THE UNIVERSITY OF

WARWICK

Original citation:
Fu, Songling, He, Ligang, Huang, Chenlin, Liao, Xiangke and Li, Kenli. (2014)
Performance optimization for managing massive numbers of small files in distributed file
systems. IEEE Transactions on Parallel and Distributed Systems . ISSN 1045-9219

Permanent WRAP url:
http://wrap.warwick.ac.uk/65275

A note on versions:
The version presented here may differ from the published version or, version of record, if
you wish to cite this item you are advised to consult the publisher's version. Please see
the 'permanent WRAP url' above for details on accessing the published version and note
that access may require a subscription.

For more information, please contact the WRAP Team at: publications@warwick.ac.uk
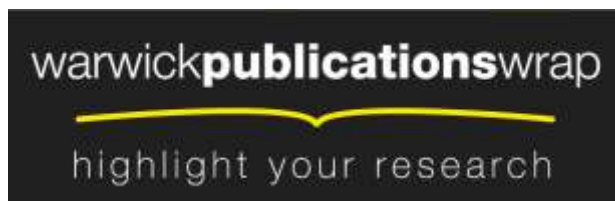
warwickpublicationswrap

highlight your research

http://wrap.warwick.ac.uk

Performance Optimization for Managing Massive Numbers of Small Files in Distributed File Systems

Songling Fu, Ligang He, Chenlin Huang, Xiangke Liao, Kenli Li

Abstract—The processing of massive numbers of small files is a challenge in the design of distributed file systems. Currently, the combined-block-storage approach is prevalent. However, the approach employs the traditional file systems such as ExtFS and may cause inefficiency when accessing small files randomly located in the disk. This paper focuses on optimizing the performance of data servers in accessing massive numbers of small files. We present a Flat Lightweight File System (iFlatLFS) to manage small files, which is based on a simple metadata scheme and a flat storage architecture. iFlatLFS is designed to substitute the traditional file system on data servers and can be deployed underneath distributed file systems that store massive numbers of small files. iFlatLFS can greatly simplify the original data access procedure. The new metadata proposed in this paper occupies only a fraction of the metadata size based on traditional file systems. We have implemented iFlatLFS in CentOS 5.5 and integrated it into an open source Distributed File System (DFS), called Taobao FileSystem (TFS), which is developed by a top B2C service provider, Alibaba, in China and is managing over 28.6 billion small photos. We have conducted extensive experiments to verify the performance of iFlatLFS. The results show that when the file size ranges from 1KB to 64KB, iFlatLFS is faster than Ext4 by 48% and 54% on average for random read and write in the DFS environment, respectively. Moreover, after iFlatLFS is integrated into TFS, iFlatLFS-based TFS is faster than the existing Ext4-based TFS by 45% and 49% on average for random read access and hybrid access (the mix of read and write accesses), respectively.

Index Terms—Distributed File System; Data Server; Small File; Performance Optimization

——————————   ◆   ——————————

1 INTRODUCTION

With the extreme popularity of social networks (e.g., Facebook [22]) and e-commerce sites (e.g., Amazon [23] and Taobao [24]), new challenges are raised to efficiently store and access massive numbers of small files, such as user messages and merchandise photos. The small files in such scenarios have their special characteristics. Typically, these small files are seldom modified and their sizes range from several KBs to tens of KBs [1-3]. For example, it is shown in [9] that the Taobao File System (TFS) used by Taobao, which is the No.1 e-commerce site in China, manages about 28.6 billion photos and the average photo size is 17.45KB. This is different from the case of big files. For example, Reference [4] shows that the files processed by some Google applications have the size of multiple GBs.

Distributed File System (DFS) is a method of storing and accessing files based on a client/server architecture, as shown in Fig. 1. In a DFS, all files are replicated and stored in many data servers, while the metadata are stored in the metadata server. A client (note that a client in this context is not a user, but an application server which interacts with DFS and provides the services to the users) must look up a file by the metadata server in a DFS, not by a file path, which is a typical way of looking up a file in traditional file systems. As shown in Fig. 1, a client accesses a file stored in a DFS by two phases: 1) querying the metadata

server to get the IP address of the data server which stores the target file; 2) connecting the data server to fetch the file data.

In traditional DFS, such as GFS [4] and HDFS [5], the divided-block-storage approach is designed to handle big files, each of which is divided into multiple fixed-size (typically 64MB) data blocks. Each data block is stored as a regular file in multiple data servers. There is at least one metadata object in the metadata server and one regular file on the data server for each file. When the files in the systems are mainly small files, the number of stored files increases sharply, resulting in a huge amount of metadata on the metadata server and the low performance in accessing files on data servers [6], [7], [8]. Therefore, it is a popular research topic in the design of DFS to improve the performance of processing massive small files.

Currently, combined-block-storage is a prevalent approach to processing massive small files and has been implemented in both Haystack [3] by Facebook and TFS [9] by the Alibaba Group. The fundamental idea of combined-block-storage is to combine small files into large data blocks to reduce the magnitude of metadata on the metadata server. The Combined-block-storage approach adopts traditional file systems, such as Ext4 [10] and XFS [18], to store data blocks as regular files on the data servers. However, traditional file systems often have very low performance when handling massive small files [11], [12].

Since the accesses to the hotspot data are commonly filtered by the multi-level cache techniques in commercial systems, such as CDN [14], the data requests arriving at the actual data servers situated behind multiple levels of caches are most likely to access the non-hotspot data that are randomly distributed in the disk. Therefore, there is no need to cache the accessed data into the memory on data servers. This is why improving I/O

————————————————

- *Songling Fu is with the School of Computer Science, National University of Defense Technology, Changsha, China. E-mail: slfu@nudt.edu.cn.*
- *Ligang He is with the Department of Computer Science, University of Warwick, Coventry, UK, E-mail: liganghe@dcs.warwick.ac.uk.*
- *Chenlin Huang is with the School of Computer Science, National University of Defense Technology, Changsha, China. E-mail: clhuang@nudt.edu.cn.*
- *Xiangke Liao is with the School of Computer Science, National University of Defense Technology, Changsha, China. E-mail: xkliao@nudt.edu.cn .*
- *Kenli Li is with the School of Information Science and Engineering, Hunan University, Changsha, China. E-mail: lkl@hnu.edu.cn*

performance of data servers becomes a critical issue.

