LoADM: Load-aware Directory Migration Policy in Distributed File Systems

Yuanzhang Wang[†], Peng Zhang[†], Fengkui Yang[†], Ke Zhou[†], Chunhua Li[†]*
[†]Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
{yuanzhangw, zhangpeng19, fkyang, zhke, li.chunhua}@hust.edu.cn
*Corresponding author: Chunhua Li (li.chunhua@hust.edu.cn)

***Abstract*—Distributed file systems often suffer from load imbalance when encountering skewed workloads. A few directories can become hotspots due to frequent access. Failure to migrate these high-load directories promptly will result in node overload, which can seriously degrade the performance of the system. To solve this challenge, in this paper, we propose a novel load-aware directory migration policy named LoADM to alleviate the load imbalance caused by hot directories. LoADM consists of three parts, i.e. learning-based directory hotness model, urgency analysis and multidimensional directory migration model. Specifically, we use a directory hotness model to identify potentially high-load directories in advance. Second, by combining the predicted directory hotness and system node status, the urgency analysis determines when to trigger a migration or tolerate an imbalance. Then, peer directory co-migration is proposed to better exploit data locality. Finally, we migrate high-load directories to appropriate storage nodes through a Particle Swarm Optimization based directory migration model. Extensive experiments show that our approach provides a promising data migration policy and can greatly improve performance compared to the state-of-the-art.**

***Index Terms*—DFS, data migration, machine learning, load balance**

I. Introduction

Data distribution comes with a cost. Current distributed file systems (DFS) employ cleverly designed balancing algorithms to facilitate data distribution [1]–[5]. However, DFS cannot avoid encountering skewed real-world workloads. The gradual load imbalance among nodes, especially hotspot data, progressively introduces the challenge of data migration. If DFS clumsily fails to trigger a data migration operation in time, it can cause dramatic performance degradation, e.g., low system throughput and significant tail latency. Even worse, overloaded nodes will directly stop writing data. Given the escalating demands for DFS in contemporary data-intensive applications, it is imperative to promptly address this challenge.

Typical file-level migration policies can usually be summarised into the following three categories: capacity greedy based methods, file life cycle based methods and dynamic file hotness based methods (*data hotness represents the access frequency*). First, the most common data rebalancing mechanism in DFS is to greedily migrate data to the most free node, especially Gluster [1] and Ceph [2], based on capacity once the node reaches a single threshold. Second, file life cycle based methods pick files based on the sequence of file creation time [5]. Third, the dynamic file hotness based approach develops a migration policy with multiple thresholds, which aims to keep the disk usage as close to the average usage as possible [4].

However, there are some limitations in file-level migration methods. First, fine-grained file-level redistribution generates a high performance tax because it comes with the additional cost of more pointers [6]. Second, file-level hotness disappears too quickly. The literature suggests that frequent migrations do not alleviate load imbalance and cause 'ping-pong' effects [7]. Instead, directory-level hotness is much more durable [8]. Third, the file-level migration method rarely considers directory locality. Finally, a single dimension (e.g., capacity or file creation time) is insufficient for identifying the data that contributes most to load imbalance.

In this paper, we present a Load-aware Directory Migration Policy (LoADM) efficiently to solve data migration problem. The first limitation is fixed efficiently and expeditiously by coarse-grained migration of hot directories to nodes sustaining less load. To solve the second limitation, LoADM enables directory hotness model to precisely guide balance. To solve the third limitation, LoADM proposes adaptive peer directory co-migration. The peer directories refer to the same node's subdirectories (with the same depth) under the same parent directory. Finally, LoADM solves the last limitation by using multidimensional information such as bandwidth and IOPS. Our contributions are as follows:

- **New perspective**.To the best of our knowledge, this is the first time to solve DFS data migration from the directory level. To leverage directory locality, we propose peer directory co-migration.
- **New method**. We design a load-aware directory migration policy (LoADM). We build a lightweight directory hotness model picking the most urgent directory and its adaptive peer directories, and use a Particle Swarm Optimization (PSO) based multidimensional migration model to better balance the workloads in DFS.
- **High performance**. We evaluate LoADM by conducting extensive experiments on practical DFS using real-world workload traces. LoADM improves IOPS and bandwidth by 67.4% and 69.2%, and reduces completion time by 41.6% compared to state-of-the-art methods.

