig. 3(b), MAPX implicitly adds a *select* operation (`select(1,layer)`) to the placement rule, so as to realize the time-dimension mapping from PGs to layers without disturbing the administrators. Internally, MAPX extends CRUSH's original *select* operation to support the *layer*-type `select()`, as shown in Algorithm 1. If *type* is not "layer", then the processing is the same as the original CRUSH (Lines $2 \sim 4$). Otherwise, we initialize an array of *layers* which stores all layers beneath the currently-processing bucket (usually the root) in an ascending order of the layers' timestamps (Line 5). We also initialize *num_layers* (the number of layers), *pg* (the placement group), and $\vec{o}$ (the output list) at Lines $6 \sim 8$. Then the loop (Lines $9 \sim 21$) adds *number* layers in the array of *layers* to the output list $\vec{o}$. In most cases $number = 1$ so that the PG could be mapped to OSDs in one layer, but it is also possible to specify a larger *number* for, e.g., mirroring between two layers of expansions.

Note that the replicas of an object are not necessarily all placed on the newest layer. For example, suppose that the last expansion (layer_2) adds only two cabinets in Fig. 3(a) (i.e., $m = 2$) but the second `select()` function (`Select(3, cabinet)`) requires three cabinets. This will cause the first `select()` function (`select(1, layer)`) to be invoked twice to satisfy the rules following the backtracking mechanism of CRUSH: when a `select()` function cannot select enough items beneath a "layer" bucket, MAPX will retain (rather than abandon) the selected items and backtrack to the root to select the lacking items beneath a previous layer. Lines $12 \sim 14$ check whether *layer* has been chosen by previous `select()` and if so we continue to the next loop, so as to avoid duplicate layer selection when performing backtracking. The double check ensures Algorithm 1 to correctly handle this situation, respectively returning layer_2 and layer_1 for the first and second `select()` functions.

3.2 Migration Control

The MAPX-based migration-free placement algorithm provides (statistical) load balancing within each layer, owing to the randomness and uniformness of the original CRUSH

algorithm, and achieves approximate load balancing among different layers by timely expanding the cluster when the load of the current layer increases to the same level as previous layers.

However, the load of a layer might change because of, e.g., removals of objects, failures of OSDs, or unpredictable workload changes. In Fig. 3, for example, it is possible that the cluster performs the second expansion ($layer_2$) when the load of the first expansion ($layer_1$) is as high as that of the original cluster ($layer_0$), but afterwards a large number of objects of $layer_1$ are removed and consequently the loads of the first two layers may get imbalanced.

To address the potential load imbalance problem, we design three flexible strategies for dynamically managing the load in MAPX, namely, placement group remapping, cluster shrinking, and layer merging.

PG remapping. MAPX supports to control object data migration by dynamically remapping the PGs. Each PG has two timestamps, namely, a static timestamp (t_{pgs}) that is equal to the expansion time of the PG's initial layer, and a dynamic timestamp (t_{pgd}) that could be set to any layer's expansion time. Different from the mapping from objects to PGs which uses static timestamps (Section 3.1), the mapping from PGs to layers is performed by comparing the PGs' dynamic timestamps to the layers' timestamps (Line 11 in Algorithm 1). Consequently, a PG can be easily remapped to any layer by manipulating the dynamic timestamp (as illustrated in Fig. 3(b)), which will be notified to all OSDs and clients via incremental map updates. The storage overhead for PGs' timestamps is moderate. For example, if we use a one-byte index for each PG timestamp (pointing to the corresponding layer's timestamp) which supports a maximum of $2^8 = 256$ layers), and suppose that one machine has 20 OSDs each responsible for 200 PGs, then the memory overhead of timestamps for a 1000-machine cluster is $1000 \times 20 \times 200 \times 2 \times 1\text{B} = 8\text{MB}$.

Cluster shrinking. When the load of a layer becomes lower than a threshold, MAPX shrinks the cluster by removing the layer's devices (such as OSDs, machines, and racks) from the cluster, as an inverse operation of cluster expansions. Given a layer Ω to be removed from the cluster, we first assign all PGs in Ω to the remaining layers according to their aggregated weights (for simplicity the reassignment does not consider the actual loads of the layers), and then migrate the PGs to the target layers through remapping (as discussed above). After shrinking the layer Ω is logically preserved (with no physical devices or PGs) and its `INIT_PG_NUM` will not change, so as not to affect the mapping from objects to PGs (following Eq. (2)).