The steps involved in a data access can be broadly divided into the steps of determining the location of the data through the metadata and the steps of accessing the physical data. The former can be regarded as the I/O overhead. It makes accessing big files very different from accessing small files, because when accessing small files, the time associated with the I/O overhead outweighs the time spent in accessing physical data. Therefore, when optimizing the performance of accessing small files, the major efforts often focus on reducing the I/O overhead.

Previous research work has presented the approaches to improving the performance of DFS by optimizing data management on data servers. Haystack [3], which is a HTTP-based photo server developed by Facebook, presents an index cache technique to eliminate the disk I/O operations when locating small photo files. It also adopts XFS to speed up accessing photos. In TFS [9], Alibaba proposes a naming technique and a disk-pre-allocating technique to simplify metadata management and decrease the number of I/O requests during file accesses.

However, there is still room to further improve the performance of these existing DFSes. For example, XFS can only deliver about 85% of the raw throughput on data servers installing Haystack [3], while the Ext4 file system delivers about 50% in TFS as shown in Section 5.

This paper focuses on optimizing the performance of data servers in accessing massive numbers of small files. We developed a Flat Lightweight File System called iFlatLFS. iFlatLFS is able to access raw disks directly, optimize the metadata management and accelerate the data-accessing procedure on data servers. iFlatLFS is designed as an efficient alternative to the underlying traditional file systems, as shown in Fig. 1.
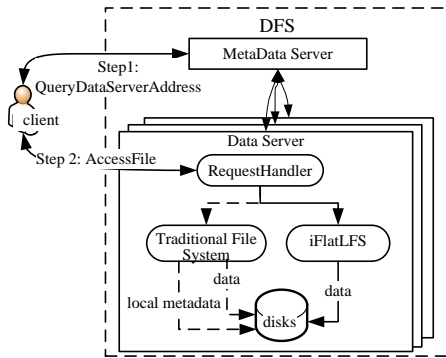

Fig. 1 The relation between iFlatLFS and DFS.

The traditional file systems can offer many benefits in general. However, in the systems we are targeting, there are massive numbers of rarely-modified small files. Under this circumstance, some operations and functionalities in traditional file systems are unnecessary or cause excessive overheads. The design of iFlatLFS carefully weighs the efficiency and the generality of the measures adopted in traditional file systems. The efficiency of iFlatLFS is achieved by removing or re-designing the measures that are not suitable in our targeted systems from the performance perspective, although these measures are adopted in traditional files systems for sake of generality in application. For example, namespace management is re-designed in iFlatLFS, so that iFlatLFS performs little more than space management, relying on the upper distributed file system (In this paper, the DFS components sitting above the local file system, such as Request handler and Metadata server in Fig. 1,

are called the upper DFS) to handle all naming issues.

Although iFlatLFS can substitute the traditional file systems, the process for a DFS client to access a file remains unchanged. The main difference between iFlatLFS and traditional file systems is that iFlatLFS accesses the data directly from disks while traditional file systems access the data through the hierarchical file tree. In iFlatLFS, a simple metadata scheme is designed, in which the metadata occupies much smaller space. As a result, the metadata can be entirely cached in memory and consequently the operation of fetching the metadata from the disks is eliminated during the file access. Moreover, a flat storage architecture is developed in iFlatLFS to store small files in raw disks. With this new flat storage architecture and the simple metadata scheme, the original complex process of accessing files in traditional file systems is greatly simplified and the amount of metadata is significantly reduced. Performance evaluation has been carried out to verify effectiveness and advantages of iFlatLFS. The evaluation results show that an iFlatLFS-based data server can achieve near optimal performance.

The rest of this paper is organized as follows. Section 2 discusses related. Section 3 analyzes the file access model on data servers. Section 4 presents the design details of iFlatLFS, including fundamental ideas, data management, file access, metadata consistency, and the implementation. Section 5 evaluates the performance of the developed file system and analyzes experimental results. Section 6 proposes a hybrid storage system. Finally, Section 7 concludes this paper.

2   RELATED WORK

DFS has been widely studied in recent years [4], [5], [25], [26-28]. GPFS [25] and PVFS [26] are designed for managing the files in clusters. TokuFS [27] is implemented using Fractal Tree indexes, which are primarily used in databases. BlueSky [28] acts as a proxy of multiple cloud storage providers to offer storage service for enterprise users. GFS [4] and HDFS [5], are designed mainly for streaming access of large files.

Some studies [1], [3], [9], [32], [33] show that small files occupy a big fraction of the entile files in current distributed systems. For example, Reference [3] shows that the photos in facebook have the average size of 64+KB.

Three main optimization approaches have been developed to support small files in DFSes: 1) reducing data block sizes; 2) combining small files into bigger ones; 3) storing small files by groups, not by data blocks.

Google's next generation of DFS, GFS2 [15], is an example of taking the first optimization approach. Zhang et al. [30] also proposed an approach which changes the block size and adopts an efficient indexing mechanism. GFS2 improves the ability of handling small files mainly by reducing the data block size from 64MB to 1MB. In order to meet the requirements of Internet applications, the data blocks must be reduced further to the size of the KB order. In this type of DFS systems, there will be over 1G data block files and 150GB inode data on a data server of 10TB storage capacity. Although the existing proposals in [29] and [31] can be used to impove the performance of the metadata server, it is difficult for data servers to handle such massive numbers of data block files.

The second approach, i.e., the approach of combining small files into bigger ones, falls into two categories: i) developing

dedicated tools, such as Hadoop Archive [16], Sequence File [17], for handling small files based on existing DFSes, and ii) designing new DFSes, such as Haystack [3] and TFS [9], with special optimization considerations for small files. Hadoop Archives, Haystack and TFS are discussed below in more detail.

Hadoop Archives (HAR) builds a layered file system on top of HDFS, and packs the small files into relatively big HDFS files to alleviate the pressure caused by too many files on the metadata server's memory. Sequence File is similar to HAR. It uses the file name as the key, and the file content as the value. Reading files in a HAR or Sequence File is slower than reading files in HDFS because of additional index operations.

