A Distributed Cache Framework for Metadata Service of Distributed File Systems

Yao Sun[†‡], Jie Liu[‡], Dan Ye[‡], Hua Zhong[‡]
[†]University of Chinese Academy of Sciences, Beijing, China
[‡]Institute of Software, Chinese Academy of Sciences, Beijing, China
{*sunyao10, ljie, yedan, zhongh*}*@otcaix.iscas.ac.cn*

***Abstract*—Most recent distributed file systems have adopted architecture with an independent metadata server cluster. However, potential multiple hotspots and *flash crowds* access patterns often cause a metadata service that violates performance Service Level Objectives. To maximize the throughput of the metadata service, an adaptive request load balancing framework is critical. We present a distributed cache framework above the distributed metadata management schemes to manage hotspots rather than managing all metadata to achieve request load balancing. This benefits the metadata hierarchical locality and the system scalability. Compared with data, metadata has its own distinct characteristics, such as small size and large quantity. The cost of useless metadata prefetching is much less than data prefetching. In light of this, we devise a time period-based prefetching strategy and a perfecting-based adaptive replacement cache algorithm to improve the performance of the distributed caching layer to adapt constantly changing workloads. Finally, we evaluate our approach with a hadoop distributed file system cluster.**

***Keywords*-metadata server cluster; distributed file system; load balancing; caching; prefetching.**

I. Introduction

File system metadata describes, among others, the file path, directory structure and file attributes of a file system. Most distributed file systems have an independent cluster to manage metadata for scalability. Most companies adopt this type of distributed file system for their cloud applications, such as the Google File system [3], Facebook HDFS [19], and Alibaba TFS [2]. There are some important characteristics of these cloud applications, including a petabyte-scale storage cluster [11], a large number of concurrent requests [5, 17], and a high proportion of small files (such as picture storage services in Facebook [21]). The first step of most file operations is to get the file metadata, such as file location information. These characteristics bring challenges to load balancing management of the metadata server (MDS) cluster.

In this paper, we target the problem of MDS cluster frequent load skews (FLS) with continuous changing workloads. This is mainly caused by uneven requests to the MDS cluster, for example a *flash crowds* scenario [18]. When the number of concurrent requests exceeds the upper limit of the MDS processing capability, the MDS suffers performance decline, with average response time increasing and the overall throughput decreasing. The cluster then becomes imbalanced. The more load skew that occurs with the MDSs, the lower is the throughput of the system. Especially, when one request needs multiple MDSs to collaborate accomplishments, the response time of the request depends on the slowest MDS. Therefore, maintaining an MDS cluster at a balanced rate is key to avoiding a system performance bottleneck. Compared with a data server (DS) cluster, the MDS cluster has three different characteristics. First, the size of the MDS cluster is small but the number of requests arriving at each MDS is large [22, 23]. A deployed distributed file system may consist of tens of MDSs and thousands of DSs. Second, the response time of the metadata request should be as little as possible [5]. Last but most important, potential multiple types of hotspots may occur at any time, including sudden hotspots, cyclical hotspots and permanent hotspots. These characteristics cause MDS clusters to suffer load skews far more frequently than DS clusters.

It is difficult to use existing distributed metadata management schemes to solve FLS problem. For example, subtree partitioning often leads to uneven request distributions among MDSs. The system has to perform metadata migration operations to avoid load skews, while metadata migration often brings negative effects to the metadata hierarchical locality. Hash-based mapping can evenly distribute storage loads to each MDS, but hotspots consisting of a single file or directory still lead to MDS cluster-based load skews. Some distributed file systems use a cache to solve the problem of request load imbalance. Although caching is an effective way to handle the uneven request distribution, when tens of thousands of clients simultaneously access the same directory or file [5, 6, 17], the cache cannot perform well when it is placed in either the client or the MDS. Therefore, we establish a separate distributed caching layer to specifically manage hotspots. In order to ensure system scalability, we organize and manage cache with a distributed hash table (DHT). A DHT can elastically control the size of the cache and the number of cache nodes. The distributed caching layer can be deployed in either a separate cluster or the idle nodes belong to the MDS cluster.

In the FLS mode, the distribution of client requests is unpredictable. Our goal is to balance the request load across the MDS cluster using a distributed caching layer. However, how to accurately and quickly identify the potential load skew across MDSs and hotspots is a challenge. A cache replacement algorithm can identify hotspots with temporal and spatial locality, frequently accessed metadata, and access patterns, but it cannot identify the potential load skew the MDS cluster. We therefore use prefetching as a technique to identify a potential

1521-9097/13 $31.00 © 2013 IEEE
DOI 10.1109/ICPADS.2013.20

load skew across MDSs and then cache hotspots belonging to these MDSs, redistributing the overload. Typically, one first monitors and then calculates the monitoring results to identify load skew nodes. When load skew frequently happens, it is difficult to set the size of the monitoring smoothing window. If the smoothing window is too long, it is difficult to capture hotspots; if the smoothing window is too short, the adjustment cost is too high. In the FLS mode, there are typically multiple peaks in a smoothing window, so it is meaningful to identify MDSs with potential load skew and prefetch hotspots while monitoring. Based on this, we designed a time period-based prefetching strategy (TPPS) to quickly identify the MDSs with potential load skew. Some prefetching may be wasteful, but fortunately the cost of useless metadata prefetching is much less than data prefetching so we can balance the "accuracy" and "quickness" to approximate optimal caching performance.