Layer merging. MAPX balances the loads of two layers (Ω and Ω') via layer merging, which could be easily realized by setting the expansion time of one layer (Ω') to be the same as that of the other (Ω).

3.3 Implementing MAPX in Ceph

We have implemented the MAPX structure in Ceph by augmenting the original CRUSH algorithm with an extra time-dimension mapping. As shown in Fig. 3(b), the internal buckets (like shelves, cabinets, and rows, but not leaf OSDs) may belong to multiple layers. Therefore, we assign an internal device in a particular layer (i.e., beneath a particular virtual node) with a virtual device ID by concatenating the physical device ID and the layer's timestamp. We use the weight fields of the virtual nodes to record the layers' timestamps, which will be compared with the PGs' dynamic timestamps for layer selection.

MAPX is not suitable for general object stores, mainly because it is nontrivial to maintain and retrieve the timestamps of arbitrary objects. The overhead of per-object timestamp maintenance is similar to that of the maintenance of a central directory, and thus should be avoided in decentralized placement methods like CRUSH and MAPX. However, MAPX is applicable to a large variety of object-based storage systems such as block storage (Ceph-RBD [3]) and file storage (Ceph-FS [4]), where the object timestamps can be maintained as higher-level metadata.

Ceph-RBD. We have implemented the metadata-based timestamp retrieval mechanism for Ceph-RBD (RADOS Block Device). Ceph stores the metadata (such as the prefix of data object names, and the information of volume, snapshot, striping, etc.) of an RBD in its *rbd_header* structure, which will be retrieved when a client mounts the RBD via *rbd_open*. Since an object of an RBD can be created after any expansions, we inherit the timestamp of the current layer (when an object is created) as the object's timestamp. Therefore, we add a per-object index (named *object_timestamp*) to the *rbd_header* structure which points to each layer's expansion time. The storage overhead for the extra metadata is moderate. For example, if we use one byte for the per-object index and each object is 4MB, then the storage overhead of the *object_timestamp* array for a 4TB RBD is at most $\frac{4\text{TB}}{4\text{MB}} \times 1\text{B} = 1\text{MB}$.

CephFS. We have also (partially) implemented the timestamp retrieval mechanism for CephFS (Ceph Filesystem). Ceph stores the file metadata (including file creation times) in the *inode* structure. A client reads *inode* when opening a file and gets the file creation time. Currently we let all the objects of a file inherit the file's timestamp, so that we could control the time-dimension mapping at the granularity of files. We also plan to support finer-grained object timestamp maintenance. If the size of a file exceeds a threshold T (e.g., $T = 100$ MB), we could divide it into subfiles each smaller than 100 MB. The file's metadata maintains both the mapping from the file to its sub-files and the creation timestamp of each subfile, so that we could control the time-dimension mapping at the granularity of subfiles.

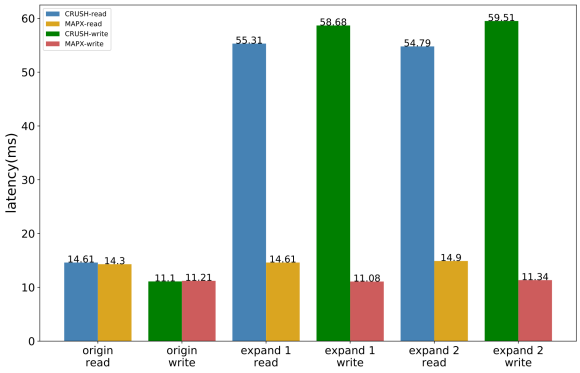

Figure 4: 99th percentile I/O latency of MAPX and CRUSH (during cluster expansions).

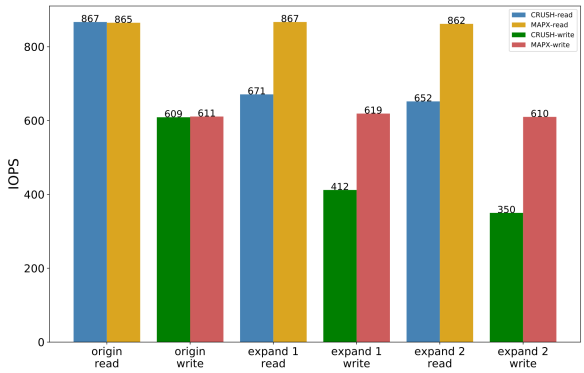

Figure 5: IOPS of MAPX and CRUSH (during cluster expansions).

