DROP: Facilitating Distributed Metadata Management in EB-scale Storage Systems

Quanqing Xu, Rajesh Vellore Arumugam, Khai Leong Yong, Sridhar Mahadevan
Data Storage Institute, A*STAR, Singapore
{Xu_Quanqing, Rajesh_VA, YONG_Khai_Leong, Sridhar_M}@dsi.a-star.edu.sg

Abstract—**Efficient and scalable distributed metadata management is critically important to overall system performance in large-scale distributed storage systems, especially in the EB era. Traditional state-of-the-art distributed metadata management schemes include hash-based mapping and subtree partitioning. The former evenly distributes workload among metadata servers, but it eliminates all hierarchical locality of metadata. It cannot efficiently handle some operations, e.g., renaming or moving a directory that requires metadata to be migrated among metadata servers. The latter does not uniformly distribute workload among metadata servers, and metadata need to be migrated to keep the load balanced roughly. In this paper, we present a ring-based metadata management scheme, called Dynamic Ring Online Partitioning (DROP). It can preserve metadata locality using locality-preserving hashing, as well as dynamically distribute metadata among metadata server cluster to keep load balancing. By conducting performance evaluation, experimental results demonstrate the effectiveness and scalability of DROP.**

I. Introduction

In modern EB-scale storage systems [1], data is stored on a storage cluster including many servers that are directly accessed by clients via the network, while metadata is managed separately by a metadata server (MDS) cluster consisting of a number of dedicated metadata servers [2]. By separating file data access and metadata transactions, object-based storage architecture [3] is a prevalent system architecture for EB-scale storage systems. The dedicated metadata server cluster manages the global namespace and the directory hierarchy of file system, the mapping from files to objects, and the permissions of files and directories. The MDS cluster just allows for concurrent data transfers between large numbers of clients and storage servers instead of being responsible for the storage and retrieval of data. Meanwhile, it provides efficient metadata service performance with specific workloads, such as renaming a large directory near the root of the hierarchy and thousands of clients updating to the same directory or accessing the same file.

The main problem of designing metadata server cluster is how to partition metadata efficiently among MDS cluster to provide high-performance metadata services [4], [5]. Metadata server cluster is involved in moving metadata to keep metadata servers storage load balancing when MDSs are added or removed dynamically, in which the overhead of metadata migration should be minimized. In order to keep good namespace locality, some MDSs are heavily overloaded in storage load, while other MDSs are lightly overloaded. A well-designed metadata server cluster should be able to achieve satisfactory storage load balancing. In addition, we have to efficiently organize and maintain very large directories [6], each of which may contain billions of files. Internet applications such as Facebook [7] already have to manage hundreds of billions of photos. As there are millions of new files uploaded by users every day, the total number of files increases very rapidly and will soon be more than one trillion. Meanwhile, we have to provide high-performance metadata services for a large-scale storage system with hundreds of billions or trillions of files. For example, Facebook serves over one million images per second at peak, and one billion new photos per week [7].

Compared to the overall data space, the size of metadata is relatively small, and it is typically 0.1% to 1% of data space [8], but it is still large in EB-scale storage systems, e.g., 1PB to 10PB for 1EB data. Besides, 50% to 80% of all file system accesses are to metadata [9]. Therefore, in order to achieve high performance and scalability, a careful metadata server cluster architecture must be designed and implemented to avoid potential bottlenecks caused by metadata requests. To efficiently handle the workload generated by a large number of clients, metadata should be properly partitioned so as to evenly distribute metadata traffic by leveraging the MDS cluster efficiently. At the same time, to deal with the changing workload, a scalable metadata management mechanism is necessary to provide highly efficient metadata performance for mixed workloads generated by tens of thousands of concurrent clients [10]. The concurrent accesses from a large number of clients to large-scale distributed storage will cause request load imbalance among metadata servers and inefficient use of metadata cache. Caching is a popular technique to handle request load imbalance, and it is both orthogonal and complementary to the load balancing technique proposed in this paper.

In this paper, we propose a novel metadata server cluster architecture named Dynamic Ring Online Partitioning (DROP). It is a highly scalable and available key-value store using chain replication [11], and it provides a simple interface: *lookup(key)* under *put* and *get* operations. In DROP, we use locality-preserving hashing (LpH) to improves namespace locality, thus increasing put/get success rate depending on fewer MDSs and upgrading put/get performance involving fewer lookups. Maintaining metadata locality improves availability and performance of metadata substantially, but it causes storage load imbalances of MDSs. We present an efficient Histogram-based Dynamic Load Balancing (HDLB) mechanism in DROP, and we also prove the convergence of the proposed mechanism. Finally, we evaluate DROP and its competing metadata management strategies by simulations from multiple perspectives including namespace locality and load distribution, and we demonstrate that DROP converges to load balancing quickly with different MDS cluster sizes. Our results demonstrate that DROP is more effective than traditional state-of-the-art metadata management approaches, and it is also highly scalable.

The rest of the paper is organized as follows. Section II describes related work. Section III presents the system design of DROP. We describe the proposed mechanism of preserving namespace locality in Section IV, and the histogram-based dynamic load balancing mechanism in Section V. In Section VI we present performance evaluation results of DROP. In Section VII we conclude this paper.

II. Related work

Large-scale distributed storage systems must partition management of the overall file system namespace across multiple metadata servers. Subtree partitioning and hash-based mapping are two common techniques used for distributed metadata management in EB-scale storage systems, while Bloom-filter-based approaches [12], [13] provide probabilistic metadata lookups instead of metadata updates.

A. Hash-based mapping

Hash-based mapping [14], [15], [16] applies hash function to a pathname or filename of a file to locate the file's metadata. It helps clients to locate and contact directly to the right metadata server. Client requests can be distributed evenly among a metadata server cluster, eliminating hot-spots consisting of popular directories. Vesta [14], RAMA [15] and zFS [16] leverage pathname hashing to locate metadata. Hashing provides a better load balancing across metadata servers and eliminates hot-spots including popular directories. However, hashing is a random distribution, in which metadata updates may incur huge network overhead, e.g., the metadata of many files has to be migrated among MDS cluster after renaming a directory. In order to verify user access permissions, it results in high overhead from prefix directories cache or path traversal as the accessed files and their prefix directories are located on different MDSs. Furthermore, it eliminates the hierarchical locality and many benefits brought by the hierarchical locality.

Lazy Hybrid [4] based on hashing exploits lazy update policies to defer and distribute update cost to address the update issue of metadata. Metadata is not moved until it is firstly accessed when metadata migration happens because of the update. Performing the update and metadata migration in the near future avoids a sudden burst of network activities among metadata servers, but it does not reduce or eliminate a large number of metadata migrations caused by renaming a directory. It also leverages a unique dual-entry access control list structure to allow file permissions to be determined directly.