The contributions of our approaches are

- Establish a request load balancing framework for an MDS cluster that can adaptively manage potential multiple hotspots under an FLS mode;
- According to metadata characteristics and the FLS mode, design and implement a TPPS and a perfecting-based adaptive replacement cache algorithm (PARC) that integrates the metadata prefetching with caching.

Systems suitable for this approach include

- HDFS [1] and other distributed file systems that adopt architecture with separated metadata servers and data servers [2, 3, 10].

II. Background and Related Work

A. Background

Roselli et al. [15] showed that metadata operations occupy more than 50% of all file system operations. Take HDFS as an example, an MDS cluster load includes two parts. One part consists of client requests, such as mkdir, rmdir, and rename. The other part consists of DS cluster reports and heartbeats. Since a system implements fault tolerance functions by a mechanism of multiple replicas, data and DS increase or decrease will cause data to move. MDSs perform a master role receiving reports and heartbeats from the DS cluster, and then MDSs need to be re-calculated and re-distributed for the changing DS cluster. It then generates a series of metadata operations. Shvachko et al. [16] showed that metadata operations generated by reports and heartbeats account for about 30% of the total number of metadata operations. This part of the workload is generated in a regular pattern, while the other workload is generated by the clients in unpredictable patterns, especially in a service with a large number of applications and users. An MDS cluster load skew is mainly caused by a client uneven request load distribution.

B. Related Work

Dynamic Load Balancing Prior work has used the idea of dynamic adjusting to address the problem of dynamic load balancing. For example, Quanqing Xu et al. [11] presented DROP to meet it. They used a 0-1 knapsack mode to find the heavy load MDSs and light load MDSs, then perform a remapping from heavy MDSs to light MDSs for MDS cluster load balancing. In this procedure, the system can preserve metadata locality. It belongs to storage load balancing. They also used caching for unevenly request load distributions. Bin Fan et al. [4] balanced request loads across a storage server cluster, by using a small front cache to manage a small number of hotspots, which led to *flash crowds*. In this work, they did a series of detailed experiments for verifying the relation between the front-end cache size and the number of back-end nodes in an adversarial access pattern. However, this does not take into account the metadata characteristics. Another scheme is Bloom Filter Arrays [23], which is a two-level decentralized scheme for distributed metadata management; it uses a global replication strategy for recently visited metadata. This is a solution for adjusting the accuracy for improving performance. These works cover three main ways for system dynamic load balancing, including remapping [7 , 11], caching [3, 4, 5], and multicopying [3, 23].

The framework of our approach also uses caching to balance request distributions, but the difference is that we have established a fully independent distributed caching layer for hotspots. We first use prefeching to improve metadata caching performance. We can deploy caching in different locations for different systems, we can also flexibly use a distributed caching layer to meet different service level objectives (SLOs), and an independent caching layer is more suitable for FLS.

Integrated Prefetching and Caching By learning from metadata access characteristics, we present the approach that integrates prefetching and caching to address FLS problems. The original motivation for prefetching is to reduce the latency of I/Os [9]. In this paper, the purpose of prefetching is to improve caching performance. The idea of integrated prefetching and caching for performance optimization has also been extensively studied, for example [13, 14, 22]. Butt et al. [13] analyzed the impact of prefetching on the performance of many well-known replacement algorithms. They showed that many detailed simulations that show prefetching can effectively improve caching performance. We were inspired by this work in our choice of an adaptive replacement cache (ARC)[12] as our basic caching replacement policy. Other investigators [14, 22] also agree that prefetching can improve the efficiency of caching but were aware that prefetching objects into a cache can pollute the cache and is particular harmful in the case of prefetching error. They established conservative prefetching rules for the integration between prefetching and caching, which avoids unnecessary performance degradation.

Compared to these work, our approach is aggressive for the integration between prefetching and caching. This is benefited from the fact that metadata prefetching error costs far less than data, and the multi-queue mechanism of ARC also provides an opportunity for our proactive prefetching strategy.

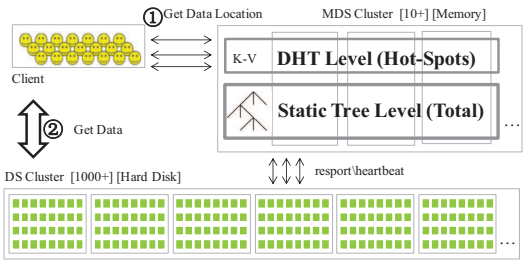

Figure 1. Two levels of distributed metadata management schemes evolved from HDFS.

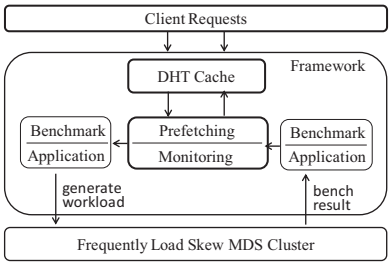

Figure 2. The framework of prefetching and caching in each MDS.