4 Evaluation

In this section we evaluate the performance of the MAPX-based Ceph and compare it with that of the original CRUSH-based Ceph. Our testbed consists of four machines, of which three machines run the Ceph OSD storage servers and the other machine runs the client. Each machine has dual 20-core Xeon E5-2630 2.20GHz CPU, 128GB RAM, and one 10GbE NIC, running CentOS 7.0. Each storage machine, installs four 5.5TB HDDs, and runs Ceph 12.2 (Luminous) with the BlueStore backend. In all experiments every storage machine is viewed as a failure domain. The Ceph monitor is co-located with one of the storage servers. The client runs the `fio` benchmark.

4.1 I/O Performance during Expansions

We compare the I/O performance of MAPX and CRUSH during expansions, respectively being used as the object placement methods for Ceph.

We use the default values of all parameters of Ceph except $OSD\_max\_backfills$. As discussed in Section 1, Ceph mitigates the migration problem of CRUSH via implementation-level optimizations. It uses the parameter $OSD\_max\_backfills \geq 1$ to trade off between the severity and duration of performance degradation caused by data migration.

By default Ceph sets the parameter $OSD\_max\_backfills = 1$, which makes migration have the lowest priority so that objects in PGs could be migrated with an extremely-low speed. Although partially mitigating the degradation problem, setting $OSD\_max\_backfills = 1$ will significantly extend the migration period and largely increase the write load before the migration completes: writes to a PG waiting for migration will first be performed to the origin OSD and then be asynchronously migrated to the target OSD. Clearly, this makes Ceph experience *less severe* performance degradation but for a *longer* period of time. We set $OSD\_max\_backfills = 10$, which is more reasonable in this experiment so that migration could get a higher priority to demonstrate the algorithm-level difference between MAPX and CRUSH. We will discuss more on the impact of migration priority in Section 5.2.

The initial Ceph cluster has three storage machines each of which has two OSDs. We create 128 PGs, and the three-way replication results in (on average) $128 \times 3 \div 3 \div 2 = 64$ PGs for which each OSD will be responsible. We create 40 RBD images (each with 20GB data) in the initial cluster. We expand the storage cluster by respectively adding one and two OSDs to each machine in the cluster. We evaluate the performance (including I/O latency and IOPS) of Ceph running the migration-free MAPX, and compare it with the performance of Ceph running the original CRUSH algorithm. The I/O size is 4KB. The iodepth is 1 and 128 in the latency and IOPS tests, respectively.

Fig. 4 shows the evaluation result for the 99th percentile tail latencies. Note that cloud storage scenarios usually care about the (99th, 99.9th, or 99.99th percentile) tail latency rather than the mean or median latency, so as to guarantee SLA. MAPX outperforms CRUSH by up to 4.25×, mainly because the migration in CRUSH severely contends with the normal I/O requests. In this experiment, MAPX always uses six OSDs of the initial cluster to serve I/O requests because it does not migrate existing RBDs to the new OSDs. In contrast, CRUSH respectively uses six, nine, and twelve OSDs, but the CRUSH-induced data migration severely degrades the performance, which is unacceptable for latency-sensitive applications.

Fig 5 shows the evaluation result for IOPS respectively in MAPX and CRUSH. Each result is the mean of 20 runs, and we omit the error bars because the variances to the mean are relatively small (less than 5%). Similar to the latency test, MAPX significantly outperforms CRUSH by up to 74.3% in the IOPS test, because CRUSH's data migration contends with the normal I/O requests.

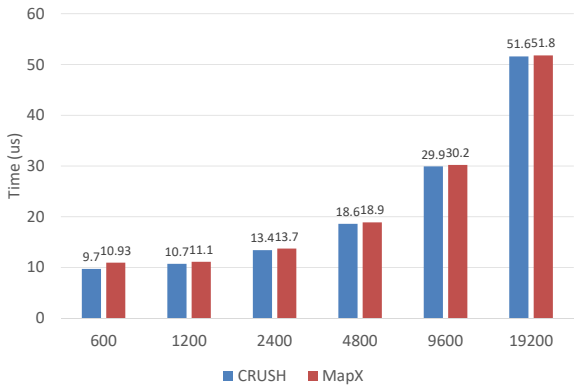

Figure 6: Computation overhead of MAPX and CRUSH.