II. Motivation

In this section, we make a deep analysis of the design insights about LoADM from the following three aspects about directory migration problem.

(1) Directory hotness and lifetime. For real workloads, the hotspots remain relatively stable over short periods of time,

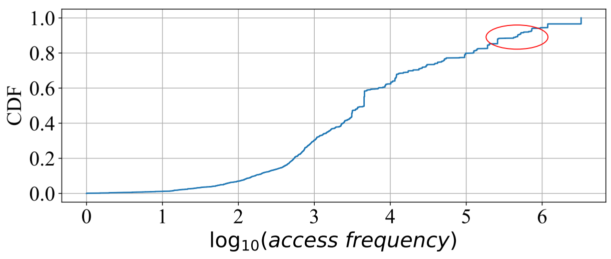


Fig. 1. The CDF of log_processed directory access frequency. Directory_level access have a long-tail characteristic, with few directories having significant access.

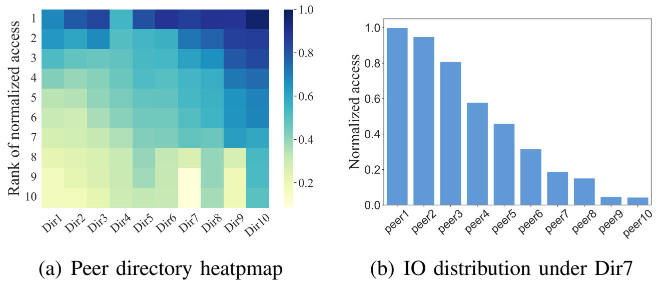


Fig. 2. (a) depicts an example of heatmap for peer directory. We randomly select 10 hot directories as the horizontal axis (select the next one if its subdirectory breadth is less than 10). The vertical axis shows the normalized access frequency rank of the subdirectories corresponding to these 10 directories, with darker colours indicating more frequent IO access. (b) describes the IO distribution of subdirectories under Dir7 in Figure (a).

even though they slowly diminish over time [9]. Generally, the hotness duration of a directory exceeds that of a single file in the directory [8]. It is not surprising, as many studies have reported good locality of the directories [10]–[13]. However, file-level migration strategies ignore this fact which fails to take advantage of directory locality. Moreover, files may have a shorter lifetime. Frequent file-level migrations are likely to be ineffective data moves triggering a 'ping-pong effect' [7]. In short, file granularity does not necessarily guarantee rebalancing [14]. Therefore, we address data migration problem in DFS from the directory level.

(2) Directory access distribution. Fig.1 and Fig.2 clearly demonstrate our observation based on the analysis of realworld traces. As shown in Fig.1, we statistically count the IO access frequency of all last-level directories and then perform a logarithmic process to plot CDF curves. **Observation1**: Directory-level access have a long-tail characteristic, with few directories having significant access. This inspires us to find those high-load directories by learned directory hotness model detailed in Section IV-C. By analysing access patterns, LoADM predicts potentially highly loaded directories and then put them to the migration queue. This policy is designed to meet future demand in advance to avoid performance bottlenecks.

(3) Adaptive peer directory co-migration. Previous research considers the peer directory impact when predicting the load [8]. However, it fails to explore the distinctive contributions of each peer directory. Namely, how many peer directories should be selected and whether an adaptive number should be set. In contrast, we studied the IO distribution between peer directories as shown in Fig.2(a) and Fig.2(b). We randomly select 10 hot directories as the horizontal axis, and select the next one if its subdirectory breadth is less than 10. These hot directories are predicted by the directory hotness model and have the same label (detailed in Section IV-C). And the vertical axis shows the normalized rank of each subdirectory, with darker colours indicating more frequent access in Fig.2(a). **Observation2**: There is a hotness correlation between peer directories: the peers of the hotter directories may also be hotter. **Observation3**: There is not an even distribution of hotness between peer directories. **Observation4**: In case of top frequently accessed peer directories, their hotness may be close to each other. We notice that the top k peer directory (having a same parent directory) access are close, but k is not fixed. For example, for the peer directories under Dir7, the top 3 access frequency is close, while for the peer directories under Dir3, the number is 2. We define k as the peer directory co-migration factor. Intuitively, k changes with workloads. Motivated by the aforementioned observations, we explore the distinctive contribution of peer directories and design peer directory co-migration detailed in Section IV-D.