B. Subtree partitioning

Static subtree partitioning [17], [18] provides a simple approach of distributing metadata operations among metadata cluster, which statically partitions the directory hierarchy and assigns each subtree to a particular metadata server. It provides better locality of reference and greater metadata server independence than hash-based mapping. Its major drawback is that the workload may not be evenly partitioned among metadata server cluster, suffering from a system performance bottleneck. In order to adjust load imbalance, migrating subtrees is necessary in some cases (e.g., PanFS [18]). Static partitions fail to adapt to the growth or contraction of individual subtrees over time, often requiring intervention of system administrators to repartition or manually rebalance metadata storage load across metadata servers.

Dynamic subtree partitioning [5] assigns subtrees of the global namespace to different metadata servers. To handle the changing workload, it uses dynamic load balancing mechanism to redistribute metadata dynamically among MDS cluster. Its smallest unit of metadata transfer is directory (subtree), and its partition granularity is smaller and more flexible than that of static subtree partitioning. It can transfer subtrees of the busy metadata servers to other non-busy metadata servers so as to balance the workload. It is a coarse adjustment strategy that takes a long time to keep the load balanced. It eliminates bottlenecks caused by hot-spots consisting of individual files by replication. However, it can not reclaim replicas for file metadata that are not popular any more.

C. Dynamic Load Balancing

Many leave-join based load balancing mechanisms have been proposed concurrently in [19], [20], [21], [22] and [23]. [19] and [20] do not cope with skewed node range distributions, while the load balancing mechanism proposed in Mercury [23] works even when there are skewed node ranges since it utilizes an effective random sampling approach. The load balancing algorithms in [19] dynamically balance load among servers without using multiple virtual nodes by reassigning lightly loaded servers to be neighbors of heavily loaded servers. However, it is not clear whether their approaches would be efficient in practice although they prove bounds on maximum node utilization and load movement. Explicit load balancing is proposed in Mercury [23], and it is near-uniform. It ensures that routing load is uniformly distributed across all active nodes using the random sampling algorithm. One-to-Many and Many-to-Many are extended to dynamic structured P2P systems [24], where One-to-Many is used for emergency load balancing of one particularly overloaded node, while Many-to-Many is used for periodic load balancing of all nodes [21]. CFS [25] allocates each server some number of virtual nodes proportional to its capacity. It also proposes a simple solution to shed the load from an overloaded server by having the overloaded server remove some of its virtual nodes. However, this scheme may result in thrashing as removing some virtual nodes from an overloaded server may result in another node becoming overloaded.

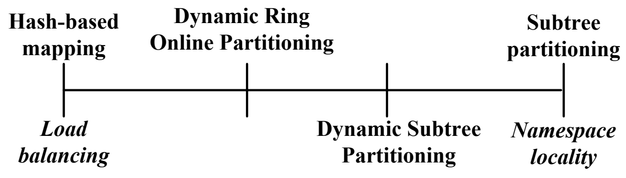


Fig. 1. Hash-based mapping vs. Subtree partitioning

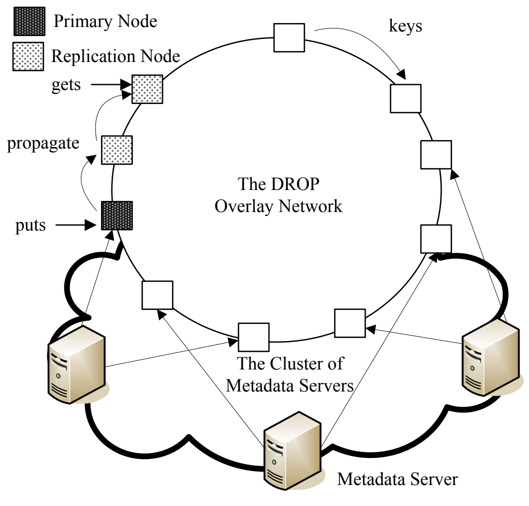


Fig. 2. System Architecture

Our work is somewhat similar to previous work, such as [21] and [24]. For example, we all use the notion of virtual nodes to balance the load, and the virtual node reassignments performed in HDLB is similar to Many-to-Many scheme [21], [24]. However, our approach is different from those proposed approaches in [21], [24] in two aspects. First, it performs virtual node reassignments as a 0-1 Multiple Knapsack Problem to keep excellent namespace locality. Second, it uses HDLB to guide virtual node reassignments such that virtual nodes are reassigned among the same group, thus reducing metadata migration overhead and achieving efficient load balancing. The proposed load balancing algorithm in this paper is effective and efficient to deal with a distributed metadata management system where metadata items are continuously inserted and deleted, and the distribution of item IDs and item sizes can be skewed.

Figure 1 depicts that hash-based mapping has a better load balancing, while subtree partitioning has a better namespace locality. Dynamic subtree partitioning (*Dynamic Subtree*) sacrifices the performance of load balancing to keep good namespace locality. On the contrary, DROP pays more attention to load balancing than namespace locality for better scalability in EB-scale storage systems. Meanwhile, it still keeps good namespace locality, which is close to that of *Dynamic Subtree*. Compared to *Dynamic Subtree*, DROP has the following advantages. Firstly, clients in DROP know exactly which MDS contains the metadata they needs, while clients cannot know how metadata is distributed in *Dynamic Subtree*, requests are directed randomly. Therefore, there are much fewer forwarded requests in DROP than in *Dynamic Subtree*. Secondly, it can adapt to a changing workload faster than *Dynamic Subtree*. The smallest unit of metadata migration is directory in *Dynamic Subtree*, so it will take a long time to achieve load balancing. DROP migrates metadata with more accurate computation results, so it can balance the relative load quickly. Finally, DROP considers the metadata migration process when the MDS cluster size changes, and moves the minimum metadata to balance the relative load. *Dynamic Subtree* does not take it into account. In addition, DROP can migrate the metadata in parallel since it is essentially a Many-to-Many scheme.

III. DROP Design

Like hash-based mapping, DROP uses hashing to distribute the metadata across the metadata server cluster. However, it also maintains hierarchical directories to support common directory hierarchy and permission semantics. It uses pathname-based locality-preserving hashing for metadata distribution and location, avoiding the overhead of hierarchical directory traversal, and maintains hierarchical directories to provide directory operations such as renaming a directory. To access data, a client hashes the pathname of the file with the same locality-preserving hash function to locate which metadata server contains the metadata of the file, and then contacts the appropriate metadata server. The process is extremely efficient metadata access, typically involving a single message to a single MDS.

A. System Architecture

The system architecture is shown in Figure 2, where a typical standard hash table evenly partitions the space of possible hash values. The hash function is random enough and many keys are sufficiently inserted, therefore those keys will be evenly distributed among the servers. Current hash-based mapping does not evenly partition the address space into which keys get mapped, causing some metadata servers get a larger portion of it. Thus, even if keys are numerous and random, some metadata servers may receive much more than others. To cope with this problem, virtual nodes are used as a means of improving load balancing [26], i.e., each real metadata server pretends to be several distinct MDSs (i.e., virtual nodes), each participating independently in the DROP network, thus its load is determined by summing over several virtual nodes'.