Haystack and TFS are the same type of DFS, which are designed for the storage of massive small photos in their own companies: Facebook and Alibaba. They employ the *combined-block-storage* technique, which combines small photos with different sizes into data block files with a fixed size, and builds an index file for each data block file. The index file stores the logical addresses of all photos. In order to further improve performance of data servers, some other measures have also been taken. Haystack caches all index data into the memory and adopts XFS [18] on data servers, which can perform random seeks quickly within a large file. TFS codes a file's logical address into its file name to decrease one lookup operation. All these approaches can simplify the metadata lookup process and reduce the number of disk operations to some degree. However, both Haystack and TFS employ traditional file systems on data servers. The additional data locating operations are needed to lookup photos' physical address in disks before the actual file access can start, which can cause low performance of data servers. According to Facebook's benchmarking performance [3], Haystack can deliver only 85% of the raw throughput of the device while incurring 17% higher latency.

FastDFS [19] is an exemplar DFS which adopts the third approach, i.e., storing small files by groups. It is an open source high performance DFS, and can meet the requirements of photo sharing sites and video sharing sites. In FastDFS, files are organized in groups, and each file exists in only one group (i.e., there are no common files between different groups). For each file, the file name exposed to clients is composed of its corresponding group name, relative path and its actual file name on the data server. The shortcoming of FastDFS is that it is difficult to handle massive numbers of KB-sized small files because the number of files is too large.

Additionally, some other techniques have also been developed recently to improve the performance of traditional file systems on a single server. For example, both Ext4 [10] and XFS [18] introduce the concept of *extent* in order to reduce metadata. This approach is only effective for big files. ReiserFS [35] employs EOTTL (extents on the twig level) and Ligquid items. The former is a fully balanced storage tree, which guarantees that all paths to the objects are of equal length, while the latter is a special format of records in the storage tree and can solve the problem of internal fragmentation. ReiserFS can optimize the performance of handling small files.

As discussed above, designing new DFSes (e.g., Haystack and TFS) is a category of the second optimization approach to support small files. These DFSes develop the combined-block-storage technique on top of traditional file systems. Our work aims to further optimize such DFSes. A lightweight file system,

called iFlatLFS, is developed in this paper to substitute the traditional file system in the DFS (to differentiate with iFlatLFS, the DFS is called the *upper DFS* in the rest of this paper).

3 ANALYSING DATA ACCESS FOR SMALL FILES

3.1 Data accessing in DFS

As discussed in the introduction section, a client accesses a file stored in a DFS by the following two phases.

1) Querying the metadata server to get the IP address of the data server that stores the file. This phase has 3 steps:

1. $T_{ClientQueryMDS}$: The client sends a message to the metadata server to query the ID of the data block which store the file data, and the IP addresses of the data servers which store the data block;
2. $T_{MDSQueryMD}$: The metadata server queries locally the ID of the data block and the IP addresses of the data servers. In the case where the client writes a new file, the metadata server will allocate an old data block which has free space or create a new data block;
3. $T_{MDSReturnMD}$: The metadata server returns the ID of the data block and the corresponding IP addresses of the data servers to the client.

2) Selecting and connecting a data server to fetch the file data. This phase also includes 3 steps:

4. $T_{ClientAccessDS}$: The client sends a message to the data server. In the case where the client writes a new file into the DFS, the message contains the file data;
5. $T_{DSAccessData}$: The data server reads (or writes) the file data from (or into) the data block in the local disks;
6. $T_{DSReturnResult}$: The data server returns the result to the client. In the case where the client reads a file from the DFS, the result contains the file data.

Therefore, the total time of accessing a file from a DFS can be expressed as in (1).

$$T_{DFS} = T_{ClientQueryMDS} + T_{MDSQueryMD} + T_{MDSReturnMD}$$
$$+ T_{ClientAccessDS} + T_{DSAccessData}$$
$$+ T_{DSReturnResult} \quad (1)$$

We observe the following three key points from (1).

1) In a DFS, the metadata server manages the namespace of all files, which is usually organized into a hierarchical tree. The namespace is stored as a regular file, and will be loaded into the memory entirely when the server boots. The metadata server can obtain the required metadata by in-memory operations. So Step 2 ($T_{MDSQueryMD}$) only takes little time, which can be neglected comparing with over-network or I/O operations.

2) On the contrary, the file data are stored as regular files in a data server. As the discussed in Section 1, when the data requests arrive at the actual data servers, which are situated behind multiple levels of caches in DFS, it is most likely that these requests are accessing the non-hotspot data. Therefore, the file data to be accessed are typically not in the memory, but in the disk. The data server has to retrieve the file data by one or more disk I/O operations. Therefore, Step 5 ($T_{DSAccessData}$) typically takes much more time than Step 2.

3) The time of steps 1, 3, 4 and 6 mainly depends on the network bandwidth. In the current mainstream hardware environments, the bandwidth of networks, such as Infiniband, is usually much bigger than the bandwidth of

disk I/Os, such as SATA disks and SAS disks. For example, the TianHe-2 supercomputer [43, 44, 45], which is developed by the National University of Defense Technology in China and is ranked No. 1 in the latest Top500 Supercomputer list, is equipped with one of the two interconnecting networks: Infiniband network or the high-speed NIC-based (network interface chip) network. The bandwidth delivered by the former in the supercomputer is about 7GB/s, while that delivered by the latter is about 20GB/s. But the tested bandwidths of SATA disk and SAS disk in the supercomputer are about 70MB/s and 100MB/s, respectively. Therefore, compared with Steps 1, 3, 4 and 6, it is far more likely that Step 5 becomes the bottleneck in the workflow of steps for accessing the file data in DFS.

The above analysis shows that reducing the time spent by Step 5, i.e., the local data accessing in data servers, is critical in order to optimize the performance of accessing the file data in the DFS. This motivates our work to optimize the accessing of small files in data servers. Next, we will analyze the steps and the performance of local data accessing in a data server.

3.2 Analytical model of read operations in a data server

The traditional file systems, which access files through a hierarchical file tree, are commonly adopted on data servers to manage file data and metadata in disks. Small files are combined into data blocks of fixed size. Each data block is stored as a regular file, called a *data block file*. For each data block file, an *index file* is built to store the logical offset addresses of all small files in this data block file.