III. System Architecture

A. General Description

Distributed Cache Management We use a replication strategy to organize distributed caching layer rather than partitioned strategy. There are two reasons: first, we do not want distributed operations across multi-caches; second, the hotspots are of relatively small size. In fact, we can also use the partitioned strategy to manage the distributed caching layer. However, the requested metadata might be located in multiple nodes, while remote operations involve network transfer and data serialization/deserialization. This factor needs to be considered.

Cache Consistency We use a distributed caching layer to solve load balancing problems. The biggest challenge is consistency. A considerable part of cloud storage applications allows a weak consistency logic. Take an e-commerce website picture storage service as an example. The seller uploads product pictures, but buyers do not need to get the pictures in real time. Time delays of a few minutes or even longer can be tolerated. Moreover, our approach mainly deals with weak consistency requirements of read workloads, while strong consistency requirements of read workloads and mixing workloads is beyond the scope of this paper. Different applications and workloads have different levels of conformance requirements, we plan to study this in the future.

B. System Architecture

Before presenting our technique, we first introduce our system architecture, which evolves from HDFS, as shown in Figure 1. It has three parts: client, namenode(MDS) and datanode(DS). The client can be a Web application, a MapReduce program, or a NoSQL database. In a typical deployment, datanode cluster consists of hundreds of commodity computers, which are responsible for storing data blocks on a disk, and each block has multiple replicas. Namenode cluster is responsible for storing the file system namespace tree and the mapping of file blocks to data nodes in memory. Namenode cluster typically consists of tens of computers.

When a client accesses a file, it first sends a request to the namenode for the metadata by remote procedure call. It then contacts the datanode directly and requests the transfer of the desired blocks. We use a distributed caching layer to manage hotspots, as shown in Figure 1. This can improve the effectiveness of the metadata cluster and make it more balanced. This DHT level can be deployed in any position, for example a separate cluster. In fact, whatever the distributed metadata management scheme, we need another layer to further ensure load balancing. This two-level structure combines the advantages of a distributed metadata management scheme and a distributed caching layer. To guarantee this structure is effective for FLS, the key point is to control the distributed caching layer toward user-changing workloads in order to dynamically adjust the perfecting strategy and the cache replacement algorithm to meet SLOs.

C. Metadata Server Structure

We then analyzed the structure of our load balancing framework at each MDS. As shown in Figure 2, the framework consists of a benchmark, a monitoring module, and a prefetching strategy module. The distributed cache layer is deployed in the metadata cluster, so each MDS also contains a caching module. A benchmark or real application generates a series of workloads for the FLS scenario. Based on the monitoring information and the benchmark results, the prefetching strategy module calculates potential load skew MDSs. The cache layer prefetching performs periodic replacement for the client requests.

IV. Approach Analysis

A. Model Description

In this section, we first describe the FLS model and then introduce TPPS and PARC. Our overall goal is to reduce the load skew of MDSs and to keep an even request distribution across MDS clusters.

TABLE I
SYMBOLS USED IN THE APPROACH ANALYSIS.

Symbol	Meaning
n	The number of MDSs
m	The number of load skew MDSs
p_i	Request fraction arriving i th MDS
S	Load skews distribution
t	A time unit for p_i
T	A load skew adjustmenting period,equal $n * t$

Model As shown in Table 1, the number of MDS cluster nodes is n. A set of workloads loading requests to each node with different probabilities will cover these n MDSs. Then m

($m < n$) MDSs will cause a load skew during a period. The load skew can be described as a distribution

$$S = (p_1, p_2, ..., p_n) \tag{1}$$

In which p_i means the fraction of requests arriving the i th MDS, $0 \leq p_i \leq 1$ and $p_1 + p_2 + + p_n = 1$. Without loss to the generality, we list the S in monotonically decreasing order in an unit time, that is,

$$p_1 > p_2 > ... > p_n \tag{2}$$

Various workload types exist in an actual system, such as uniform, latest, and hotspot. To address this problem, we assume that the MDS cluster only serves the hotspot workload, which is the main factor for load skew. In the cloud storage service, hotspots consist of three types: permanent, cyclical, and sudden. Each type may lead to load skew in the MDS cluster, but the most frequent troublemaker is the sudden hotspot. In mathematical sciences, "sudden" means unpredictability, known as randomness. Different from randomness, sudden has certain period properties. In our model, a time period T is divided into n consecutive units, described as

$$T = t_1 + t_2 + ... + t_n \tag{3}$$

where, T denotes a load skew adjustment period for S in Eq.(1), t_i is the time unit for p_i, and p_i expresses *flash crowds* randomly. There are m load skew MDSs among S in T, namely the frequent load skews.

Based on Eq.(3), we utilize p_i in the time unit t_i to estimating MDS load, instead of p_i in the whole adjustment period T. In other words, we use load changing trend instead of the mean value to estimate MDS load. Such a change can help us identify load skews caused by sudden hotspots and reduce performance jitters caused by FLS.