A real MDS has to allocate storage space for each virtual node to store necessary data structures. It means that more virtual nodes need more data structure space, leading to better namespace locality and load balancing. Compared to Chord [26] by allocating $\log N$ virtual nodes per physical server[1], DROP can achieve better namespace locality and load balancing by allocating more than $\log N$ virtual nodes per metadata server since metadata IDs are not uniformly distributed and metadata items may have the different size. The data structures are typically not so expensive from the perspective of space, thus it is not a serious problem. We

[1] N is the number of nodes in the system.

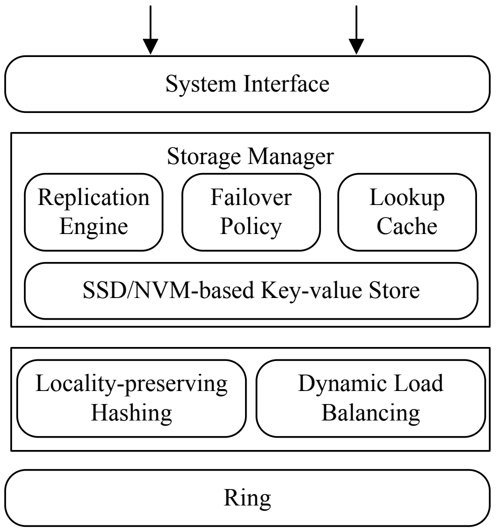


Fig. 3. The Architecture of Metadata Server

have to consider a much more significant problem arising from network bandwidth. In general, to maintain connectivity of the network, every virtual node frequently pings its neighbors to make sure them still alive, and replaces them with new neighbors if they are not alive any more. In order to maintain the DROP network, there is a multiplicative increase in network traffic because of running multiple virtual nodes in each real MDS, but the DROP network is located in a data center that has enough bandwidth.

B. MDS Architecture

The architecture of a single MDS is shown in Figure 3, and it is a part of the ring-based structure. Storage Manager administers a SSD/NVM-base key-value storage system under three components, i.e., Replication Engine, Failover Policy and Lookup Cache. Replication Engine implements the redundancy mechanisms and determines how to put or get metadata in the MDS cluster, and it is invoked when reading and writing metadata. Replication Engine exploits chain replication [11] that is used to support large-scale storage services that exhibit high throughput and availability with strong consistency. In chain replication, updates are sent to the head of the chain and passed along to each member of the chain, while queries are sent to the tail of the chain. It provides a simple system interface, *lookup* under *put* and *get* operations.

DROP balances storage load using Histogram-based Dynamic Load Balancing (HDLB) that we will talk about in Section V, while request load (i.e., the number of metadata items accessed in a MDS) is also important because some files may be accessed more than others. To balance storage load in DROP is orthogonal to balance request load using *Lookup Cache*, so DROP alleviates temporary hot spots using retrieval caches in server side, thereby balancing both storage and request load. Lookup Cache is a cache mechanism to find hot-spots, which are replicated to eliminate bottlenecks via Replication Engine.

IV. Preserving Namespace Locality

In large-scale storage systems, we can achieve near-optimal namespace locality by assigning keys that are consistent with the order of full pathnames.

A. Locality-preserving Hashing

To attain near-optimal locality, the entire directory tree nested under a point has to reside on the same MDS if there is not an explicit subtree assignment, e.g., */usr/lib* may be assigned to one MDS, while */usr/lib/grub* may be assigned to another. Path names are directly used with fixed-size keys, where every lookup message should contain a key as large as the longest path. In order to limit message overhead without modifying routing mechanisms, we use a more compact key encoding in DROP as shown in Figure 4.

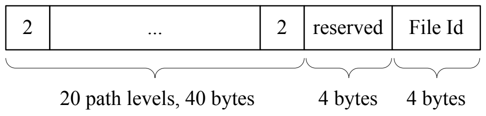


Fig. 4. Locality-preserving Hashing

The locality-preserving hash function is shown in Algorithm 1. The file path is encoded with the first 40 bytes, and each directory is encoded with 2 bytes. For longer paths, the next 4 bytes are reserved for the rest of path since 40 bytes are only sufficient for 20 path levels in terms of space (lines 4-9). Each component in the file path is encoded using simhash [27], [28] (lines 5,8,10). Although locality for files in longer paths will not be preserved, they make up 0.042%, 0.018% and 0.0% of the files in Linux, Microsoft Windows build server production and Harvard traces, and there are an even smaller percentage of the accesses. We plot the cumulative distribution function (CDF) of path length in the three analyzed traces in Figure 5, and we can see that the longer the path length is, the smaller the proportion is. The next 4 bytes are allocated for a file name (line 10), and they can represent 2^{32} files per directory in theory. Eventually, the 48-byte key enables up to many exabytes in size.

The key encoding mechanism provides a good trade-off between key size and file count, and it enables naming of

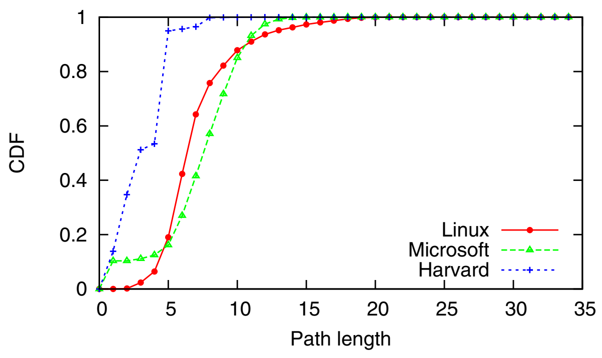


Fig. 5. CDF of path length in three traces

Algorithm 1: Locality-preserving Hashing

```
Input: A file path $filePath$
Output: The file path's hash value $key$
1  $ks$ = [];
2  Split $filePath$ into $parts$;
3  $pathDepth$ = min(len($parts$), $D$); // D=20+2
4  for $part \in parts[: pathDepth - 1]$ do
5      $h$ = simhash($part$, 16);
6      Insert bytes in $h$ into $ks$ from high to low;
7  if $pathDepth < D$ then
8      $h$ = simhash('', 16*$D$-16*($pathDepth$-1));
9      Append bytes in $h$ to $ks$ from high to low;
10 $h$ = simhash($parts$[-1], 32);
11 Append bytes in $h$ to $ks$ from high to low;
12 $key$ = 0;
13 for $i \in range(0, len(ks))$ do
14     $key$ |= $ks$[i] << 8*(len($ks$)-i-1);
15 return $key$;
```

new files and directories. In addition, a file may be moved to a different directory, and its key can be quickly changed to reflect the new path using the encoding mechanism. Furthermore, related metadata items are organized into a group using it to preserve in-order traversal of file system, e.g., files in the same directory are related.

B. Metadata Publishing

Metadata publishing is the process of making files and directories metadata available to clients, and it is the foundation upon which EB-scale storage systems are being built. Therefore, we must take care in metadata publishing to ensure the structural integrity of the storage systems built on top of it. It benefits both storage systems and clients, such as making system building easier and facilitating federated search for clients. The metadata distribution of a directory is illustrated in Figure 6. Each metadata server maintains one or more hierarchical directory metadata, which is located by the directory pathname using Algorithm 1. Meanwhile, a file metadata is stored and located in a specific MDS by hashing the file pathname with this locality-preserving hashing function.