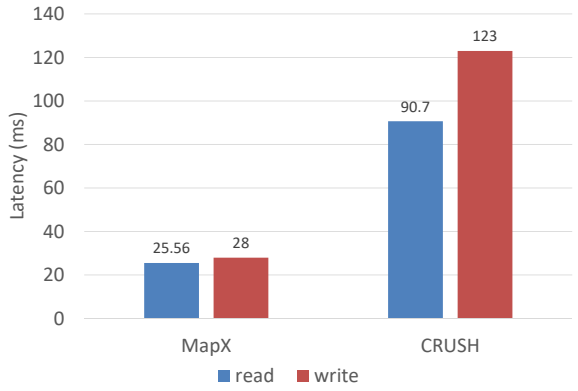

Figure 7: 99th percentile I/O latency of MAPX and CRUSH (during cluster shrinking).

4.2 Computational Overhead

We compare the computation times of MAPX and CRUSH by simulating a Ceph cluster of different numbers of OSDs (varying from 600 to 19,200). The result (Fig 6) shows that both MAPX and CRUSH can map an object to an OSD in tens of microseconds. The small extra times of MAPX compared to CRUSH come from the computation of the time-dimension mapping beneath the root.

4.3 I/O Performance during Shrinking

We evaluate the I/O performance of MAPX (used as the object placement methods for Ceph) in shrinking. The Ceph cluster has three storage machines each initially having three OSDs, and we expand the cluster by adding one OSD to each of the three machines using the same configurations as that in Section 4.1. We then remove the newly-added layer (i.e., removing one OSD from each of the three machines), following the shrinking method (introduced in Section 3.2). We control the migration speed by setting the number of concurrently migrated PGs to eight.

Fig. 7 depicts the 99th percentile I/O latency of MAPX during cluster shrinking. For reference, Fig. 7 also shows the 99th percentile latency of CRUSH in shrinking by removing one OSD from each of the three machines. Ceph shrinks the cluster by directly modifying the cluster map. Note that the result does not necessarily mean that MAPX has lower latency than CRUSH in shrinking, because they adopt different throttling mechanisms. However, MAPX outperforms CRUSH during cluster shrinking in that MAPX requires less migration than CRUSH. For instance, removing an OSD in CRUSH will lower the entire subtree's weight and thus may result in unnecessary data migration. In contrast, MAPX never causes migration between preserved OSDs because shrinking occurs at the granularity of layers. We omit the result for IOPS during shrinking due to lack of space, which has similar trends with that for I/O latency.

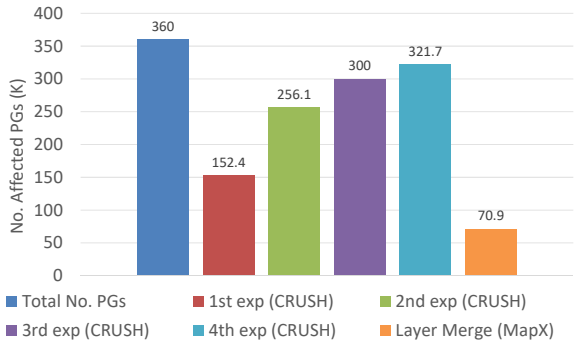

Figure 8: Number of affected PGs in layer merging in MAPX (after four expansions). Since CRUSH does not support merging, for reference we measure the number of affected PGs after each expansion in CRUSH.

4.4 Layer Merging

We use `CrushTool` [6] to emulate layer merging in MAPX. We adopt three-way replication where each object has three replicas stored on three OSDs. Initially the storage cluster consists of 5 racks each having 20 machines. One machine has 20 OSDs. There are totally 100 machines and 2000 OSDs, storing 200,000 PGs. We expand the cluster four times. In each expansion, we add a new layer of one rack (of 20 machines and 400 OSDs), and add 40,000 new PGs to the new layer. Clearly, MAPX maps all the new PGs onto the newly-added OSDs and thus no migration happens. After the four expansions, there are totally 9 racks, 180 machines, and 3600 OSDs, storing 360,000 PGs. We then merge the 40 machines of the first and second expansions (as introduced in Section 3.2), and measure how many PGs are affected by the merging in MAPX.

The result is depicted in Fig. 8, where layer merging in MAPX affects 70,910 PGs among all the 80,000 PGs of the two merged layers. The relatively high ratio of affected PGs in layer merging of MAPX is decided by the

nature of CRUSH. For reference, we also emulate the four expansions in CRUSH, where we let the cluster initially have 360,000 PGs and do not add new PGs during expansions, because otherwise CRUSH will change the mapping from objects to PGs causing many more PGs to be migrated. Fig. 8 also shows how many PGs are affected by each expansion in CRUSH. For instance, almost 90% of all the PGs are affected in the fourth expansion when the number of machines increases from 160 to 180.