B. Time Period-based Prefetching Strategy

In the FLS scenario, the challenge is how to quickly locate overloaded MDSs. We are therefore mainly concerned with where and when prefetching occurs. What is being prefetched is considered to be easily achievable, we can use an identifier to get it. "Accuracy" and "Quickness" are two important features for a prefetching strategy, but both are difficult in the FLS scenario. For "Accuracy", the goal is to identify the most potential load skew MDSs and hotspots, which will become load skews in the future. For "Quickness", the goal is to minimize identify time. Based on Eq.(1), Eq.(2), and Eq.(3), we can utilize the well-known *secretary problem* model [8] to locate prefetching MDSs for the FLS scenario.

TPPS derives from the classical *secretary problem* model in which the employer needs to interview n candidates over time, one by one for the best one. Essentially, as shown in algorithm1, this is a model for frequent transforming degrees of the different candidates to winning. algorithm1 is therefore suitable is suitable for the "Accuracy" goal. According to the FLS model, we need to get m load skew MDSs in T. In the fifth line of algorithm1, if p_i has the bestscore, we will select it for prefetching because p_i is a random variable between 0% and 100%. Because the previous p_1 to p_{i-1} is the same as p_i, each has an equivalent opportunity to be the bestscore, so the probability of p_i is $1/i$ to be the bestscore. According to the indicator random variable theory,

$$E[p_i] = \frac{1}{i}$$

Further, according to Eq.(1),

$$E[S] = E\left[\sum_{i=1}^{n} p_i\right] = \sum_{i=1}^{n} E[p_i] = \ln n + O(1) \tag{4}$$

So, in FLS model, it will occur $m = O(\ln n)$ times load skew in T. Compared to this case, if S is in a monotonically decreasing order in T, $m = 1$, this is the worst situation for FLS; if S is in a monotonically increasing order in T, $m = n$, this is the best situation for FLS.

Algorithm 1 Identify load skew MDSs (n)

```
1: best ← 0
2: for p_i ← p_1 to p_n do
3:    compare p_{i-1} and p_i
4:    if p_i is better than best then
5:       best ← i
6:       select p_i
7:    end if
8: end for
```

Algorithm 2 Online identify potential load skew MDSs (k, n)

```
1: bestscore(current heaviest load node) ← −∞
2: for i ← 1 to k do
3:    if p_i > bestscore and bestscore > warning_level then
4:       bestscore ← p_i
5:    end if
6:    for i ← k+1 to n do
7:       if p_i > bestscore then
8:          return p_i
9:       end if
10:   end for
11:   return p_n
12: end for
```

To achieve the "Quickness" goal, we optimize algorithm 1 to algorithm 2. In contrast to algorithm 1, we compare the front K MDSs and select the one with the best score. We then continue to search for a better one within the remaining MDSs; the first winner will be considered. So, the key insight of algorithm 2 is how to determine the value of variable k; combining probability analysis theory, we obtain the optimal k is n/e.

Proof: To successfully locate potential load skew MDS (p_i, in the seventh line of algorithm2), we need to meet two conditions. The first is that the algorithm must choose p_i at position i, and the second one is that the algorithm cannot choose p_i from $k+1$ to $i-1$. This means that p_i has the

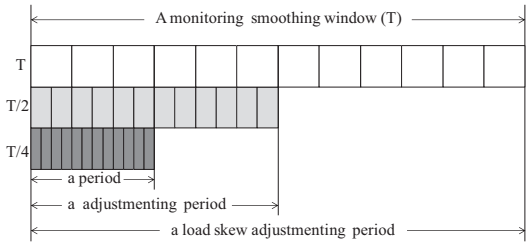

Figure 3. In a monitoring smoothing window, we adopt multiple different lengths of adjustment periods at the same time to identify the FLS.

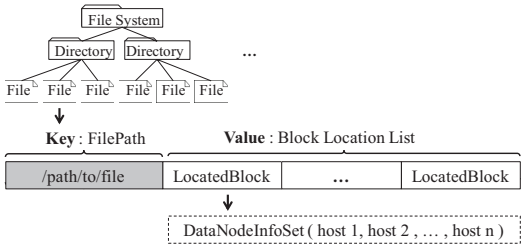

Figure 4. K-V data structure instead of subtree, each file consist of multiple blocks, and each block is replicated at several datanodes.

best score among p_1 to p_i, and best score only present in former k, which is among p_1 to p_{i-1}. We use S to denote the successful event for selecting p_i, S_i denotes the successful event for selecting p_i at position i, B_i denotes the event for p_i must be at position i, O_i denotes the event for selecting null among p_{k+1} to p_{i-1}, B_i and O_i are independent of each other, B_i depends on whether p_i is greater than the score of all others, and O_i only depends on the relative order of the score among p_1 to p_{i-1}. Accordingly,

$$Pr\{S_i\} = Pr\{B_i \cap O_i\} = Pr\{B_i\}Pr\{O_i\}$$

So, the probability of $Pr\{B_i\}$ is $1/n$, because bestscore may be at any one of n positions. In terms of O_i, if O_i occurs, then the bestscore must be present in former k, among p_1 to p_{i-1}, and bestscore may be at any one of the former $i-1$ positions. Then, $Pr\{O_i\} = k/i-1$, $Pr\{S_i\} = k/(n(i-1))$. Besides, if the bestscore present in former k positions among n nodes, we will not succeed. Accordingly,

$$Pr\{S\} = \sum_{i=k+1}^{n} \{S_i\} = \sum_{i=k+1}^{n} \{\frac{k}{n(i-1)}\} = \frac{k}{n}\sum_{i=k}^{n-1}\frac{1}{i}$$