Metadata publishing by locality-preserving hashing improves overall performance and availability, but it also may cause two problems. The first one is that the files of identical content with different paths will be stored on multiple MDSs because of their different keys. By contrast, they would only be stored once if they were addressed using content hashes since their hash values would be the same. The increased storage cost is necessary for preserving namespace locality since the same file may be stored in multiple places. The second one is that the failure of a replica group (i.e., consecutive metadata servers) in DROP leads to the unavailability of *most* of metadata for *a few* clients. On the contrary, the same failure in traditional hash-based mapping results in the unavailability

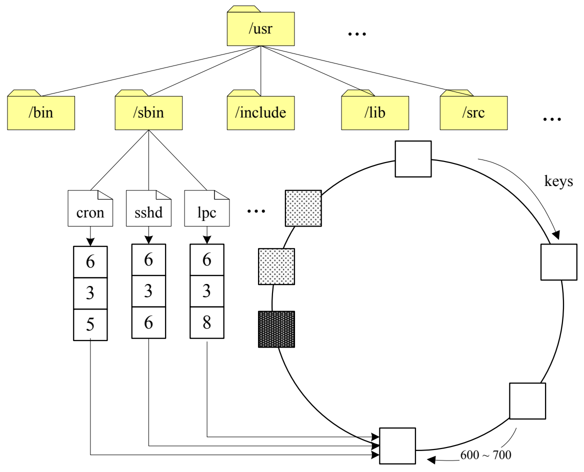


Fig. 6. Metadata Publishing

of *some* data for *many* clients. The trade-off is unavoidable since a much lower user-perceived failure rate for all clients happens due to isolating failures to impact fewer clients in a typical storage system.

V. Histogram-based Dynamic Load Balancing

Since the key distribution is no longer uniform in DROP, load balancing is a great challenge that we have to face and address. In this section, we propose a simple but efficient dynamic load balancing algorithm to guarantee load balancing with losing negligible locality in metadata placement. The load balancing algorithm is simple, fully distributed, and converge quickly, which is similar to other dynamic load balancing methods [19], [23].

A. Metadata Histogram Maintenance

In order to achieve load balancing, a simple yet efficient metadata histogram maintenance mechanism is firstly proposed as shown in Algorithm 2. It is used by MDSs for maintaining histograms of metadata storage load. Its basic idea is to measure the range-density histogram locally and exchange these histograms throughout the system in a heartbeat protocol.

Let $\mathbb{N}$ denote the neighbor metadata server set of a MDS. Each MDS periodically samples MDSs in $\mathbb{N}$ and produces a local estimate of metadata items. Each of these MDSs reports back its local range-density. As time progresses, a metadata server builds a list of tuples of this form: $\{virtual\_node, [min, max], load, timestamp\}$, where the timestamp is used to age out old records.

If the DROP system needs to get an average histogram of node range-density, the collected range-densities can be used exactly as they are collected. When a MDS joins the cluster, it is required to have a duty for some ranges of locality-preserving hash values via its virtual nodes. However, files and directories in a particular range of values may exhibit a much greater popularity than other ranges. This would cause the MDS responsible for the popular range to become overloaded.

Algorithm 2: Metadata Histogram Maintenance

```
  Input: A MDS Cluster $S$
  Output: Light MDSs $S_l$, Heavy MDSs $S_h$
1 forall the $s \in S$ do
2     $s$ measures range-density histogram locally;
3     $s$ gossips about the histogram;
4 Find heavy MDSs $S_h$ and light MDSs $S_l$;
5 return [$S_l$, $S_h$];
```

We leverage our metadata histogram maintenance mechanism to help implement load balancing in DROP. Firstly, each metadata server can get the average load $\overline{L}$ using histograms, thus determining if it is relatively heavily or lightly loaded in the system. Secondly, the histograms contain information about which parts of the overlay are lightly loaded. Using this information, heavily loaded MDSs can send probes to lightly loaded parts of the network. Once a probe encounters a lightly loaded MDS, it requests this lightly loaded MDS to gracefully take some virtual nodes from the heavily loaded MDSs in DROP, which effectively decreases the load of the heavily loaded MDSs. Note that this process is in parallel.

B. Load Balancing

We present a simple but effective Histogram-based Dynamic Load Balancing (HDLB) approach. Each metadata server periodically contacts its neighbors in the system. The DROP system is said to be load-balanced when all the MDSs satisfy Definition 1, i.e., if the largest load is greater than t^2 times the smallest load, dynamic load balancing will be performed by reassigning virtual nodes to from heavily loaded MDSs to lightly loaded MDSs.

Definition 1 (MDS_i is load balancing). MDS_i is load balancing if its load satisfies $1/t \leq L_i/\overline{L} \leq t$ ($t \leq 2$).

Given a set of m metadata servers $S = \{s_i, i = 1, \ldots, m\}$ and a set of n virtual nodes $V = \{v_j, j = 1, \ldots, n\}$, each virtual node v_j has a weight w_j that means how many files in a range are maintained by v_j, and each metadata server s_i has a remaining capacity (weight) W_i that means the difference between the average storage load (capacity) $\overline{W}$ and the existing weight in the metadata server s_i. The problem can be formulated as a 0-1 Multiple Knapsack Problem [29] (MKP) that is a NP-hard problem, i.e., it is to determine how to reassign n virtual nodes to m metadata servers in a way that minimizes the wasted space in the MDSs as follows:

$$\text{maximize} \qquad z = 1/\sum_{i=1}^{m} s_i \tag{1a}$$

$$\text{s.t.} \qquad \sum_{i=1}^{m} x_{ij} = 1, j \in N = \{1, \ldots, n\} \tag{1b}$$

$$\sum_{j=1}^{n} w_j x_{ij} + s_i = W_i y_i, i \in M = \{1, \ldots, m\} \tag{1c}$$

$$x_{ij} \in \{0, 1\}, y_i \in \{0, 1\}, i \in M, j \in N \tag{1d}$$

where

$$x_{ij} = \begin{cases} 1 & \text{if virtual node } j \text{ is reassigned to MDS } i \\ 0 & \text{otherwise} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{if MDS } i \text{ is used} \\ 0 & \text{otherwise} \end{cases}$$

$$s_i = \text{space left in MDS } i$$

Constraint (1b) makes sure that each virtual node is only assigned to a real metadata server. Constraint (1c) ensures that the total number of files assigned to each metadata server is less than the capacity of metadata server. Constraint (1d) states it is a 0-1 knapsack problem.

We use $t = 2$ so that MDS loads differ by at most a factor of 4 in steady state. Each MDS stores both primary and secondary replicas, but only the primary replica count is used as the load value for the purpose of this approach. When primary load on all MDSs is balanced, then total load, including both primary and secondary replicas, will be balanced as well. For example, there is a metadata server A, which has three neighbors B, C and D. They include virtual nodes as shown in Table I, where the number means the load of a virtual node. There are a set of virtual nodes $V = \{3, 2, 7, 6, 2\}$ that will be reassigned to light MDSs $S = \{C, D\}$, which have the remaining capacities 11 and 12 respectively. After solving the 0-1 MKP, we can see that there is a rough load balancing from Table I.