III. Problem Statement

We break directory migration problem into two steps, that is, which directories to migrate and how to migrate them.

(1) Which to migrate? Intuitively, directories may contain a non-negligible number of files, which can be a challenge for directory-level migrations. The first key solution is to combine information from multiple dimensions (IOPS and bandwidth of the directory). Some dimensional information is insignificant. For example, NetApp reports that capacity imbalance does not necessarily cause system performance degradation [3]. Therefore, using multidimensional load information promotes thorough consideration. Moreover, it is important to note that we are concerned with migrating highly loaded directories where size is not significant. Therefore, scenarios containing many large files are out of the scope of our study.

The second challenge is how to identify the hotspot data. To solve this problem, we employ a lightweight classification model to divide the directories into four hotness levels (see Section IV-C for more details).

(2) How to migrate? After finding the hot directories and the corresponding co-migrating peer directories, we determine the location according to a multidimensional directory migration model based on PSO. To provide a more precise understanding, we give the following mathematical definitions. Define *Balance* as the load balance degree which can be calculated as follows:

$$Balance = \frac{v}{\sqrt{\frac{1}{m}\sum_{i=1}^{m}(\frac{R_c(i,r)}{R_t(i,r)} - v)^2}}, i \in [1, ..., m] \quad (1)$$

In Equation (1), i represents i_{th} node. m represents the max number of the nodes. v represents the mean of the resource utilization of each node. $R_c(i,r)$ and $R_t(i,r)$ represent the current and the limit resource value of i_{th} node where r $\in$[IOPS, Bandwidth]. In short, the migration model serves to maximise the balance without exceeding the upper limit of node resources.

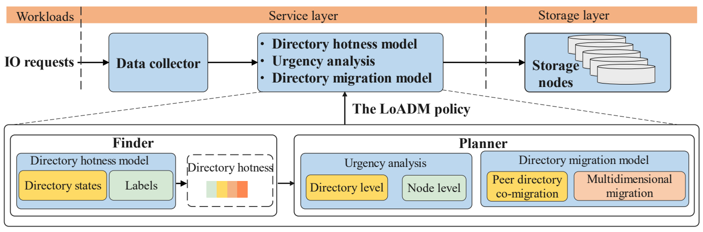


Fig. 3. The architecture of LoADM, which consists of three main modules, Data collector monitoring the load and processing data from storage nodes, Finder labeling directories and finding hot directories by learned model, Planner deciding which directories to migrate and to which nodes.

IV. Design

A. Overview of LoADM

The key idea of LoADM is to use a load-aware approach for data migration from the directory-level ensuring the best DFS balance possible. Figure 3 provides a brief overview, our method is designed in the service layer. LoADM is composed of three main modules, Data collector monitoring the load and processing data from storage nodes, Finder labeling directories and finding hot directories by learned model, and Planner deciding which directories to migrate and to which nodes.

B. Data collector

The data used for classification is sampled and processed from the data collector module. The directory request features includes directory depth, time_label, file_object, IO_Time_mean, parent directory breadth, timeStampDiff(the average difference between timeStamps of the same access), directory IOPS_mean and Bandwidth_mean.

C. Finder

This module can label directories and identify hot directories by learned models. If directory hotness is defined as different levels, it is essentially a classification problem. Then we can use a learning model to identify high-load directories in advance. Learning models can recognise data patterns and perform specific classification tasks by processing large amounts of historical data, which are widely available in storage systems [15]–[17]. For existing directories, their request information can be collected statistically by the data collector. However, their labels are unknown.

To solve this challenge, we propose a novel hotness score measure to guide learning models as shown in the Equation 2:

$$HotnessScore = \lambda \times F + \frac{1-\lambda}{\Delta T}, \lambda \in [0,1] \qquad (2)$$

where λ refers to a learned factor (we set the λ to 0.5) , F refers to access frequency to the directory, and ΔT refers to the mean value of the time intervals between every two access in the time window. The small ΔT means the directory is potentially a hot directory. We statistically count the upper and lower limits of the scores, and then divide the score intervals. For example, we divide the entire score range according to the ratio of 2.5:2.5:4:1, which corresponds to one label respectively. Then the group of labels is [*hotness_0*, *hotness_1*, *hotness_2*, *hotness_3*], and the data in the top 10% is identified as the most hot (*hotness_3*). This division is based on experience and analysis of real workloads, and we can also flexibly adjust the division ratio.