5 Related Work

5.1 CRUSH in Ceph

Ceph [66] is a widely-used object-based storage system supporting block storage [3], file storage [4], and simple object storage [8] (like S3 [1]). To deterministically and uniformly maps data objects onto OSDs without relying on a central directory, Ceph applies CRUSH by taking the following two steps.

In the first step, Ceph computes the placement groups (PGs) of the objects. The actual computation of PGs is slightly more complicated than simple hashing and modulo (discussed in Section 2.1) when the PG number (PG_NUM) is not a power of two: it computes the *pgid*s with *double-modulo* by using two values of $2^{n^{\text{th}}}$ power near PG_NUM, so as to minimize *pgid* changes when changing the numbers of PGs. For instance, consider two objects A and B with $\texttt{HASH}(A) = 25$ and $\texttt{HASH}(B) = 29$. Suppose that at first the PG has $\texttt{PG\_NUM}_1 = 8$, which results in $pgid_A = 1$ and $pgid_B = 5$. Then, suppose that we increase the PG number to $\texttt{PG\_NUM}_2 = 12$. Since $2^3 < 12 < 2^4$, Ceph first computes the modulo for A and B using $2^4 = 16$, and respectively gets $pgid_A = 9$ and $pgid_B = 13$. For $pgid_A < \texttt{PG\_NUM}_2$, Ceph will take $pgid_A = 9$ as the final *pgid* of A. In contrast, for $pgid_B > \texttt{PG\_NUM}_2$, Ceph will compute the modulo again using $2^3 = 8$ and get $pgid_B = 5$ as the final *pgid* of B. Clearly, the double-modulo mechanism makes the *pgid*s not to change when the first modulo is between $\texttt{PG\_NUM}_2 = 12$ and $2^4 = 16$.

In the second step, Ceph maps *pgid*s onto OSDs in the storage cluster, where the hierarchy is composed of OSDs and buckets. Buckets can contain any number of OSDs or other buckets. OSDs are always at the leaves and are assigned weights by the administrator to control the relative amount of data they are responsible for. Bucket weights are the sum of the weights of its items. Currently CRUSH has five types (uniform, list, tree, straw, and straw2) of buckets, and different bucket types use different formulas to choose a given number of items beneath the bucket. The straw2 buckets are the most popular because they have the smallest migration overhead when changing the cluster map or the number of PGs. By default all buckets in Ceph have the *straw2* type.

5.2 Load Balancing & Migration Overhead

Ceph developers have realized the performance degradation problem due to expansion-caused migration. They alleviate this problem through implementation-level optimizations by lowering the priority of migration tasks to avoid bursty migration after the expansion [7]. However, the PGs calculated by CRUSH have to be *eventually* migrated. Further, the conservative migration settings significantly extend the migration period during which a large fraction of PGs are waiting for migration. This complicates their write procedure (first being written to the origin OSDs and then to the target OSDs), unnecessarily increasing the load.

In contrast, MAPX provides administrators with the ability to *control* the migration at the algorithm level: the migration may never happen if (as in most cases) there is not severe imbalance between the loads of different layers. Further, sometimes CRUSH needs to increase the number of PGs, for example to reduce the per-OSD load, which causes a large fraction of objects to be migrated even using the double-modulo method (Section 5.1), while MAPX could smoothly add PGs during expansions without migration.

Focusing on OSD failure caused data migration, Ref. [36] proposes to use *cluster device flags* to selectively label failed OSDs for reducing data transfer. However, it is not clear how to use the flags to address/alleviate the migration problem when expanding the storage clusters.

Consistent distributed hash tables (DHTs) [63, 57, 74, 59, 60, 38, 73] are widely used for decentralized *overlay* storage. Early DHTs require multi-hop routing to locate the data and thus are not suitable for distributed object storage. For example, Chord [63] uses hashing to map both the IDs of storage nodes and the keys of data onto a ring. A node is responsible for a key if it is the nearest node after the key on the ring. Each node only has routing information about a subset of nodes on the ring, and it takes $O(\log N)$ time to locate a key in an N-node Chord network. Later DHT networks (like OneHop [18]) support direct key locating by maintaining all routing information on each node in the system, and have been adopted in some decentralized object stores including Amazon Dynamo [28], S3 [1], and OpenStack Swift [11].