TABLE I
EXAMPLE LOAD

MDS	range-densities	removed range-densities	results
A	{3, 2, 7, 12}	{3, 2, 7}	{12}
B	{15, 6, 2}	{6, 2}	{15}
C	{1}	∅	{2, 7, 2}
D	∅	∅	{3, 6}

C. Convergence

We proceed to prove the convergence of histogram-based dynamic load balancing. All nodes (MDSs) in the DROP network consist of a sequence of graphs (G_n) including G_n and its subgraphs. we introduce three important concepts: *Cut Norm*, *Distance between G and G′* and *Cut Metric between Arbitrary Graphs* as follows.

Definition 2 (Cut Norm). Let M be an $n \times n$ matrix, and the cut norm of M is given by

$$\|M\|_\square = \max_{i,j} |\sum M_{ij}|$$

Definition 3 (Distance between G and G'). The distance between G and G' is given by

$$d_\square(G, G') = \frac{1}{v^2}\|A_G - A_G^{'}\|_\square = \frac{1}{v^2}\max_{i,j} |\sum v_i v_j (e_{ij} - e_{ij}^{'})|$$

where G and G' are weighted graphs with n common vertex weights $v_1, \ldots, v_n$, s.t. $\sum_i v_i = v$, different edge weights e_{ij} and $e_{ij}^{'}$, and adjacency matrices A_G with $(A_G)_{ij} = v_i e_{ij} v_j$ and $(A_{G'})_{ij} = v_i e_{ij}^{'} v_j$.

Definition 4 (Cut Metric between Arbitrary Graphs). The cut metric between arbitrary graphs is defined as follows

$$\delta_\square(G, G') = \min_X d_\square(G[X], G'[X]))$$

where G and G' are weighted graphs with n and n' common vertex and edge weights v_i, e_{ij} and $v_s^{'}$, $e_{st}^{'}$, the minimum goes over all fractional overlays: all couplings X of v_i and $v_i^{'}$, with $X_{is} \geq 0$, $\sum_s X_{is} = v_i$ and $\sum_i X_{is} = v_s^{'}$.

Definition 3 can easily be generalized to a weighted graph G and its variant $\breve{G}$ with good load balancing in DROP. Theorem 1 tells that all the MDSs in the DROP network are able to eventually achieve load balancing after running the histogram-based dynamic load balancing algorithm.

Theorem 1. Let (G_n) be a sequence of graphs of all nodes in DROP. Then the graphs in the sequence (G_n) can be relabeled in such a way that the resulting sequence $(\breve{G}_n)$ of labeled graphs converges to $\overline{W}$ with average node weights, i.e., $\|W_{\breve{G}_n} - \overline{W}\|_\square \to 0$.

Proof: In the DROP network, there are obviously no dominant nodes ($\frac{\max_i \alpha_i(G_n)}{\alpha_{G_n}} \to 0$ as $n \to \infty$), and edge weights are either 1 or -1, so (G_n) is a convergent sequence of weighted graphs with no dominant nodes and uniformly bounded edge weights. $W_{G_n} \to \overline{W}$ in the $\delta_\square$ distance, i.e., $\delta_\square(W_{G_n}, \overline{W}) \to 0$ by the Weak Regularity Lemma [30], so we can approximate (G_n) by a sequence of graphs $(\breve{G}_n)$ with node weights one. Without loss of generality that all graphs in the sequence (G_n) have total node weight $\alpha_{G_n} = 1$. Define $a_n = \max_i \alpha_i(G_n)$, and choose ε_n in such a way that $\varepsilon_n \to 0$ and $a_n 2^{40/\varepsilon_n^2} \to 0$ as $n \to \infty$. We can construct a partition $\mathbb{P}_n$ of $V(G_n)$ into $q_n \leq 2^{20/\varepsilon_n^2}$ classes such that $d_\square(G_n, (G_n)_{\mathbb{P}_n}) \to 0$ and the classes in $\mathbb{P}_n$ have almost equal node weights, i.e., $|\sum_{x \in V_i} \alpha_x(G_n) - \sum_{y \in V_j} \alpha_y(G_n)| \leq a_n$ for all $i, j \in [q_n]$. Consider the sequence of graphs $\breve{G}_n$ from $G_n/\mathbb{P}_n$ by changing all node weights to 1. Since the classes of $\mathbb{P}_n$ have almost equal weights, we have $\|W_{\breve{G}_n} - W_{G_n/\mathbb{P}_n}\|_\square \leq q_n^2 a_n \to 0$, i.e., $\|W_{\breve{G}_n} - \overline{W}\|_\square \to 0$. ■

D. Traffic Control

During load balancing, a metadata item may be moved multiple times. It often occurs when some files in a large directory are renamed since the directory initially is assigned to a single MDS with a high probability. DROP uses metadata pointers to minimize metadata migration overhead. For a metadata pointer, a MDS retrieves the metadata when it has held the pointer for longer than the pointer's stabilization time. Using metadata pointers only temporarily hurts data locality when balancing the load. Besides reducing load balancing overhead, pointers also enable writes to succeed even when the target MDS is at capacity, pointers can be used to divert metadata items from heavily loaded MDSs to lightly loaded MDSs. However, the MDS at capacity will eventually shed some load when balancing the load, just causing temporary additional indirection. Suppose that a MDS *X* is heavily loaded, and a MDS *Y* takes some virtual nodes of *X* to take some of *X*'s load. Now *X* must transfer some of its metadata items to *Y*. Instead of having *X* immediately transfer some of its metadata items to *Y* when *Y* gets some virtual nodes from *X*, *Y* will initially maintain metadata pointers to *X*. Later *Y* will transfer the pointers to *Z*, and *Z* will ultimately retrieve the actual metadata from *X* and delete the pointers.

There is an example of convergence of HDLB as shown in Figure 7, in which there is load imbalance caused by locality-preserving hashing. There are ten MDSs, and an arbitrary network topology with three hub MDSs (2, 3 and 6) that are heavily overloaded. Note that any MDS must connect one of other metadata servers. After solving the 0-1 Multiple Knapsack Problem via hub MDSs, we can see there is a good load balancing in Figure 7. Some virtual nodes are very heavily overloaded, and they have to be split into two or more virtual nodes. In this example, there are three virtual nodes that are split into two virtual nodes. Note that there is no metadata migration at the beginning since we can assign virtual nodes to MDSs after finishing the computation task.

VI. Performance Evaluation

In this section, we present a detailed performance evaluation of DROP using simulations. We have developed a detailed event-driven simulator to validate and evaluate our design decisions and choices. We firstly empirically evaluate the namespace locality effectiveness, and secondly present the convergence rate of HDLB. In the third and fourth parts, we measure the load distribution of DROP and the scalability of DROP. Lastly, we also measure the metadata migration overhead of DROP. We evaluate DROP and compare its performance with other distributed metadata management schemes: 1) Subtree that is to manually partition directories and assign each subtree to a metadata server; 2) FileHash that is to randomly distribute files according to their pathnames, each of which is assigned to a metadata server; 3) DirHash that is to randomly distribute directories like FileHash.