In traditional file systems such as ExtFS, all files are organized into a hierarchical tree. A file is divided into disk blocks, which may not be stored in the continous physical locations in the disk. A file has an inode, which uses the address pointers to link the scattered disk blocks of a file together, so that the file has a continous logical address in the file system. Each disk block of a file has a typical size of 1KB. In ExtFS, therefore, every 1KB data needs an address pointer, whose size is 8 Bytes. In all address pointers of the disk blocks, the first 12 pointers are stored in the file inode structure, and others are stored in disk blocks called address blocks. In ExtFS, an application accesses the file data by their logical addresses, not by their physical addresses (i.e., the pointers of the disk blocks). Thus during the file access, there is an extra data lookup operation of retrieving physical addresses, which may be located in the inode or in some address block depending on the logical address.

In DFS environments, read operations dominate, compared with writes [3], [9]. In this section, we analyze the performance of reading small files on a data server (the write performance will be discussed in Section 4.3 when we discuss data consistency). Section 3.2.1 and Section 3.2.2 present the analytical models without and with considering the effects of memory caches, respectively.

3.2.1 Models without considering cache effects

As shown in Fig. 2, the process of file read on data servers includes 7 steps, which can be organized into 2 phases as follows.

Phase 1: Reading local metadata to retrieve the logical address of the target file data in the corresponding data block file. This phase includes 3 steps:

1. $T_{1:ReadIFInode}$: *Reading the inode of index file;*
2. $T_{2:ReadIFData}$: *Reading index data from the index file;*

3. $T_{QueryLA}$: *Querying the corresponding index item from the index data to get the logical address of the file.*

Phase 2: Reading file data. This phase includes 4 steps:

4. $T_{3:ReadDBFInode}$: *Reading the inode of the data block file;*
5. $T_{4:ReadAddressBlock}$: *Reading the corresponding address block from the disk if the logical address is beyond the size of 12 disk blocks;*
6. $T_{QueryPA}$: *Querying the physical address of the target file data from the inode or the address block;*
7. $T_{5:AccessData}$: *Accessing file data using a disk operation.*
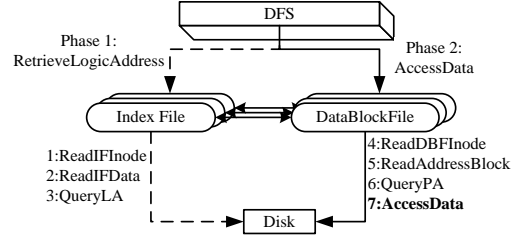

Fig. 2 File access model in DFS based on traditional file systems

It can be seen from above that the data management model in traditional file systems is very complex. The metadata needed for file access on data servers include the inodes of index files, the inodes of data block files, the index data and all address blocks of data block files. Consequently, the metadata can be huge. For example, when Ext4 is used, 92.8GB of metadata is needed to store 10TB data with the average file size of 16KB. The total time of reading a file on a data server can be expressed in (2).

$$T_{original} = T_{1:ReadIFInode} + T_{2:ReadIFData} + T_{QueryLA} \\ + T_{3:ReadDBFInode} + T_{4:ReadAddressBlock} \\ + T_{QueryPA} + T_{5:AccessData} \qquad (2)$$

Each variable T_i in the right side of (2), except $T_{QueryLA}$ and $T_{QueryPA}$, represents the time of a disk operation. There are total 5 disk operations during a file read, although only $T_{5:AccessData}$ is the time which is used to read the actual data being requested. Because the time of a disk I/O operation is far heavier than that of a memory operation, Equation (2) can be simplified as (3).

$$T_{original} \approx T_{1:ReadIFInode} + T_{2:ReadIFData} + T_{3:ReadDBFInode} \\ + T_{4:ReadAddressBlock} + T_{5:AccessData} \qquad (3)$$

It can be seen from the above analysis that although data servers only store data for the upper DFS, its complex hierarchical file management model incurs heavy overhead.

3.2.2 Models with the cache effect

The effects of the memory cache are not considered in the analysis model presented in Section 3.2.1. In fact, the memory cache acts as an important role while designing the performance optimization approaches.

If any performance optimization approaches used in Haystack and TFS (as discussed in the related work section) are applied, Equation (3) can be further simplified. For example, in TFS, the index files are accessed using the *mmap* operation while the direct I/O operation is used for accessing data block files. In the *mmap* operation, the index file is mapped to memory, and the index data are transparently loaded from disk into memory. An application can then access the index file in the same way as it accesses the memory. Further, the direct I/O operation can bypass the cache of the Ext4 file system, therefore greatly saving the main memory. The saved main memory can be used to cache the inodes and the index data. Thus the

data access time in TFS can be further simplified as (4), where $T_{2:ReadIFData}$ can be regarded as zero if the required index data have been in memory cache.

$$T_{original\_TFS} \approx T_{2:ReadIFData} + T_{4:ReadAddressBlock}$$
$$+ T_{5:AccessData} \qquad (4)$$

3.3 Performance benchmarking in a data server

We tested the performances of random read and write for small files with different sizes on CentOS 5.5 (the kernel is 2.6.18). 10GB data are stored into 160 files, each with the size of 64MB. In the test, a file is selected randomly, and a logical address is selected randomly from this file. The test then begins to read or write data ranging from 1 KB to 64 KB in size at the selected logical address. For each size, the same test is performed 1000 times. The results are shown in Fig. 3, where the y-axis is the time of the data access (i.e., overhead).

The result shows that for the same disk operation (read or write), the overhead of accessing small files of different sizes is almost the same, and that the response time is almost independent of file size. This is because the disk seek time is far longer than data transferring time for small files. These results indicate that if the number of disk operations can be reduced when accessing small files, the data accessing performance can be improved. Ideally, each file access needs only one disk operation if all data locating operations are performed in memory.

Based on the above models and the benchmarking results, this paper aims to develop a flat file system (called iFlatLFS) to reduce I/O overhead for accessing small files.
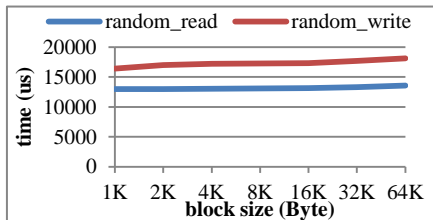

Fig. 3 Random read and write performance in Ext4

4 iFLATLFS

The design of iFlatLFS is mainly oriented towards optimizing performance of accessing massive small files which are seldom modified. We assume that all files are stored using the combined-block-storage approach in DFSes, that is, small files will not be broken into pieces, but are combined into huge data blocks and stored into data servers. We also assume that there is no need to cache the accessed data into the main memory of the data server. This assumption is reasonable because the data requests are filtered by multiple levels of caches, as discussed in the introduction section. Therefore, the data requests that finally arrive at the data server are most likely to access the non-hotspot data that are randomly distributed in the disk space. iFlatLFS aims to substitute the traditional files systems on data servers in the our targeted systems, namely, the systems in which there are massive numbers of rarely-modified small files.