Compared to CRUSH, most DHTs cannot express the storage hierarchy including OSDs, machines, racks, etc. DHT-based storage systems have to use additional mechanisms to model the hierarchy (e.g., Cassandra [41] and CubeX [71] respectively adopt virtual nodes and multi-level cubic ring [70], and hierarchy-aware DHTs[33, 51, 29, 39, 69] adopt hierarchical routing tables), which are inflexible compared to CRUSH. Further, load assignment in DHTs is decided by the positions of the nodes and keys on the ring, and thus adding a new node will only make a portion of the load of its successor move to it, which inevitably causes imbalance (although introducing less migration).

5.3 Storage Systems

Decentralized Object storage systems. In recent years, decentralized object storage has been widely used in various scenarios. For example, Twitter uses virtual buckets to store its photos [2], LinkedIn designs Ambry [54] which adopts logical grouping and asynchronous replication to realize geo-distributed object storage [61], and Facebook designs F4 [52] which adopts erasure coding [45] to reduce replication factors for its *warm* objects. Key-value (KV) storage systems [10, 20, 28, 40, 47] could be viewed as generalized object stores that provide an interface for reading, writing, deleting and modifying the values associated with keys. Unlike general object stores, their values are often relatively small.

Centralized Object storage systems. Some object stores adopt a centralized metadata directory to simplify data placement. Haystack [15] is a centralized object store for Facebook's large amounts of small objects like photos, audio/video pieces, H5 files, etc. Haystack places object data (packed into needles) in large files stored in data servers, and stores object positions (i.e., on which machines) in a central directory. Similar to Haystack, Lustre [16] and HDFS [9] leverage a central directory to maintain object positions which helps keep existing objects unaffected during cluster expansions. The central directory based placement methods are inefficient in scalability and robustness. Further, the multi-phase I/O of metadata and data leads to poor performance and complicates consistency issues [23, 22, 55, 34] and thus cannot satisfy the requirement of the emerging OLDI (online data-intensive) applications [25, 68]. Compared to the centralized placement methods, MAPX preserves the benefits of decentralized CRUSH placement algorithm while providing flexible control over data migration in expanding the storage clusters.

Block storage systems. Large-scale block storage systems [65, 49, 42, 35] adopt distributed protocols [12, 17] to provide block interface to remote clients. For example, Ursa [44] designs a hybrid block store for optimizing SSD-based storage [46, 14, 27, 26, 13]. Salus [64] provide virtual disk service based on HBase [31]. Blizzard [50] realizes high-performance parallel I/O based on FDS [53]. PARIX [45, 72] performs speculative partial writes to alleviate the inability of erasure coding (EC) [19, 62, 37] and efficiently support random small writes.

File systems. Distributed file systems spread the data of a file across many storage servers [22, 24, 30, 32, 35, 43, 48, 58]. For instance, GFS [30] is a large-scale fault-tolerant file system for data-intensive cloud applications. Zebra [32] uses striping on RAID [21] and logs for high disk parallelism. BPFS [24] focuses on persistent memory hardware and uses epoch barrier to provide an in-memory file system with ordering guarantees. OptFS [22] improves the journaling file system [56] by decoupling durability from ordering.

6 Conclusion

The contention between decentralized and centralized data placement methods has been long lived in the design of large-scale object storage systems. The decentralized CRUSH method achieves high scalability, robustness, and performance, but suffers from uncontrollable data migration in cluster expansions. This paper presents MAPX, a novel extension to CRUSH that embraces the best of both decentralized and centralized methods. MAPX controls data migration by introducing an extra time-dimension mapping from object creation times to cluster expansion times, while still preserving the randomness and uniformness of CRUSH. We have applied MAPX to Ceph-RBD and CephFS, respectively by extending the *rbd_header* and *inode* metadata structures. In our future work, we will study how to reduce the maintenance overhead of object timestamps, so as to apply MAPX to a broader range of object-based storage scenarios.

Acknowledgement

We would like to thank John Bent, our shepherd, and the anonymous reviewers for their insightful comments. We thank Mingya Shi and Haonan Wang for helping in the experiments, and we thank the Didi Cloud Storage Team for their discussion. Li Wang and Yiming Zhang are co-primary authors. Jiawei Xu implemented some parts of MAPX when he was an intern at Didi Chuxing. This research is supported by the National Key R&D Program of China (2018YFB2101102), the National Natural Science Foundation of China (NSFC 61772541, 61872376 and 61370018). and the Joint Key Project of the NSFC (U1736207).

References

[1] https://aws.amazon.com/s3/.

[2] https://blog.twitter.com/engineering/en_us/a/2012/blobstore-twitter-s-in-house-photo-storage-system.html.

[3] https://ceph.com/ceph-storage/block-storage/.