Such design in Equation 2 is derived from three aspects. First, it combines historical information and the current state. Second, it exploits the possible locality of the data in the time dimension, i.e. the closer the access time is, the hotter it is likely to be. Thirdly, it represents the trade off between access frequency and time. The first half of the Equation 2 reflects access frequency, while the other part reflects the mean access speed. Combining them together shows the trend.

Based on this novel hotness measure, we conduct a comprehensive analysis of the 9 most popular classification models. Note that we use SMOTE to balance the samples collected by the data collector. For simplicity, we use scikit-learn [18] to train the models. We must strive for a high recall and F1 score, as they represent the likelihood of accurately detecting distinct hotness. Furthermore, the F1 score serves as a valuable metric for assessing the equilibrium between two key measures, namely, precision and recall. And F1 score is defined as $F1\ score = \frac{2*(precision*recall)}{precision+recall}$, the higher, the better).

The top five best-performing models are GBDT, LightGBM, HistGBDT, RF and DT, and their F1 scores are 82.99%, 82.62%, 81.24%, 79.10% and 75.64%. It shows that there is little gap in the performance of the best three models, with values of F1 score above 80%. Compared with other models, GBDT achieves the best results: the highest recall (83.58%), F1 score (82.99%). However, the training cost of GBDT is also the highest. Naturally, the shorter the offline training time for storage systems, the better. Given that LigbtGBM satisfies both performance and overhead requirements, we have chosen it as the primary model. Moreover, we compute the hotness score periodically to prevent the problem of model aging caused by drastic data changes.

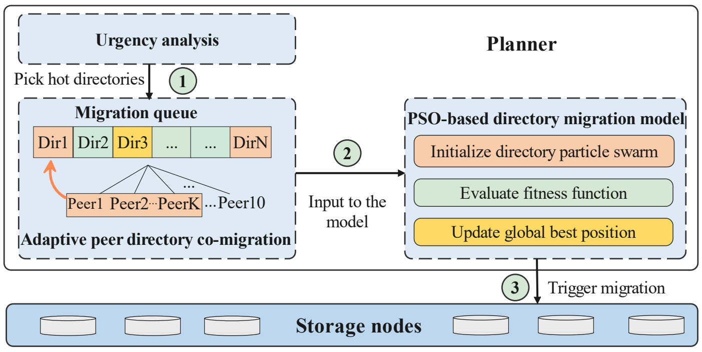

Urgency analysis
Planner
Pick hot directories
1
Migration queue
Dir1
Dir2
Dir3
...
...
DirN
Peer1
Peer2
PeerK
Peer10
Adaptive peer directory co-migration
2
Input to the model
PSO-based directory migration model
Initialize directory particle swarm
Evaluate fitness function
Update global best position
3
Trigger migration
Storage nodes

Fig. 4. The workflow of Planner. Based on the directory hotness model, we perform an urgent analysis to determine whether a migration action will be triggered. Then we put hot directories into migration queue. Finally, we migrate highly loaded directories to appropriate storage nodes through a PSO-based directory migration model.

D. Planner

The Planner is response to balance DFS by making migrating decisions which consists of two sub-module: urgency analysis module and directory migration module. Fig.4 depicts the workflow.

(1) Urgency analysis. Urgency analysis module reacts to the trend of imbalance in DFS by directory state and node state. First, based on the Finder's output we can get the directory's hotness level. Typically, we select the directories with the highest hotness as alternatives, indicating that they have the potential to influence the balance. The hotter the directory, the higher the urgency. Second, we analyse the state of the nodes to determine whether the distributed cluster has reached the imbalance cliff. On the one hand, the urgency analysis module will check whether the node's current state violates the load's dynamic threshold. On the other hand, the balance degree reflects significantly the multidimensional resource differences between nodes.