4.1 The design of data management

4.1.1 The fundamental idea

The fundamental idea of iFlatLFS is to improve the performance of file access on data servers by optimizing local metadata management and accelerating the data-accessing procedure. Instead of using the file-based local metadata management as in traditional file systems, a simple metadata management mechanism (discussed in Fig. 5) is designed in iFlatLFS. It combines file metadata, such as file ID, file type, creation time, checksum etc., into the file header and stores it together with the file data. As a result, the size of the metadata for a single file is cut down to 12 Bytes, and it is possible to load the entire metadata into the memory when the data server boots. For example, using the metadata management mechanism developed in this work, 10TB data will only need about 7.5GB of metadata when the average file size is 16KB. By doing so, all metadata operations can be performed in memory. Consequently, the overhead involved in locating the file data can be greatly reduced without the need of performing disk I/O operations.

There are many fasincating functionalities in traditional file systems. However, many of these funcationalities are not necessary or even hurt the performance of the file system deployed in the data servers in the DFS context. These unnecessary funcationalities mainly include:

1) Hierarchical file structure: Data server is to DFS what hard disk is to operating system. Since DFS manages the namespace of the data in the metadata server, data servers can simply act as the storage facility for DFS and there is no need for the file system in a data server to include complicated data management model. Therefore, in order to speed up data accessing, the complex file management designs in traditional file systems, such as hierarchical file structure and inode, are abolished in iFlatFLS. Instead, one disk partition is used to store all small files and their metadata for each data block. The files are directly accessed from the disk partitions in a flat fashion by means of physical address and data size in metadata.

2) Data prefetching: Since iFlatLFS handles small files, data prefetching in the traditional file systems has a negligible positive impact on the performance. So this function is unnecessary.

3) Data caching: As discussed in Section 1, the accesses to the hotspot data are commonly filtered by the multi-level cache techniques in DFS, the data requests arriving at the data servers are most likely to access the non-hotspot data. Therefore, data caching is unnecessary.

4.1.2 The flat storage architecture

In order to optimize the management of massive numbers of small files, a flat storage architecture is designed in iFlatLFS. The data management is also greatly simplified with smaller metadata and fewer data blocks.

In iFlatLFS, each disk partition consists of a header followed by a series of storage areas, each of which contains a header, a metadata area and a data area. The layout of a disk partition is shown in Fig. 4. For each data block, all files and their metadata are stored directly into a fix-sized super big disk partition, whose size is typically set as the greatest common divisor of all disk sizes, but can be as big as possible in theory.

Fig. 5 shows the flat storage architecture developed in this work. Each data area holds a vast amount of small files, which are stored in sequence. Each file consists of a header and its data. Each metadata area consists of a sequence of index slots. Each slot represents a small file stored in the corresponding data area. When a small file is created a corresponding index item is built and stored into an index slot. To reduce the metada-

ta as much as possible, each index item includes only 2 fields: the file size (4 Bytes) and the file physical address (8 Bytes) in disk. Other metadata, such as file identifier, file type, file size, checksum of file data, are stored in the file header. The index slot id will be coded into the file identifier as the file's logical address and returned to the upper DFS.

The disk space of the metadata area and the data area should be determined dynamically according to the DFS parameters such as data block size and average file size. The data block size can be determined by the hardware configuration of data servers, and the average file size can be derived through the statistical means or historical data [4], [9]. The metadata area cannot be changed after the disk space has been allocated. Once the metadata area runs out of its disk space, the free space of the data area will be automatically re-allocated as a new storage area and linked to the last storage area.
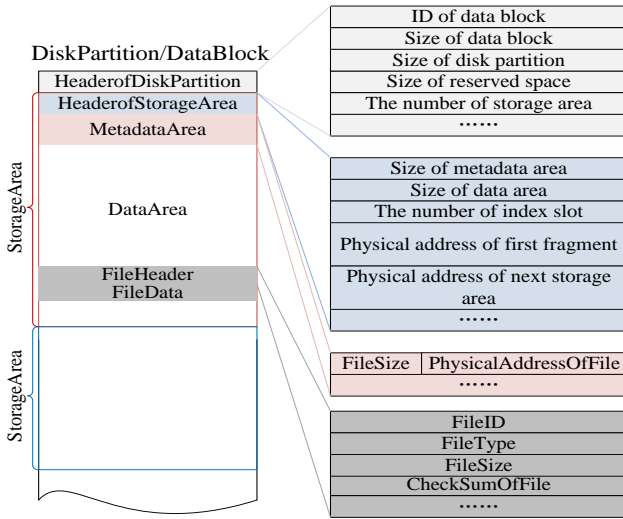

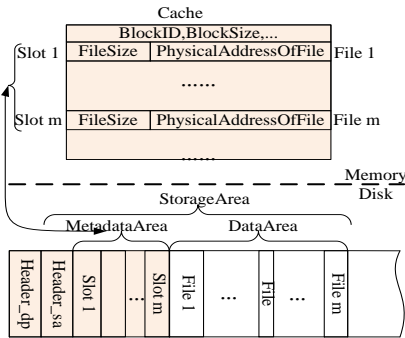Fig. 4 The Format of a disk partition in iFlatLFS

Fig. 5 The flat storage architecture in iFlatLFS

4.1.3 Managing reserved space

The physical damage of disk sections will reduce the storage space. Also, if there are too many fragments in disk partitions, then the effective storage space is also shrinked. In order to handle the shrinking of storage space, iFlatLFS introduces reserved space for each disk partition. The size of reserved space can be set by administrators, typically 1~10% of the whole partition space. In principle, new files will not be stored in the disk partition after its data areas run out of their disk space. However, if the files are generated because of upgrading the old files to bigger sizes or recovering the lost files due to physical disk failure, they are allowed to be stored in the reserved space. When the total faulty disk space is bigger than the reserved

space, the disk partition will be abandoned and the corresponding data block be stored in another disk partition.