[4] https://ceph.com/ceph-storage/file-system/.

[5] https://docs.ceph.com/docs/master/releases/luminous/.

[6] https://docs.ceph.com/docs/mimic/man/8/crushtool/.

[7] https://docs.ceph.com/docs/mimic/rados/configuration/osd-config-ref/.

[8] https://github.com/ceph/ceph/tree/master/src/rgw.

[9] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[10] https://rocksdb.org/.

[11] https://www.swiftstack.com/product/open-source/openstack-swift/.

[12] AIKEN, S., GRUNWALD, D., PLESZKUN, A. R., AND WILLEKE, J. A performance analysis of the iscsi protocol. In *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on* (2003), IEEE, pp. 123–134.

[13] ANAND, A., MUTHUKRISHNAN, C., KAPPES, S., AKELLA, A., AND NATH, S. Cheap and large cams for high performance data-intensive networked systems. In *NSDI* (2010), USENIX Association, pp. 433–448.

[14] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *SOSP* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 1–14.

[15] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in haystack: facebook's photo storage. In *Usenix Conference on Operating Systems Design and Implementation* (2010), pp. 47–60.

[16] BRAAM, P. The lustre storage architecture. *arXiv preprint arXiv:1903.01955* (2019).

[17] CASHIN, E. L. Kernel korner: Ata over ethernet: putting hard drives on the lan. *Linux Journal 2005*, 134 (2005), 10.

[18] CASTRO, M., COSTA, M., AND ROWSTRON, A. I. T. Debunking some myths about structured and unstructured overlays. In *NSDI* (2005).

[19] CHAN, J. C., DING, Q., LEE, P. P., AND CHAN, H. H. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (2014), pp. 163–176.

[20] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *Acm Transactions on Computer Systems 26*, 2 (2008), 1–26.

[21] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR) 26*, 2 (1994), 145–185.

[22] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 228–243.

[23] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012* (2012), p. 9.

[24] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.

[25] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[26] DEBNATH, B., SENGUPTA, S., AND LI, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 25–36.

[27] DEBNATH, B. K., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. *PVLDB 3*, 2 (2010), 1414–1425.

[28] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. *Acm Sigops Operating Systems Review 41*, 6 (2007), 205–220.

[29] GANESAN, P., GUMMADI, P. K., AND GARCIA-MOLINA, H. Canon in g major: Designing dhts with hierarchical structure. In *ICDCS* (2004), pp. 263–272.

[30] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[31] HARTER, T., BORTHAKUR, D., DONG, S., AIYER, A., TANG, L., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (2014), pp. 199–212.

[32] HARTMAN, J. H., AND OUSTERHOUT, J. K. The zebra striped network file system. *ACM Transactions on Computer Systems (TOCS) 13*, 3 (1995), 274–310.

[33] HARVEY, N. J. A., JONES, M. B., SAROIU, S., THEIMER, M., AND WOLMAN, A. Skipnet: A scalable overlay network with practical locality properties. In *USENIX Symposium on Internet Technologies and Systems* (2003).

[34] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[35] HILDEBRAND, D., AND HONEYMAN, P. Exporting storage systems in a scalable manner with pnfs. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)* (2005), IEEE, pp. 18–27.

[36] HUANG, M., LUO, L., LI, Y., AND LIANG, L. Research on data migration optimization of ceph. In *2017 14th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)* (2017), IEEE, pp. 83–88.

[37] JIN, C., FENG, D., JIANG, H., AND TIAN, L. Raid6l: A log-assisted raid6 storage architecture with improved write performance. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)* (2011), IEEE, pp. 1–6.

[38] KAASHOEK, M. F., AND KARGER, D. R. Koorde: A simple degree-optimal distributed hash table. In *IPTPS* (2003), pp. 98–107.

[39] KARGER, D. R., AND RUHL, M. Diminished chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *IPTPS* (2004), pp. 288–297.

[40] LAKSHMAN, A., AND MALIK, P. Cassandra:a structured storage system on a p2p network. In *Proc Acm Sigmod International Conference on Management of Data* (2009).

[41] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[42] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices* (1996), vol. 31, ACM, pp. 84–92.

[43] LEUNG, A. W., PASUPATHY, S., GOODSON, G. R., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *USENIX annual technical conference* (2008), vol. 1, pp. 2–5.

[44] LI, H., ZHANG, Y., LI, D., ZHANG, Z., LIU, S., HUANG, P., QIN, Z., CHEN, K., AND XIONG, Y. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), ACM, p. 15.