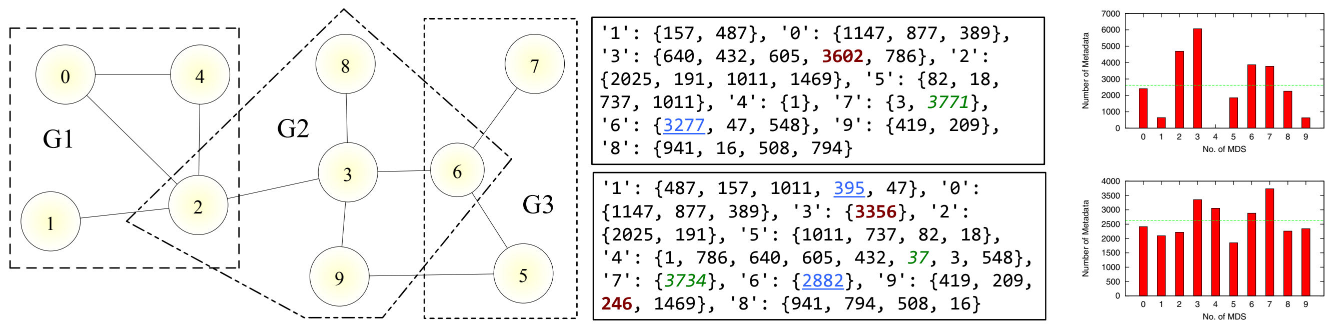
We demonstrate the effectiveness, performance and scalability over different MDS cluster sizes. A virtual node's identifier is a 384-bit key obtained from the SHA-384 hash function. There are three real traces we study as shown in Table II. The *Linux* trace is from four Linux machines using a crawler. *Microsoft* means Microsoft Windows build server production traces from BuildServer00 to BuildServer07 within 24 hours, and its data size is 223.7GB. Finally, *Harvard* is a research and email NFS trace used by a large Harvard research group, and its data size is 158.6GB.

TABLE II
Traces

Trace	# of unique files	Path metadata	Maximum length
Linux	2,216,596	147M	22
Microsoft	7,725,928	416M	34
Harvard	7,936,109	176M	18

A. Namespace Locality

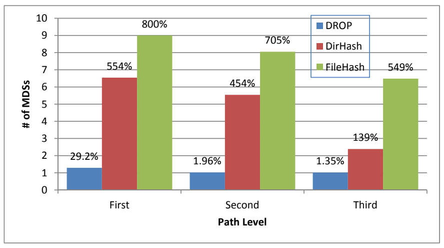
Locality-preserving hashing in DROP is clearly a suboptimal strategy to keep excellent namespace locality. Namespace



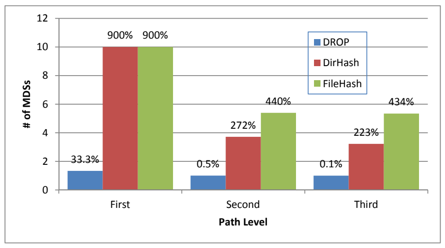
```
'1': {157, 487}, '0': {1147, 877, 389},
'3': {640, 432, 605, 3602, 786}, '2':
{2025, 191, 1011, 1469}, '5': {82, 18,
737, 1011}, '4': {1}, '7': {3, 3771},
'6': {3277, 47, 548}, '9': {419, 209},
'8': {941, 16, 508, 794}
```

```
'1': {487, 157, 1011, 395, 47}, '0':
{1147, 877, 389}, '3': {3356}, '2':
{2025, 191}, '5': {1011, 737, 82, 18},
'4': {1, 786, 640, 605, 432, 37, 3, 548},
'7': {3734}, '6': {2882}, '9': {419, 209,
246, 1469}, '8': {941, 794, 508, 16}
```

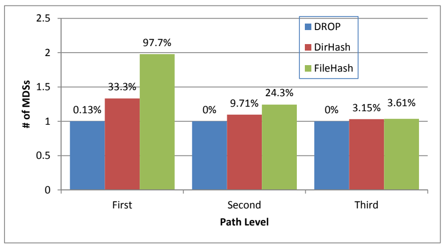
Fig. 7. An Example of Convergence of HDLB



(a) Linux trace



(b) Microsoft Windows trace



(c) Harvard trace

Fig. 8. Locality Comparisons of Three Level Paths over Different Traces in the Cluster of 10 MDSs

locality is very important to large-scale distributed metadata management, and it is utilized to improve the performance of MDS cluster by reducing I/O requests. It can be measured: $locality = \sum_{j=1}^{m} p_{ij}$, where p_{ij} (0 or 1) represents if a subtree path p_i ($\in \mathbb{P}$) is located in MDS j. The metric represents how many metadata servers the path $\mathbb{P}$ is split across. Figure 8 presents namespace locality comparisons of three level paths on three traces using three different distributed metadata management mechanisms. Note that we do not plot the results of static subtree partitioning since each path is maintained by only one metadata server according to its definition, but its load imbalance is severe.

Figure 8(a), Figure 8(b) and Figure 8(c) illustrate that DROP has much better namespace locality than DirHash and FileHash for the given three traces. The percentage above a box is calculated as follows: $\frac{N-S}{S} \times 100\%$, where S is the number of MDSs using Subtree (S=1), N is the number of MDSs using one of other three approaches. DROP performs only negligibly worse than static subtree partitioning except the first level paths in both the *Linux* trace and the *Microsoft* Windows trace. The reason for this is that DROP can achieve suboptimal namespace locality using locality-preserving hashing, i.e., assigning keys that are consistent with the order of pathnames. For both DirHash and FileHash, the order of pathnames is not considered so that namespace locality is lost thoroughly.

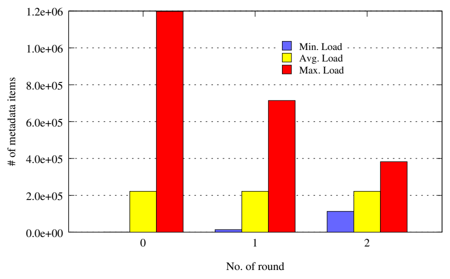
B. Convergence Rate

We have proved the convergence of HDLB as shown in Theorem 1. However, the convergence rate is also critically significant in distributed systems. Figure 9 depicts the convergence rate of HDLB on three traces in the cluster of 10 metadata servers. We can see that there are only two rounds to converge to load balancing on the first two traces as shown in Figure 9(a) and Figure 9(b), there is only one round to converge to load balancing on the *Harvard* trace as shown in Figure 9(c) because there are much more directories with fewer files in the *Harvard* trace than in both the *Linux* and *Microsoft* Windows traces, and the system is easy to reach load balancing.