We use an integral to approximate constraint the upper and lower bounds for the sum,

$$\int_k^n \frac{1}{x}dx \le \sum_{i=k}^{n-1}\frac{1}{i} \le \int_{k-1}^{n-1}\frac{1}{x}dx$$

Solving the definite integral,

$$\frac{k}{n}(\ln n - \ln k) \le Pr\{S\} \le \frac{k}{n}(\ln(n-1) - \ln(k-1))$$

Because we want to maximize the successful rate for selecting the potential load skew MDS and we are concerned about how to select the value of k, the goal is to maximize the lower bounds for $Pr\{S\}$. Derivative expressions $k(\ln n - \ln k)/n$,

$$\frac{1}{n}(\ln n - \ln k - 1)$$

Let this derivative be equal to 0, we can then obtain

$$k = \frac{n}{e} \tag{5}$$

So, in the FLS problem, we can use time $n * t/e$ to get the potential load skew MDS at the probability of $1/e$.

According to Eq.(4), we conclude the following. To enhance the probability of selecting a potential load skew node, we need to perform the algorithm approximately $\ln n$ times during the different lengths of the load skew adjustment period. As shown in Figure 3, we use a diminishing period to complete it at the same time point. We verify this in the performance evaluation section.

C. Perfecting-based Adaptive Replacement Cache Algorithm

We describe the PARC for caching replacement. PARC brings the prefetching content into the caching layer. Based on prefetching, PARC can exploit the blocks that caused the FLS. Meanwhile PARC can also prevent useless prefetching from wasting the cache queue.

ARC There are two LRU queues (denoted as Q1 and Q2) in ARC. the blocks in Q1 are accessed more than once and the blocks in Q2 are accessed only once. Q1 is true caching for applications, and Q2 is a ghost caching for potential hotspots. When requests hit the blocks in Q2, replacement will be triggered, moving the more popular blocks to Q1. Ghost cache only stores block identifiers for recording the accessed history of blocks. Cache size is 8, and ghost cache size is 2.

PARC The key point of our algorithm is using a prefetching content to fill Q2, so PARC can avoid cache pollution and take full advantage of the prefetching strategy. It is noteworthy that PARC is a scalable solution, for the size limit on Q1 is a parameter that varies according to I/O workload patterns. Our ultimate goal is to integrate prefetcing with caching to improve its performance, so we evaluate the effectiveness of PARC, not based on the hit rate but based on the average response time [20].

D. Summary

TPPS improves caching performance by initializing and cyclically adjusting for the cache; PARC provides a greater tolerance for the prefetching strategy accuracy using ARC mechanism of two queues.

V. Implementation and Performance Evaluation

We first present some details of the framework implementation and then present a detailed performance evaluation. We compare the performance on throughput and response time between TPPS and PARC.

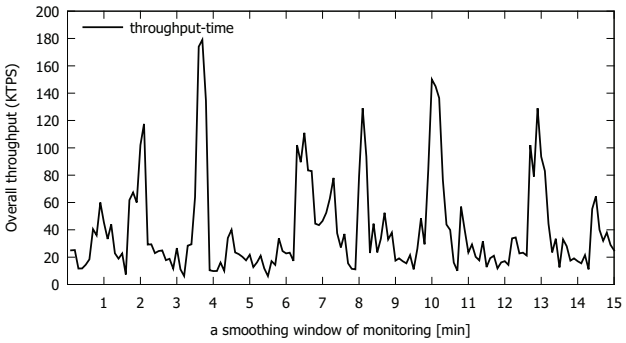

Figure 5. Arriving request distribution in a monitoring smoothing window.

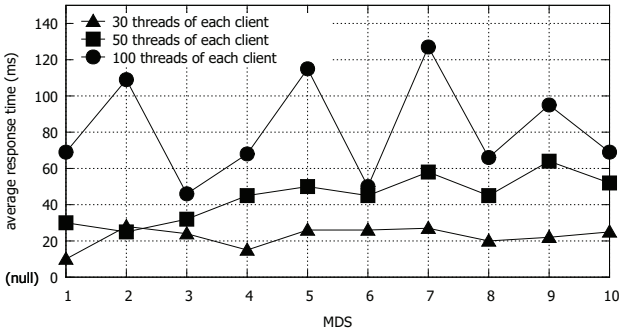

Figure 6. The degrees of FLS with different numbers of concurrent threads.