4.1.4 Fragmentation management

Fragments may be generated in each disk partition when deleting or modifying files. iFlatLFS designed a strategy for fragmentation management, which is illustrated in Fig. 6. A special flag and the fragment size are stored in the fragment header. All fragments form a list of unused disk space, whose head (i.e., FirstFrag in Fig. 6) is stored in the metadata header. Normally, when a new file is generated, it is appended to the corresponding disk partition, and the partition pointer, which points to the first physical address of the available disk space, moves forward by the size of the file accordingly. After the partition pointer reaches the end of the disk partition, iFlatLFS must find a fragment available to store the target file. Although the concept of reserved space is introduced in iFlatLFS, there is no disk space physically allocated for the reserved space. iFlatLFS views all fragments as a part of the reserved space. Thus, the fragments are not used unless the total disk space of all fragments is bigger than the pre-set size of the reserved space. If there are too many fragments in a disk partition, iFlatLFS can copy these scattered files into another new disk partition, and the original partition will be assigned to another data block. The benefit of this fragment management strategy is to simplify the complexity of disk space management.
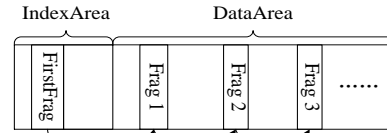

Fig. 6 The fragment management in iFlatLFS

4.2 File access

We now present how to access the small files. The file access model in a DFS with iFlatLFS is shown in Fig. 7.
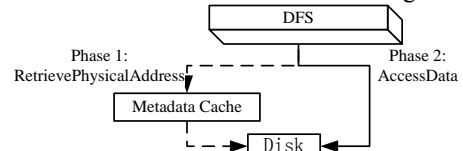

Fig. 7 The file access model of DFS with iFlatLFS

iFlatLFS is transparent for the clients of upper DFSes. Compared with the original file access model, the only difference is the process of the file access on the local data server. All metadata (i.e., the MetadataArea as shown in Fig. 5) are loaded and cached in memory from disk when the data server boots, so that all metadata operations can be performed in memory.

The whole process of file access includes 2 phases.

Phase 1: Querying the physical disk address of the target file from the metadata cache (i.e., the metadata loaded into the main memory). No disk operation is needed. The time overhead of this phase is denoted as $T'_{QueryPhysicalAddress}$.

Phase 2: Accessing the file data directly from the corresponding disk partition by one disk operation. Its time is denoted as $T'_{1:AccessData}$.

(5) shows the total time for file access in iFlatLFS.

$$T_{iFlatLFS} = T'_{QueryPhysicalAddress} + T'_{1:AccessData} \quad (5)$$

Based on the same reason of simplifying (2), Equation (5) can be simplified as (6).

$$T_{iFlatLFS} \approx T'_{1:AccessData} \qquad (6)$$

The time of data access from disks with iFlatLFS is equal to that in a traditional file system, (7) holds.

$$T'_{1:AccessData} \approx T_{5:AccessData} \qquad (7)$$

Thus, Inequality (8) holds.

$$
\begin{aligned}
T_{iFlatLFS} &\approx T'_{1:AccessData} \approx T_{5:AccessData} \\
&< T_{1:ReadIFInode} + T_{2:ReadIFData} + T_{3:ReadDBFInode} \\
&\quad + T_{4:ReadAddressBlock} + T_{5:AccessData} \\
&= T_{original}
\end{aligned} \qquad (8)
$$

Inequality (8) shows that the performance of DFSes based on iFlatLFS is higher than that based on traditional file systems. Additionally, by comparing (4) and (7), we can draw the same conclusion that $T_{iFlatLFS}$ is smaller than $T_{original\_TFS}$ with the cache effect considered.

In summary, any file can be accessed once its physical address is found from the metadata cache on a data server installing iFlatLFS. iFlatLFS needs only one disk operation to access each small file. However, when a small file is accessed with a traditional file system, the following steps need to be performed. Firstly, an index file must be read, and the corresponding index item must be found. Then the physical address of the file is retrieved from the disk. Finally, the disk operation is performed to access the file. Therefore, in a traditional file system, 5 operations are typically required to access a file.

4.3 Metadata consistency

After all metadata in disks are cached into the memory, there will be two copies of metadata in the system: one is in the disks and the other in the memory. Usually, the applications only access the memory to retrieve the metadata. It is a technical challenge to maintain the consistency of the two copies of metadata, especially when the unexpected server failures occur.

All current major traditional file systems are either journaling file systems (such as Ext4, XFS, ReiserFS) or Copy-on-Write file systems (such as ZFS, Btrfs). Metadata or data consistency is protected in these file systems. However, many good functionalities in the traditional file systems for maintaining consistency are far beyond what the DFSes need. Only the Write through and write back policies are sufficient for a DFS.

In iFlatLFS, two metadata consistency policies have been implemented: 1) Strong Consistency (write through) policy and 2) Weak Consistency (write back) policy. In the strong consistency policy, a write operation is atomic. After receiving new data from applications, iFlatLFS first writes these data and their metadata into disks, then updates the metadata into memory, and lastly returns a value to the application. All read operations on metadata are still executed in memory. In weak consistency, the new data and the corresponding metadata are both stored in memory temporarily, and a value is returned to the application. iFlatLFS only periodically writes the dirty data and metadata back into the disks to prevent data loss caused by the unexpected server failures. In case of the server failures, those dirty data which have not been written into the disks are lost.

When there are only read operations, no metadata consistency operations are required. Consistency has to be considered only when writes are performed. We established the analytical model in the above three scenarios to compare the performance of the DFS based on traditional file systems, iFlatLFS under the strong consistency policy and iFlatLFS under weak consistency.

4.3.1 Modeling write operations for traditional file systems

Journaling functionality has a marked negative impact on the write efficiency in traditional file systems. For example, by default, Ext4 first writes the data to the disk, then writes its related metadata to the journal, and finally checkpoints the metadata to the disk. Therefore, in the following modeling process, we do not consider the journaling effect. If the journaling functionality is considered, the worse performance is expected.