In addition to this, we have considered several priority rules. For example, first, priority is given to directories with small capacity, thus reducing migration overhead. Second, directories with high hotness level have also a high priority. The predicted hotness level affects urgency analysis significantly. A directory that contributed to a low predicted hotness is given a much lower priority. In particular, we focus on migrating highly loaded directories that are not large in size to minimise migration overhead. In general, these rules are designed to identify the directories that are most worthy of migration.

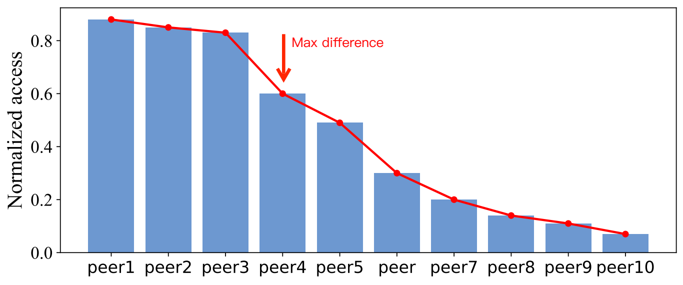


Fig. 5. An example about how to compute peer co-migration factor k. Peer directories under a same parent directory in this figure. The difference between every two points can be calculated to find the access frequency corresponding to the largest change. In this example, we set k = 3.

(2) Peer directory co-migration. As we described in the motivation, each peer directory contributes distinctively to the node load. We notice that the top k peer directory (having a same parent directory) accesses are close, but k is not fixed. Intuitively, k changes with workloads. Besides, there can be a Matthew effect even in the hottest directories, with the top few subdirectories' loads being dominant while the next ones decaying rapidly.

To better explore data locality, we propose a peer directory co-migration method. The peer directories refer to the same node's subdirectories (with the same depth) under the same parent directory. Co-migration is defined as k peer directories (k is an adaptive factor) are treated as a whole to be migrated to the appropriate node. The idea behind this design is to exploit data locality as much as possible in order to speed up IO access, especially sequential access.

However, does putting k hot directories together on the same node create another hotspot? To solve this problem, we use a simple but effective method. First, sort the migration queue based on load. Second, compute the average historical load of the migration queue. Third, if the top ones exceed α multiples of the average, the peer factor for those ones is set to 0, implying no peer co-migration for them is launched. We have empirically set the parameter α to 2. For example, for a migration queue [*Dir1,Dir2,Dir3,Dir4,Dir5,Dir6*] sorted by load, the load of *Dir2* is more than 2 times the average load, then *Dir2* doesn't start peer co-migration. In this way, we avoid the problem of creating additional hotspot. Besides, it should be noted that not every item in the queue will have hot peer directories by analysing real-world workloads.

Algorithm 1 Adaptive Peer Directory Co-migration Algorithm

Input: Peer directory set **P**, Parameter k, Parameter $index$
Output: k

```
1:  $k \leftarrow 0$
2:  $PeerLoad \leftarrow DataCollector(P)$
3:  $DescendingSort(PeerLoad[index])$
4:  for each $index$ do
5:      $\Delta PeerLoad \leftarrow PeerLoad[index+1] - PeerLoad[index]$
6:      if $max \leftarrow \Delta PeerLoad[index]$ then
7:          $k \leftarrow index$
8:          $break$
9:      end if
10: end for
11: $k \leftarrow CheckTimeCorrelation(PeerLoad[0, 1, ...k])$
12: return $k$
```

Fig.5 illustrates an example of how to compute the peer co-migration factor k. And the pseudo-code of the adaptive peer directory co-migration is shown in Algorithm 1. First, for the alternative hot directory to be migrated, we count the performance information of its peer directories (Line 2). Second, sort the peer directory load in descending order (Line 3). Third, calculate the fitted curve of load and index and find the point with the largest slope, i.e., the inflection point. In other words, the difference between every two points can be calculated to find the index corresponding to the first largest change (Line 10 - Line 9). This step explores the correlation of the peer directories' load. And then check the temporal correlation of these k peer directories (Line 11). Specifically, the temporal correlation is to check if all these k peer directories have appeared in the same time window (5 minutes). If anyone doesn't appear, delete it. Following the above analysis of load and temporal correlation, we obtain the peer directories for adaptive co-migration (Line 12). And experiments in Section V-B demonstrate our results.