A. Implementation Details

We chose Hadoop filesystem (HDFS) as the baseline file system for optimization. HDFS 2.0 or a later release implements multiple namenode schemes and namespace management in terms of sub-tree partitions. The core metadata structure is INode, which organizes the namespace as a tree structure. INode is an abstract class, whereas INodeFile and INodeDirectory are the implementation for INode. HDFS responds to client requests through the interface getBlock-Location (), which is used to obtain the locations of the file blocks. As shown in Figure 4,. we use a key-value data structure instead of INode for managing metadata in the DHT caching layer. Key denotes FilePath, and value denotes the Block Location List. In addition, we implemented a new client protocol for the intercommunication between the client and DHT caching layer. Note that the DHT caching layer is independent of the original system.

TPPS In order to monitor MDS performance, we deployed ganglia on each node, which is an excellent cluster performance monitoring software. We parse its data structures and then output the performance indicators at any time period. TPPS runs when the MDS cluster shows a load skew, and the specific algorithm is shown in the TPPS section.

PARC We implemented a new cache replacement algorithm, just as the PARC section description. We used a replication strategy to organize the distributed cache, and then the client can randomly select any node and read hotspots. In the FLS mode, a small amount of hotspots cause flash crowds. Cache size does not depend on the total amount of metadata, which is also confirmed by Bin Fan et al. [4]. Besides, the DHT caching layer deployed in the metadata cluster and shared memory with MDSs, and the metadata size is smaller than common data. Based on the above considerations, we decided to set the cache space to 200M. In the details of the PARC, we chose 8 and 2 to be the cache size and ghost cache size, respectively. The essential reason is that cache stores blocks, and the ghost cache only stores block identifiers. So the cache size is greater than the ghost cache size, and such allocation can accelerate storage accessing and improve cache performance.

Benchmark We use the benchmark that built-in HDFS. To achieve a high concurrent workload, we extended the original benchmark for supporting multi-threads. In addition, we used an actual cloud storage project for the test, named CloudShare. This was developed by us for enterprise document sharing and collaboration.

TABLE II
IMPLEMENTATION DETAILS OF THE TESTBED

Item	Configuration
Hardware	Intel i7-2600 3.4GHz (4-cpu), RAM 16GB, HDD 2TB
OS	Ubuntu-11.04 x86_64
JDK	Hotspot 64-Bit Server VM (build 1.6.0_27-b07)
Hadoop	Version-0.23.3,similar with hadoop2.0.0
Test Client	NNbench
Monitor	Ganglia
Network	1000Mbps Ethernet, Catalyst-3750 10GigE switches

TABLE III
DETAILS OF NNBENCH WORK LOADS

Item	Workload-1	Workload-2
Operation Type	create_write	open_read
Request Distribution	uniform	hotspot
Job Count	1	10
Task Count/job	10	10
Thread Count	10*100	10*10*100
Smoothing Window(min)	15	15
File Count(*1000)	1000	10
Block Size(bytes)	1	1

B. Experiment Overview

Our experiments were performed on a 10-node MDS cluster. The configuration detail are listed in Table 2. HDFS utilizes multiple namenodes (MDS), called federation. Every namenode manages a part of the namespace tree, which is specified in the configuration file. As we focus on the test of the MDS cluster, the cluster nodes are all deployed as namenode.

C. Workload

The workload includes the *create_write* operation and *open_read* operation in *NNBench* (HDFS benchmark), detailed in Table 3. To simulate the real-world FLS scenario, two parameters are used to determine which MDS turns into load skew and how frequent it is. The one is the number of concurrent threads in each client, the other is the number of active metadata in each MDS. We used the different number of client threads and the metadata items to simulate the

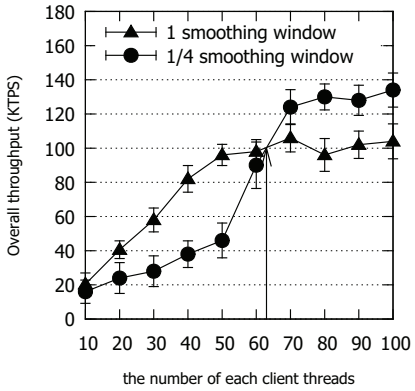

Figure 7. Accuracy of algorithm 1 with different length adjusting periods.

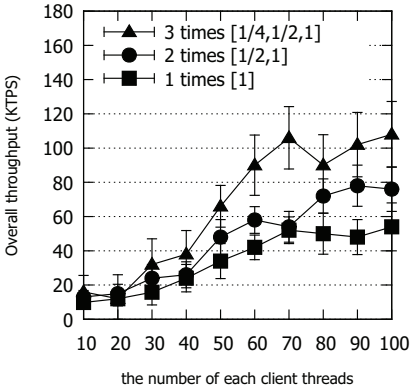

Figure 8. Accuracy of algorithm 2 with different concurrent adjusting times.

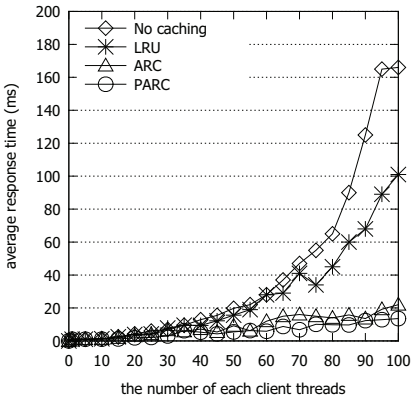

Figure 9. Comparing performance of PARC, APC, LRU, and no caching.