T_{ow} denotes the time of a write operation in a DFS based on the traditional file systems. The analytical model of T_{ow} can be represented in (9).

$$
\begin{aligned}
T_{ow} = T_{1:ReadIFInode} &+ T_{2:WriteIFInode} + T_{3:ReadDBFInode} \\
&+ T_{4:WriteAddressBlock} + T_{5:WriteDiskBlock}
\end{aligned} \qquad (9)
$$

In (9), $T_{1:ReadIFInode}$ is the time spent in reading the inode of the index file, $T_{2:WriteIFInode}$ is the time in writing the corresponding metadata into the index file, $T_{3:ReadDBFInode}$ is the time in reading the inode of the data block file, $T_{4:WriteAddressBlock}$ is the time in writing the file address into the address block of the disk, $T_{5:WriteDiskBlock}$ is the time in writing the file data into the disk.

Equation (9) does not consider the memory cache effect. If the cache effect is taken into account, Equation (9) can be further simplified. For example, the time of a write operation in TFS can be modeled using (10).

$$
\begin{aligned}
T_{ow_{TFS}} \approx T_{2:WriteIFInode} &+ T_{4:WriteAddressBlock} \\
&+ T_{5:WriteDiskBlock}
\end{aligned} \qquad (10)
$$

4.3.2 Modelling write operations for iFlatLFS under the metadata consistency policy

T_{iw} denotes the time of a write operation in an iFlatLFS-based DFS when the strong metadata consistency is applied. Equation (11) represents the model of T_{iw}, where $T'_{1:WriteMetadata}$ is the time spent in writing the file metadata into the disk, and $T'_{2:WriteData}$ is the time in writing the file data into the disk.

$$T_{iw} = T'_{1:WriteMetadata} + T'_{2:WriteData} \qquad (11)$$

By comparing T_{ow} and T_{iw}, Inequality (12) holds, which indicates that a write operation in an iFlatLFS-based DFS with the strong metadata consistency policy is faster than that in a traditional file system without considering the cache effect. Even if the cache effect is taken into account, we can draw the same conclusion. For example, In TFS, by comparing T_{ow_TFS} and T_{iw}, Inequality (13) holds.

$$
\begin{aligned}
T_{iw} &= T'_{1:WriteMetadata} + T'_{2:WriteData} \\
&\approx T_{4:WriteAddressBlock} + T_{5:WriteDiskBlock} \\
&< T_{1:ReadIFInode} + T_{2:WriteIFInode} + T_{3:ReadDBFInode} \\
&\quad + T_{4:WriteAddressBlock} + T_{5:WriteDiskBlock} \\
&= T_{ow}
\end{aligned} \qquad (12)
$$

$$
\begin{aligned}
T_{iw} &= T'_{1:WriteMetadata} + T'_{2:WriteData} \\
&\approx T_{4:WriteAddressBlock} + T_{5:WriteDiskBlock} \\
&< T_{2:WriteIFInode} + T_{4:WriteAddressBlock} + T_{5:WriteDiskBlock} \\
&= T_{ow\_TFS}
\end{aligned} \qquad (13)
$$

When the iFlatLFS-based DFS deploys the weak consistency policy, the metadata consistency operation also needs to be performed for new files that are written. In this aspect, the weak policy is the same as the strong policy. However, the metadata consistency operation is performed periodically in the weak policy, which gives the system the opportunities to optimize the I/O performance. First, if the same data are written several times before next consistency operation, the system only needs to write the data into the disk once. Second, in a consistency

operation, much metadata are written together which may give the system the opportunity to combine small I/O writes into a bigger write and therefore reduce I/O overhead. From the above discussions, we can conclude that the weak consistency policy incurs less overhead than the strong policy.

In summary, according to (8), (12), (13) and the analysis about the overhead of weak consistency, we can conclude that no matter which consistency policy is used, the iFlatLFS-based DFS can always deliver better performance than the DFS based on traditional file systems for both read and write.

4.4 Size of metadata

In the Ext4-based TFS, the size of a data block file is 64 MB. A data block file has a index file, which stores the index data of all small files in the data block. The index data of each small file occupy 20 Bytes. In Ext4, both the index file and the data block file have an inode of 256 Bytes, and an address pointer of 8 Bytes is required for each 1KB data in the data block files. Suppose the total file size is x TB and the average file size is y KB. Then the total number of small files is (x/y) G.

Based on the above discussions, we can use (14), (15) (16) and (17) to calculate the total number of the data block files (denoted by Num_{dbf}) and the total number of the index files (Num_{if}), the size of total inodes ($Size_{inode}$), the size of total index data ($Size_{if}$), and the size of total address data in the data block files ($Size_{address}$).

$$Num_{dbf} = Num_{if} = xTB/64MB = 16K \times x \qquad (14)$$
$$Size_{inode} = 256B \times \left(Num_{if} + Num_{dbf}\right) = 8MB \times x \quad (15)$$
$$Size_{if} = Num_{if} \times \left((64MB/yKB) \times 20B\right)$$
$$= 20GB \times x/y \qquad (16)$$
$$Size_{address} = Num_{dbf} \times \left((64MB/1KB) \times 8B\right)$$
$$= 8GB \times x \qquad (17)$$

Consequently, the total metadata size (denoted by $Size_{metadataofext4}$) can be calculated by (18).
$$Size_{metadataofext4} = Size_{inode} + Size_{if} + Size_{address}$$
$$= 8MB \times x + 20GB \times x/y + 8GB \times x \quad (18)$$

As we have presented in Section 4.1.2, an index item in iFlatLFS occupies 12 Bytes. Then, the total metadata size in the iFlatLFS-based TFS can be calculated by (19).
$$Size_{metadataofiFlatLFS} = (xTB/yKB) \times 12B$$
$$\approx 12GB \times x/y \qquad (19)$$

As can be seen from (18) and (19), the total metadata size in the Ext4-based TFS mainly depends on the total file size and the total number of files, while the total metadata size in the iFlatLFS-based TFS only depends on the total number of files.

Moreover, from (18) and (19), we can calculate the ratio of the size of metadata in iFlatLFS to that in Ext4, i.e.,
$$Size_{metadataofiFlatLFS}/Size_{metadataofext4}$$
$$\approx 12/(20 + 8 \times y) \qquad (20)$$

Equation (20) shows that the metadata size in iFlatLFS is a fraction of that in Ext4. For example, when the average file size is 16KB, $Size_{metadataofiFlatLFS}/Size_{metadataofext4} \approx 8\%$.

Because the ratio of the metadata size to the file data size is so small in iFlatLFS, iFlatLFS is able to support storing a large volume of data. For example, in a data server with 10TB, the same as that of Haystack deployed by Facebook [3], only 7.5GB main memory are needed for caching the metadata.