(4) PSO-based multidimensional migration algorithm. We construct an online, load-aware, peer directory-collaborative migration algorithm that performs node selection. Briefly, the main idea is to first select the right directories to put into the migration queue and then using PSO-based model to enable the maximum fitness in Equation 3. The pseudo-code is shown in Algorithm 2.

In general, the solution for dealing with multidimensional resources for the constraint is dimension reduction. This process transforms the multidimensional resources into a normalized dimension using a fitness function. Subsequently, the resources are allocated to logical nodes based on an allocation policy that ensures the maximum target score defined. Our fitness function is defined as follows:

$$Fitness = \frac{1}{2}(Balance_{IOPS} + Balance_{BW}) + \sigma(Node_{hotnessScore}) \tag{3}$$

where $Balance_{IOPS}$, and $Balance_{BW}$ denote the balance degree of the IOPS and bandwidth, and $\sigma(Node_{hotnessScore})$ denotes the standard deviation of the hotness scores of all nodes respectively (*hotnessScore* short for HS in Algorithm 2). We consider not only the load balancing degree but also the node hotness score in the fitness function design. The reason for considering the hotness score calculated by Equation 2 is that it combines the effects of frequency and time and shows future trends.

Why use PSO? The basic idea of the PSO method is to simulate the information exchange and cooperative behavior among particles in a bird flock, guiding individuals in the search space towards better solutions. Note that many heuristics (e.g., genetic algorithms) can also be used to solve this problem. However, PSO algorithms usually have a fast convergence rate, which is important for latency-sensitive applications.

Algorithm 2 Multidimensional PSO_based Migration

```
Input: Directory list DR, Directory Information DI
Output: $BalanceDegree$
/*PSO-based multidimensional migration function*/
function PSO
    $InitializeDirParticleSwarm()$
    $Evaluate()$
    $Node \leftarrow UpdateGlobalPosition()$
end function
/*Main function starts */
$HotDirectories \leftarrow Finder(DR, DI)$
$MigrationQueue \leftarrow Urgency(HotDirectories)$
for each $item$ in $MigrationQueue$ do
    $peerInfo \leftarrow GET\_PEER\_Directory(item)$
    $IOPS, BW, HS \leftarrow Compute(item, peerInfo, DI)$
end for
$Load \leftarrow UpdateLoad(IOPS, BW, HS)$
$PSO(MigrationQueue, Load)$    ▷ Construct the policy that maximises the PSO model fitness function to find the recommended migration nodes.
$BalanceDegree \leftarrow PerformMigration()$
return $BalanceDegree$
```

V. Evaluation

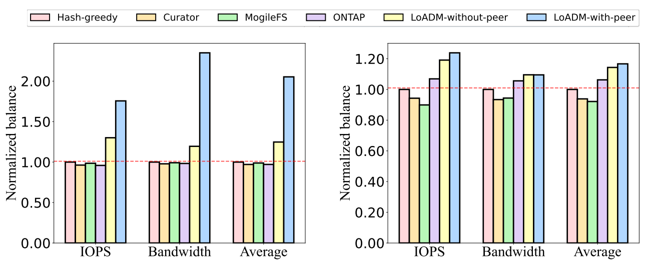
A. Experiment setup

To evaluate the proposed method, we conducted experiments on two widely used real-world traces (**MSN Storage File Server** and **Livemap**) [1] containing 29,345,013 and 44,755,552 traces, respectively. We use scikit-learn [18] to split the dataset into 60% training data and 40% for model testing. We built GlusterFS with code version 8.2. The system consists of 4 server nodes and 1 client node. Each server node has 64G RAM and 2TB HDD storage capacity.

Next, we briefly describe the advanced baselines that have been developed to address data migration problem in DFS. *Hash_Greedy* [1], [2] is greedily choosing the node with the most free capacity to migrate data into; *ONTAP* [3] determines the data distribution based on the file size and can be customised with a similar greedy migration policy; *MogileFS* [5] selects files for migration based on file creation time; *Curator* [4] employs an advanced dynamic data migration scheme based on file popularity.