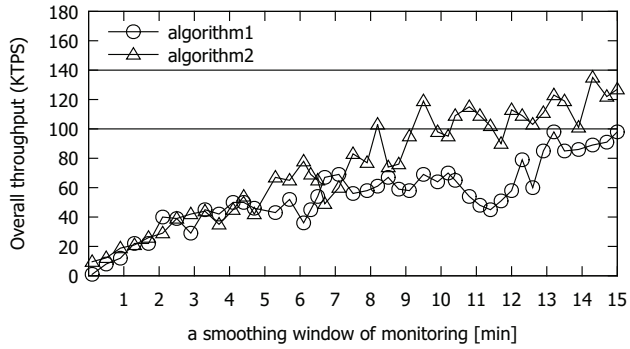

Figure 10. Comparison of quickness between algorithm 1 andalgorithm 2.

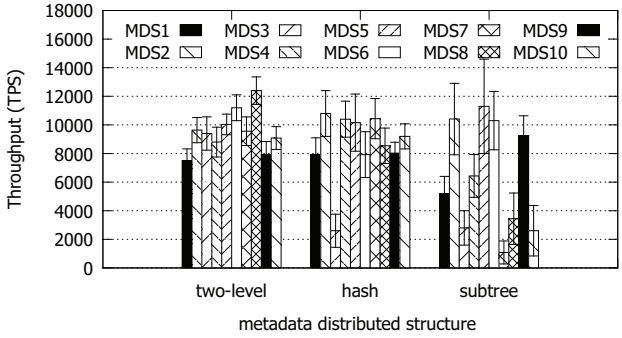

Figure 11. Performance comparison of our framework, hash, and sub-tree.

dynamic request distribution among the MDS clusters. With the concurrent client threads increasing, we ran a number of different request jobs, leading to the load's drastic change of each MDS. As HDFS federation uses the classic client side mounts table method, the client knows the distribution of all metadata. Figure 5 shows the uneven request distribution at the MDS over time; the other MDSs are similar. The abscissa represents a smoothing window of monitoring, and the ordinate represents the number of arriving requests (KTPS:1000* transactions per second). Figure 6 shows the average response time with different numbers of concurrent threads in the MDS cluster. The three max/min differences were 18 ms, 39 ms, and 81 ms. This indicates that the workload becomes increasingly skewed as the number of concurrent threads increases.

A realistic application is also used to validate our approach. CloudShare is a document sharing software for enterprises and is similar to *Doopbox* [24]. We collected the actual user operation records (mkdir, rmdir, and rename) and used them in loadrunner to generate a high concurrent workload. The recording contains two time indicators: T1 (loadrunner reaction time) and T2 (cloudshare calls the file system time). In addition, the web service response time could be calculated by T1-T2. In flash crowds access pattern, T1 becomes longer but (T1-T2) is steady, indicating a bottleneck in file system. We deployed our caching layer into cloudshare, after repeated running. T1 basically stays stable, which gives a rough certification for the effectiveness of our approach.

D. Verify "Accurately" and "Quickly" of TPPS

We verified the "Accurately" and "Quickly" of two prefetching algorithms. We repeated all experiments for 20 runs, and illustrate the average, max, and min of the overall throughput with an error bar. To study "Accurately", we ran algorithm 1 with different lengths of adjustment periods. Figure 7 demonstrates a critical point of the number of client threads,which is 63 as shown as arrows pointing, above which the shorter adjusting period means higher prefetching accuracy. Figure 8 presents the testing results of algorithm 2. According to Eq.(5), algorithm 2 only identifies load skew once to get the most potential overload MDS in a smoothing window at the probability of $1/e$. In order to get higher "Accurately", we simultaneously ran algorithm 2 multiple times in a smoothing window. In order to capture the different FLS, we configure different adjusting time periods in a smoothing window, for example 1/4, 1/2, and 1, as Figure 3 shows. We found that the more times algorithm 2 was simultaneously run, the higher overall throughput. However, the running times have an upper limit because the cost of prefetching cannot be tolerated in a small enough adjusting period. According to Eq.(4), we ran the algorithm $\ln n$ times. It can be seen in Figure 7 and Figure 8 that both algorithms have an ideal "Accurately". As for "Quickly", we simultaneously ran two algorithms

in a smoothing window. The result in Figure 10 implies that algorithm 2 is better. Algorithm 1 gets the maximum overall throughput (100KTPS) in 13 minutes, and algorithm 2 achieves it in 8 minutes. Moreover, algorithm 2 finally obtains the maximum throughput(140KTPS).

E. PARC vs Other Cache Replacement Algorithms

We compared the performances of PARC, ARC, and LRU, using average response. Based on testing results, the response time of a common metadata request is less than 10ms, so we take 10ms as the threshold to determine whether the client request drops in performance. As shown in Figure 9, the PARC algorithm has basically reached this standard. The original ARC algorithms also did a good job. Even so, the fact that PARC is better than ARC shows the effectiveness of the integration prefetching over caching. As LRU has ignored the requests frequency degree, it is almost inefficient. Meanwhile, it also shows that the FLS model makes a negative impact to the performance of the MDS cluster service, which is exposed by the no caching line.