[45] LI, H., ZHANG, Y., ZHANG, Z., LIU, S., LI, D., LIU, X., AND PENG, Y. Parix: speculative partial writes in erasure-coded systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), USENIX Association, pp. 581–587.

[46] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

[47] LU, L., GOPALAKRISHNAN, H., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in ssd-conscious storage. *Acm Transactions on Storage 13*, 1 (2017), 5.

[48] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS) 2*, 3 (1984), 181–197.

[49] MEYER, D. T., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Parallax: virtual disks for virtual machines. In *ACM SIGOPS Operating Systems Review* (2008), vol. 42, ACM, pp. 41–54.

[50] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., KHAN, O., AND NAREDDY, K. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 257–273.

[51] MISLOVE, A., AND DRUSCHEL, P. Providing administrative control and autonomy in structured peer-to-peer overlays. In *IPTPS* (2004), pp. 162–172.

[52] MURALIDHAR, S., LLOYD, W., ROY, S., HILL, C., LIN, E., LIU, W., PAN, S., SHANKAR, S., SIVAKUMAR, V., AND TANG, L. f4: Facebook's warm blob storage system. In *Usenix Conference on Operating Systems Design and Implementation* (2014), pp. 383–398.

[53] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., , AND SUZUE, Y. Flat datacenter storage. In *OSDI* (2012).

[54] NOGHABI, S. A., SUBRAMANIAN, S., NARAYANAN, P., NARAYANAN, S., HOLLA, G., ZADEH, M., LI, T., GUPTA, I., AND CAMPBELL, R. H. Ambry:linkedin's scalable geo-distributed object store. In *International Conference on Management of Data* (2016), pp. 253–265.

[55] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J. K., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *SOSP* (2011), pp. 29–41.

[56] PIERNAS, J., CORTES, T., AND GARCÍA, J. M. Dualfs: a new journaling file system without meta-data duplication. In *Proceedings of the 16th international conference on Supercomputing* (2002), ACM, pp. 137–146.

[57] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R. M., AND SHENKER, S. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA* (2001), pp. 161–172.

[58] REN, K., ZHENG, Q., PATIL, S., AND GIBSON, G. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE Press, pp. 237–248.

[59] ROWSTRON, A. I. T., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware* (2001), pp. 329–350.

[60] SHEN, H., XU, C.-Z., AND CHEN, G. Cycloid: A constant-degree and lookup-efficient p2p overlay network. *Perform. Eval. 63*, 3 (2006), 195–216.

[61] SPIROVSKA, K., DIDONA, D., AND ZWAENEPOEL, W. Optimistic causal consistency for geo-replicated key-value stores. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 2626–2629.

[62] STODOLSKY, D., GIBSON, G., AND HOLLAND, M. Parity logging overcoming the small write problem in redundant disk arrays. In *ACM SIGARCH Computer Architecture News* (1993), vol. 21, ACM, pp. 64–75.

[63] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[64] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 357–370.

[65] WARFIELD, A., ROSS, R., FRASER, K., LIMPACH, C., AND HAND, S. Parallax: Managing storage for a million machines. In *HotOS* (2005).

[66] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 307–320.

[67] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006), IEEE, pp. 31–31.

[68] ZAHARIA, M., CHOWDHURY, M., DAS, T., AND DAVE, A. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012), pp. 1–14.

[69] ZHANG, Y., CHEN, L., LU, X., AND LI, D. Enabling routing control in a dht. *IEEE Journal on Selected Areas in Communications 28*, 1 (2009), 28–38.

[70] ZHANG, Y., LI, D., GUO, C., WU, H., XIONG, Y., AND LU, X. Cubicring: Exploiting network proximity for distributed in-memory key-value store. *IEEE/ACM Transactions on Networking 25*, 4 (2017), 2040–2053.

[71] ZHANG, Y., LI, D., AND LIU, L. Leveraging glocality for fast failure recovery in distributed ram storage. *ACM Transactions on Storage (TOS) 15*, 1 (2019), 1–24.

[72] ZHANG, Y., LI, H., LIU, S., XU, J., AND XUE, G. Pbs: An efficient erasure-coded block storage system based on speculative partial writes. *ACM Transactions on Storage (TOS) 15* (2020), 1–26.

[73] ZHANG, Y., AND LIU, L. Distributed line graphs: A universal technique for designing dhts based on arbitrary regular graphs. *IEEE Transactions on Knowledge and Data Engineering 24*, 9 (2011), 1556–1569.

[74] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (2004), 41–53.