B. Balance degree

Balance degree serves as crucial metrics for evaluation, as discussed in Section III. The higher the balance the better, which shows how well the system is balanced in each dimension. As shown in Fig.6, our LoADM achieves the best performance. Specifically, as for MSN traces our approach LoADM (with peer directory co-migration) improves 1.8×, 2.3×, 2.1× in IOPS and bandwidth, and average compared to *Hash_greedy*. As for Livemap, LoADM (with peer directory co-migration) improves 1.2×, 1.1× and 1.2× in IOPS and bandwidth, and average compared to *Hash_greedy*. Another clue is that our method without peer co-migration yields a comparable improvement. This illustrates that migration from the directory level is a promising option.



(a) MSN (b) Livemap

Fig. 6. (a) and (b) depict the balance degree of different migration methods. LoADM with peer co-migration performs the best.

Hash_greedy greedily selects data for migration based on capacity, ignoring the impact of load. As for *MogileFS*, files created early are not necessarily high load. *Curator* intelligently makes high disk utilisation as close to the average as possible, but it tends to select cold data. On the one hand, LoADM can effectively identify the hotspot directories through the learning model, so as to filter out the most worthy alternatives for migration. On the other hand, the PSO-based migration model can comprehensively consider multi-dimensional information. In addition, the mechanism of peer directory collaborative migration also substantially exploits the data locality.

C. Performance in DFS

In this section, we implemented LoADM in GlusterFS. We use SAR [19] to collect performance statistics. Fig.7, Fig.8 and

[1]Dataset: http://iotta.snia.org/traces/158.

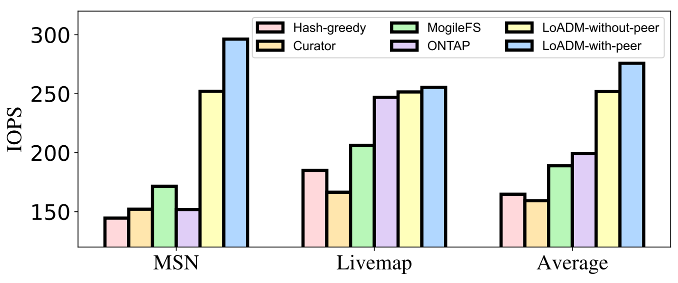


Fig. 7. IOPS of different migration methods.

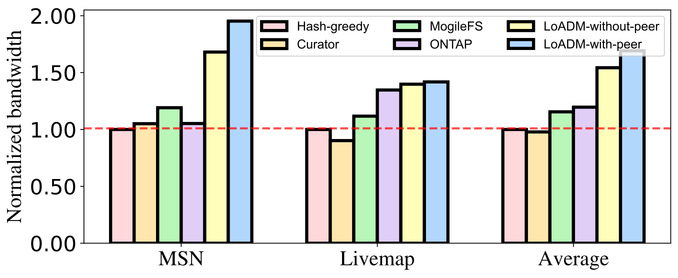


Fig. 8. Bandwidth of different migration methods.

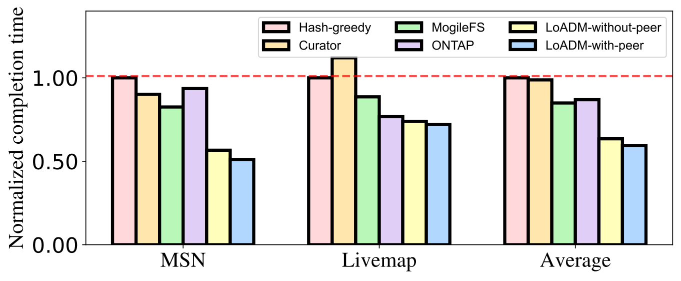


Fig. 9. Completion time of different migration methods.

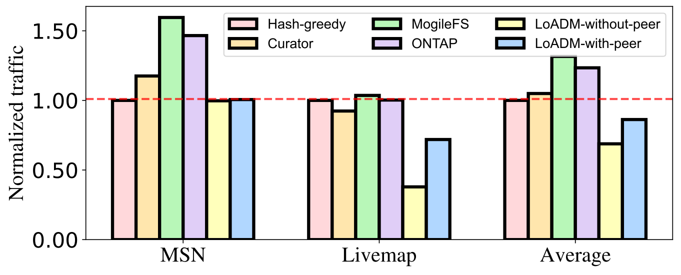


Fig. 10. Migration traffic of different methods.