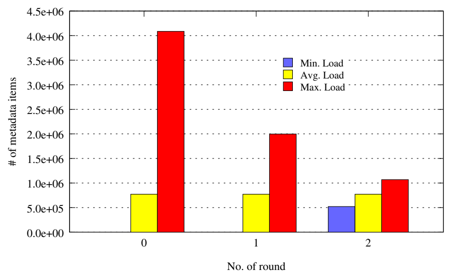
We define *Load Factor* as follows: $LoadFactor = \frac{Max.Load}{Min.Load}$. In Figure 9(a), minimum load and maximum load are 27 and 1,198,073 metadata items in the *Linux* trace after running locality-preserving hashing, and the *Load Factor* is 44,373.07. In round 1 and round 2, the *Load Factors* are 50.72 and 3.38 after solving the 0-1 Multiple Knapsack Problem. Our dynamic load balancing approach can quickly converge to an ideal load balancing state on the *Linux* trace as shown in Figure 9(a). Similarly, it is also quickly convergent on other two traces as shown in Figure 9(b) and Figure 9(c).

C. Load Distribution

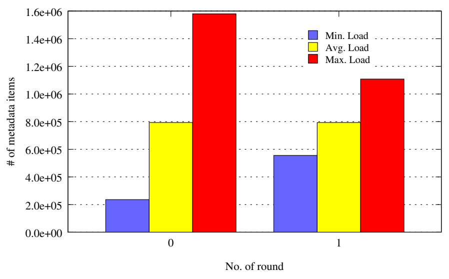
In this experiment, we compare DROP with other three approaches. Figure 10 illustrates the number of metadata items in each metadata server with different metadata management methods. From Figure 10(a), Figure 10(b) and Figure 10(c), we can see that FileHash based on the hash value of the file pathname is the best among all four methods, in which metadata items are evenly assigned to each metadata server. Subtree is the worst one, where load imbalance is greatly bad. Note that we loose the constraint of Subtree on the *Microsoft* trace: the unit is the second-level path instead of the first-level path because there are only three first-level paths: Disk1, Disk2 and Disk3. Both DirHash and DROP have good load balancing where metadata items are relatively uniform distributed to each MDS. The *Load Factors* of DirHash are 1.19, 1.25 and 1.93



(a) Linux trace

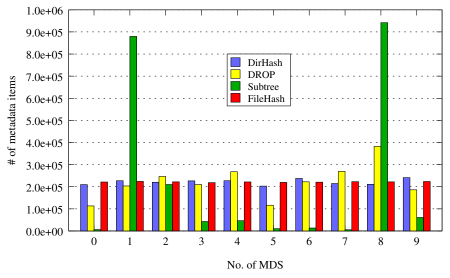


(b) Microsoft Windows trace

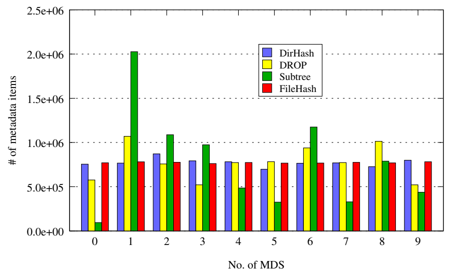


(c) Harvard trace

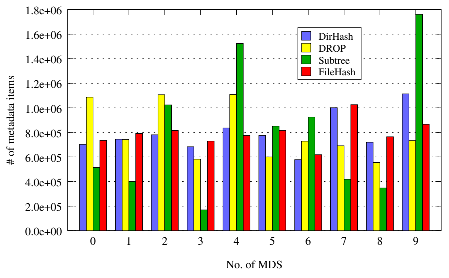
Fig. 9. Convergence Rate of Histogram-based Dynamic Load Balancing in the Cluster of 10 MDSs



(a) Linux trace



(b) Microsoft Windows trace



(c) Harvard trace

Fig. 10. Load Distribution over Different Traces in the Cluster of 10 MDSs

on *Linux*, *Microsoft* and *Harvard*, while the *Load Factors* of DROP are 3.38, 2.05 and 2.00 on the three traces.

D. Scalability

The primary overhead of DROP's performance gains comes from active load balancing. In this experiment, we evaluate the relative performance and scalability of DROP by scaling the number of MDSs. Figure 11 presents the relative performance with varying the number of MDSs. Figure 11(a) demonstrates that histogram-based dynamic load balancing (HDLB) has excellent convergence rate. For the given three traces, it reaches a satisfactory load balancing state within four rounds even as the number of MDSs is 40. Therefore, HDLB can quickly converge to load balancing in fully distributed systems.

Figure 11(b) shows that HDLB has excellent load balancing performance with different MDS cluster sizes. Figure 11(c) illustrates that HDLB has excellent efficiency with different numbers of metadata servers. The histogram-based dynamic load balancing mechanism can effectively assign and migrate loads among metadata servers. A large faction of loads are reassigned and migrated firstly within the same group and lastly within the entire network via hub MDSs, thus enabling fast and efficient load balancing.

E. Metadata Migration Overhead

As files are added, modified and deleted, or MDSs join and depart, the key distribution of files in the system changes, and DROP may have to migrate metadata in order to maintain load balancing. It brings two questions: the first one is how DROP can balance the storage load over time, and the second one is how much metadata migration overhead is required to keep the storage load still balanced. The previous experiment has answered the first question perfectly.

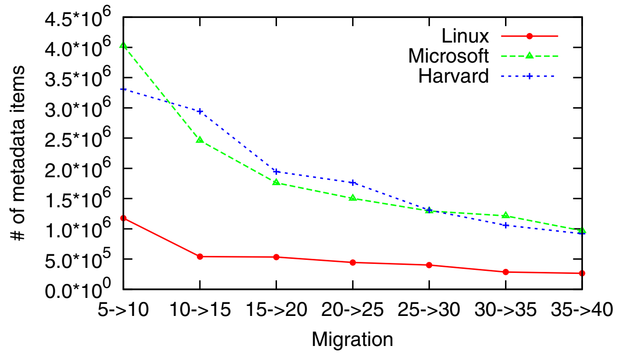
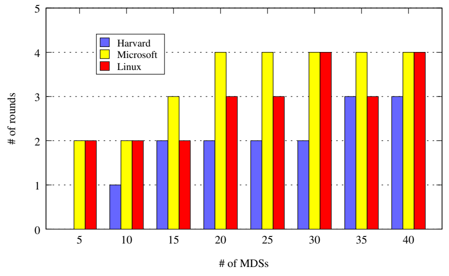


Fig. 12. Migration Overhead

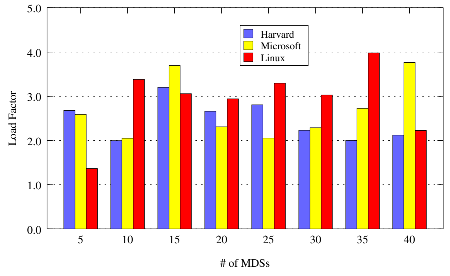
Figure 12 shows the metadata migration overhead with excellent scalability. We perform this experiment as follows. All the MDSs in the DROP system are in a satisfactory load balancing state at the beginning. Five MDSs join the system randomly, and the system will reach a new load balancing state. We investigate how many metadata items are migrated into new MDSs. As we talk about the scalability of HDLB in the previous subsection, HDLB tries, at each step, to reduce the metadata migration overhead by making virtual node assignments among the same group.