F. Our Framework vs Subtree and Hash

We compared our framework with subtree and hash. All algorithms were run 20 times with 100 concurrent threads of each client. As shown in Figure 11, subtree is the worst case for the problem of request load balancing and our framework is the best one of these three structures. In the subtree partition situation, the fact that file and directory belong to the same directory results in aggregation of client requests. In addition, metadata distribution is always imbalanced with sub-tree partition. All of the above reasons determine its poor performance. As for the hash structure, there are some uneven request distributions. This is mainly caused by a large number of requests accessing a file or a directory at the same moment. Encouragingly, the independent caching layer is a useful way for *flash crowds*, which has also been validated in our experiment. Our framework consists of a basic distributed metadata management scheme and a distributed caching layer, which have greatly improved the effectiveness of load balancing. It should be emphasized that the error bar of our framework is the shortest among the three solutions, indicating that the range of load changing is small.

VI. Conclusion

We first designed a distributed cache framework above the metadata management service to handle potential multiple hotspots and *flash crowds* access patterns. We then proposed a TPPS and a PARC to improve the performance of the distributed caching layer to meet constantly changing workloads. By combining caching and prefetching techniques, our approach can improve throughput of MDS clusters, which was verified by an HDFS cluster.

VII. acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No. 61170074 and 61202065, the National High-Tech Research and Development Plan of China under Grant No. 2012AA011204, and the National Key Technology R&D Program of China under Grant No. 2012BAH05F02.

References

[1] Hadoop. http://hadoop.apache.org/. 2013.
[2] TFS.http://code.taobao.org/p/tfs/src/. 2013
[3] S.Ghemawat, H.Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.
[4] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *SOCC*, 2011.
[5] Jin Xiong, Yiming Hu, Guojie Li, Rongfeng Tang, and Zhihua Fan. Metadata Distribution and Consistency Techniques for Large-Scale Cluster File Systems. In *TPDS*, vol. 22, no. 5, pp. 803-816, 2011.
[6] Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bouge, Matthieu DORIER. BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map/Reduce Applications. In *IPDPS*, pp. 1-11, 2010.
[7] Weijia Li, Wei Xue, Jiwu Shu and Weimin Zheng. Dynamic hashing: adaptive metadata management for petabyte-scale file systems. In *MSST*, 2006.
[8] Moshe Babaioff, Nicole Immorlica, and Robert Kleinberg. Matroids, secretary problems, and online mechanisms. In *SODA*, pp. 434-443, 2007.
[9] Beth Trushkowsky, Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *FAST*, pp.163-176, 2011.
[10] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *FAST*, pp. 17-33, 2008.
[11] Quanqing Xu, Rajesh Vellore Arumugam, Khai Leong Yong and Sridhar Mahadevan. DROP: Facilitating Distributed Metadata Management in EB-scale Storage Systems. In *MSST*, 2013.
[12] Nimrod Megiddo and Dharmendra S.Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, 2003.
[13] Ali R.Butt, Chris Gniady, and Y.Charlie Hu. The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms. In *SIGMETRICS*, June 6-10, 2005.
[14] Pei Cao, Edward W. Felten, Anna R. Karlin, Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. In *TOCS*, vol. 14, no. 4, pp. 311-343, 1996.
[15] Drew Roselli, Jacob R. Lorch and Thomas E. Anderson. A Comparison of File System Workloads. USENIX Technical Conference. pp. 41-54, 2000.
[16] KV Shvachko. "HDFS Scalability: The limits to growth," Usenix Vol 35, no.3. www.usenix.org/publications/login/2010-04/openpdfs/shvachko.pdf. May 2010.
[17] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *SC*, 2004.
[18] Ismail Ari, Bo Hong, Ethan L. Miller, Scott A. Brandt and Darrell D. E. Long. Managing flash crowds on the Internet. In *MASCOTS*, pp. 246-249, 2003.
[19] Dhruba Borthakur, Jonathan Gray and Joydeep Sen Sarma. Apache hadoop goes realtime at Facebook. In *SIGMOD*, pp. 1071-1080, 2011.
[20] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan and Xiaodong Zhang. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In *FAST*, 2005.
[21] Doug Beaver, Sanjeev Kumar, Harry C.Li, Jason Sobel, Peter Vajgel. Finding a needle in Haystack: Facebook' sphotostorage. In *OSDI*, 2010.
[22] Wei-Guang Teng, Cheng-Yue Chang, and Ming-Syan Chen. Integrating Web Caching and Web Prefetching in Client-Side Proxies. In *TPDS* ,VOL. 16, NO. 5, 2005.
[23] Yifeng Zhu, Hong Jiang, Jun Wang, and Feng Xian. HBA:Distributed Metadata Management for Large Cluster-Based StorageSystems. In *TPDS*, vol. 19, no.6, pp. 750-763, 2008.
[24] Doopbox. www.dropbox.com/. 2013.