Fig.9 demonstrate the results. Our scheme outperforms the file-level migration schemes in all performance metrics. On average, LoADM yields improvement in IOPS (67.4%), bandwidth (69.2%) and gets the lowest completion time (reduces 41.6%), demonstrating the advantage of the directory migration method.

D. Overhead

Fig.10 shows the amount of migration data. It can be seen that our approach migrates the least amount of data. All results are normalized to the *Hash_greedy* method. On average, the overhead of our method with peer co-migration is reduced by 14.7% compared to the baseline (*Hash_greedy*). The maximum reduction is 35.5% compared to *MogileFS*. On both real-world traces, our approach has the lowest overhead. This is due to the fact that LoADM mainly tends to migrate highly loaded directories with small capacities, which is ignored by other methods. To summarize, our proposed method not only has the best performance but also has the least overhead.

VI. Conclusion

This paper proposes a load-aware directory migration policy for achieving DFS load balance. We build a directory hotness model to identify hot directories. Thus, it will trigger the migration if the urgency analysis is satisfied. In addition, we propose peer directory co-migration to exploit directory locality. Finally, a directory multidimensional information migration model is designed. The experimental results show our proposal LoADM not only smooths workload balance but also substantially improves practical DFS performance, such as IOPS and completion time by 67.4% and 41.6% in contrast to state-of-the-art methods.

Acknowledgment

This work was supported by the National Key Research and Development Program of China (Grant No.2023YFB4502701) and National Natural Science Foundation of China (Grant No.62232007,No.61821003).

References

[1] Red Hat, "Gluster," 2019. [Online]. Available: https://www.gluster.org/
[2] Sage A. Weil et al., "Ceph: A scalable, high-performance distributed file system," in *OSDI'06*, 2006, pp. 307–320.
[3] NetApp, "Netapp ontap flexgroup volumes best practices and implementation guide," 2021. [Online]. Available: https://www.netapp.com/pdf.html?item=/media/12385-tr4571.pdf
[4] Ignacio Cano et al., "Curator: Self-Managing storage for enterprise clusters," in *NSDI'17*, 2017, pp. 51–66.
[5] Danga Interactive, "Mogilefs." [Online]. Available: https://github.com/mogilefs/mogilefs-docs
[6] Ram Kesavan et al., "Flexgroup volumes: A distributed WAFL file system," in *ATC'19*, 2019, pp. 135–148.
[7] Yiduo Wang et al., "Lunule: an agile and judicious metadata load balancer for cephfs," in *SC'21*, 2021, pp. 47:1–47:16.
[8] Yuanzhang Wang et al., "Ldpp: A learned directory placement policy in distributed file systems," in *ICPP'22*, 2022, pp. 1–11.
[9] Ziyue Qiu et al., "Frozenhot cache: Rethinking cache management for modern hardware," in *EuroSys'23*, 2023, p. 557–573.
[10] Siyang Li et al., "Locofs: A loosely-coupled metadata service for distributed file systems," in *SC'17*, 2017, pp. 1–12.
[11] Annamalai Muthukaruppan et al., "Sharding the shards: managing datastore locality at scale with akkio," in *OSDI'18*, 2018, pp. 445–460.
[12] Yang Zhan et al., "Efficient directory mutations in a full-path-indexed file system," *TOS*, vol. 14, no. 3, pp. 1–27, 2018.
[13] Wenhao Lv et al., "Infinifs: An efficient metadata service for Large-Scale distributed filesystems," in *FAST'22*, 2022, pp. 313–328.
[14] Peter Macko et al., "Survey of distributed file system design choices," *TOS*, vol. 18, no. 1, pp. 1–34, 2022.
[15] Giulio Zhou et al., "Learning on distributed traces for data center storage systems," *MLSys'21*, pp. 533–549, 2021.
[16] Mohammed Bakr Sikal et al., "Thermal- and cache-aware resource management based on ml- driven cache contention prediction," in *DATE'22*, 2022, pp. 1384–1388.
[17] Ke Liu et al., "A lightweight and adaptive cache allocation scheme for content delivery networks," in *DATE'23*, 2023, pp. 1–6.
[18] Scikit-learn, "scikit-learn," 2019. [Online]. Available: https://scikit-learn.org/stable/
[19] Sysstat, "sysstat-system performance tools for the linux operating system." [Online]. Available: https://github.com/sysstat/sysstat