VII. Conclusions and Future Work

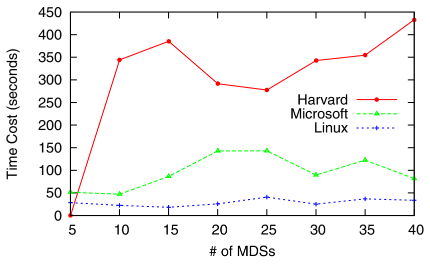
In this paper, we present Dynamic Ring Online Partitioning (DROP), a dynamic and scalable distributed metadata management architecture to serve EB-scale storage systems. In order to keep excellent namespace locality, DROP exploits locality-preserving hashing to distribute metadata among the metadata servers. When storage load changes dynamically, DROP introduces the Histogram-based Dynamic Load Balancing (HDLB) strategy to quickly adjust the metadata distribu-



(a) Convergence Rate



(b) Load Balancing



(c) Time Cost

Fig. 11. Performance with Varying the Number of MDSs

tion. After the adjustment, DROP ensures that the namespace locality maintained by MDSs is still good. Besides, DROP can balance the metadata storage load as good as static hash-based mapping. When the size of the MDS cluster changes, DROP uses the HDLB strategy to move the minimal metadata to maintain the storage load balancing. Compared to other distribute metadata management techniques, DROP brings multiple advantages, such as balancing metadata storage load efficiently, high scalability and no bottlenecks with negligible additional overhead.

In future, we are going to design and implement a hybrid lookup cache architecture to manage the whole life cycle for all hot-spots in storage systems, which potentially reduces metadata request loads. The hybrid cache architecture consists of DRAM and MRAM since MRAM is available in quantity. Meanwhile, MRAM is used to keep the consistency of metadata by logging updates, like NetApp WAFL [31]. We are also going to integrate DROP into our distributed object-based storage system.

Acknowledgment

The authors would like to thank Garth Gibson from Carnegie Mellon University, Jun Wang from University of Central Florida, Yafei Dai from Peking University and Mingzhong Xiao from Beijing Normal University for helpful comments. This work is supported by A*STAR Thematic Strategic Research Programme (TSRP) Grant No. 1121720013.

References

[1] I. Raicu, I. T. Foster, and P. Beckman, “Making a case for distributed file systems at exascale,” in *LSAP*, 2011, pp. 11–18.

[2] G. A. Gibson and R. V. Meter, “Network attached storage architecture,” *Commun. ACM*, vol. 43, no. 11, pp. 37–45, 2000.

[3] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in *OSDI*, 2006, pp. 307–320.

[4] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue, “Efficient metadata management in large distributed storage systems,” in *IEEE Symposium on Mass Storage Systems*, 2003, pp. 290–298.

[5] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, “Dynamic metadata management for petabyte-scale file systems,” in *SC*, 2004, p. 4.

[6] S. Patil and G. A. Gibson, “Scale and concurrency of giga+: File system directories with millions of files,” in *FAST*, 2011, pp. 177–190.

[7] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, “Finding a needle in haystack: Facebook’s photo storage,” in *OSDI*, 2010, pp. 47–60.

[8] E. L. Miller, K. Greenan, A. Leung, D. Long, and A. Wildani. (2008) Reliable and efficient metadata storage and indexing using nvram. [Online]. Available: dcslab.hanyang.ac.kr/nvramos08/EthanMiller.pdf

[9] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. D. Kupfer, and J. G. Thompson, “A trace-driven analysis of the unix 4.2 bsd file system,” in *SOSP*, 1985, pp. 15–24.

[10] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, “Scalable performance of the panasas parallel file system,” in *FAST*, 2008, pp. 17–33.

[11] R. van Renesse and F. B. Schneider, “Chain replication for supporting high throughput and availability,” in *OSDI*, 2004, pp. 91–104.

[12] Y. Zhu, H. Jiang, J. Wang, and F. Xian, “Hba: Distributed metadata management for large cluster-based storage systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 6, pp. 750–763, 2008.

[13] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, “Supporting scalable and adaptive metadata management in ultralarge-scale file systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 580–593, 2011.

[14] P. F. Corbett and D. G. Feitelson, “The vesta parallel file system,” *ACM Trans. Comput. Syst.*, vol. 14, no. 3, pp. 225–264, 1996.

[15] E. L. Miller and R. H. Katz, “Rama: An easy-to-use, high-performance parallel file system,” *Parallel Computing*, vol. 23, no. 4-5, pp. 419–446, 1997.

[16] O. Rodeh and A. Teperman, “zfs - a scalable distributed file system using object disks,” in *IEEE Symposium on Mass Storage Systems*, 2003, pp. 207–218.

[17] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, “Nfs version 3: Design and implementation,” in *USENIX Summer*, 1994, pp. 137–152.

[18] D. Nagle, D. Serenyi, and A. Matthews, “The panasas activescale storage cluster - delivering scalable high bandwidth storage,” in *SC*, 2004, p. 53.

[19] D. R. Karger and M. Ruhl, “Simple efficient load balancing algorithms for peer-to-peer systems,” in *SPAA*, 2004, pp. 36–43.

[20] P. Ganesan, M. Bawa, and H. Garcia-Molina, “Online balancing of range-partitioned data with applications to peer-to-peer systems,” in *VLDB*, 2004, pp. 444–455.

[21] A. Rao, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica, “Load balancing in structured p2p systems,” in *IPTPS*, 2003, pp. 68–79.

[22] J. Pang, P. B. Gibbons, M. Kaminsky, S. Seshan, and H. Yu, “Defragmenting dht-based distributed file systems,” in *ICDCS*, 2007, p. 14.

[23] A. R. Bharambe, M. Agrawal, and S. Seshan, “Mercury: supporting scalable multi-attribute range queries,” in *SIGCOMM*, 2004, pp. 353–366.

[24] B. Godfrey, K. Lakshminarayanan, S. Surana, R. M. Karp, and I. Stoica, “Load balancing in dynamic structured p2p systems,” in *INFOCOM*, 2004.

[25] F. Dabek, M. F. Kaashoek, D. R. Karger, R. Morris, and I. Stoica, “Wide-area cooperative storage with cfs,” in *SOSP*, 2001, pp. 202–215.

[26] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *SIGCOMM*, 2001, pp. 149–160.

[27] M. Charikar, “Similarity estimation techniques from rounding algorithms,” in *STOC*, 2002, pp. 380–388.

[28] G. S. Manku, A. Jain, and A. D. Sarma, “Detecting near-duplicates for web crawling,” in *WWW*, 2007, pp. 141–150.

[29] C. Chekuri and S. Khanna, “A polynomial time approximation scheme for the multiple knapsack problem,” *SIAM J. Comput.*, vol. 35, no. 3, pp. 713–728, 2005.

[30] A. Frieze and R. Kannan, “Quick approximation to matrices and applications,” *Combinatorica*, vol. 19, pp. 175–220, 1999.

[31] D. Hitz, J. Lau, and M. A. Malcolm, “File system design for an nfs file server appliance,” in *USENIX Winter*, 1994, pp. 235–246.