4.5 Implementation

A prototype of iFlatLFS has been implemented in CentOS re-

lease 5.5 (kernel is 2.6.18-308.8.1.el5.plusxen x86-64). Because Haystack is not open source, we selected the open source project TFS [9] designed by the Alibaba Group [20] as the upper DFS. The Alibaba Group is a top e-business service provider in China. According to Alibaba's report in 2010, TFS has managed about 28.6 billion photos, whose average size is 17.45KB [9].

In this implementation, the disks must be partitioned manually and several configuration parameters must be determined in advance, such as data block size, the involved disk partitions and their sizes, average file size. The average file size can be derived by statistical means or historical data, while others parameters can be determined by the hardware configuration of the data server. The average file size relates to the application scenarios, while the data block size can be determined using (21), where $Size_{DiskPartition}$ is the size of disk partition, which can be determined in the way discussed in Section 4.1.2; $Size_{MetaData}$ is the size of the total metadata of this data block; $Size_{MetaData}$ is calculated by (19); $Size_{ReservedSpace}$ is the size of reserved space, which is set by the administrators empirically (typically 5% of $Size_{DiskPartition}$).
$$Size_{DataBlock} = Size_{DiskPartition} - Size_{ReservedSpace}$$
$$-Size_{MetaData} \qquad (21)$$

The principle of setting $Size_{ReservedSpace}$ is to gain the experience about the level of fragmentation (i.e., total size of the fragmented disk spaces) in the disk partition. If the level of fragmentation is greater than the size of reserved space, it will cause the data to be written into the fragmented disk space when other spaces including the reserved space are full, which will hurt the disk accessing performance. On the contrary, if the reserved space is bigger than the fragmented disk space, the data accessing performance will not be negatively affected, but the disk utilization will become lower. Therefore, setting $Size_{ReservedSpace}$ needs to strike the balance between disk accessing performance and disk utilization.

We extended an existing dedicated formatting tool, named *stfs*, to allocate the disk space for the metadata area and the data area according to these parameters. Finally, the disk partition headers and storage area headers are written into the disk.

The size of the disk partition header is 1 KB, so is the storage area header. Additionally, we use a bitmap to represent the states of all index slots in the metadata area. Each bit in the bitmap represents the state of one index slot. If a bit is "1", then the index slot is occupied by a file. If the bit is "0", the index slot is free. In order to prevent the data failure, iFlatLFS generates a copy of these headers and bitmap, and stores them into the disk. iFlatLFS verifies mutual integrity automatically when the data are loaded into cache.

As shown in Section 4.2, the file access in iFlatLFS consists of two phases: querying physical address and accessing data. Due to space limitation, the detailed outlines of the two phases are included in the supplementary file of this paper.

5 Performance Evaluation

We have evaluated the performance of iFlatLFS and several existing file systems, including Ext4, XFS [18] and ReiserFS [35]. XFS is the file system used in Haystack [3], which is a DFS developed by Facebook to improve the performance of handling photos (relatively small files), while ReiserFS is a file system that can achieve good performance in accessing small

files in local machines.

We have implemented iFlatLFS into TFS [9] because Haystack is not open source. We compared the performance between the iFlatLFS-based TFS and the original implementation of TFS, which is based on Ext4.

In the experiments, we used the production workload traces observed in Facebook [3]. The testing cases in this paper include random read and mix random access. As discussed in [3], [9], the production workload in the real world is typically dominated by read. The experiments used the same ratios of random read to write as set in [3] and the file size to be read or written ranges from 1 KB to 64 KB with the average of 16KB.

5.1 Performance of iFlatLFS

Firstly, we evaluate the performance of iFlatLFS in a typical DFS environment on a Dell PC, which has a 2.33GHz Intel Q8200 processor with 4 cores and 4MB L2 cache, 4GB of memory and a 500GB SATA disk. The partition size in all file systems is 128GB. The CentOS release 5.5 with kernel 2.6.18-308.8.1.el5.plusxen for x86-64 was installed. To support Ext4, XFS and ReiserFS, we also installed the e4fsprogs package of version 1.41.12-2.el5, the xfsprogs package of version 2.9.4-1.el5 and the reiserfs-utils package of version 3.6.19-2.4.1.

Based on the above platform, we used the open source multithreaded disk I/O program named *Randomio* [21] to establish a baseline for the maximum read/write performance (regarded as the optimal performance that a file system can achieve), which is also how the baseline performance is established in reference [3]. We then tested and compared the performance achieved by *Randomio*, iFlatLFS, Ext4, XFS and ReiserFS. In the experiments, the file size ranged from 1 KB to 64 KB.

Since the objective of the experiments is to evaluate how well the file systems (i.e., iFlatLFS, ext, xfs and ReiserFS) work in the context of DFS, the data storage layout in the data servers and the accessing patterns to these stored data were generated in the experiments to mimic the real DFS context. In Ext4 and ReiserFS, 2048 data block files, each with the size of 64 MB, were, while in the iFlatLFS and XFS, a total of 8 data partitions with the size of 16G, each of which corresponds to a data block, were created. Therefore, the entire system has the storage of 128GB. Note that XFS is used in Haystack developed by Facebook for handling photos. Haystack is not released. So we configure XFS to mimic the DFS context only according to the information available in the literature [3].

We then wrote 80GB of small-size data (ranging from 1KB to 64KB) into these data block files in Ext4 and ReiserFS, and also into the data partitions in iFlatLFS and XFS. The experiments were then conducted for both the existing file systems and iFlatLFS in the following 3 steps: 1) selecting a data block randomly, and also selecting a random offset on the data block; 2) reading or writing the data from the offset of the selected data block; 3) repeating step 1 and step 2 for 1000 times. As in TFS [9], the number of I/O operations performed per second (iops) is used as the metric to measure I/O performance, which can be regarded as the throughput of a file system.

Fig. 8a shows the performance in terms of iops when there are only read operations. As can be seen from Fig. 8a, iFlatLFS can significantly outperform other file systems in all cases (by 48% on average in case of Ext4), and the performance achieved by iFlatLFS is very close to the baseline performance obtained

by *randomio*. These results suggest that iFlatLFS is able to achieve near-optimal I/O performance. A closer observation from Fig. 8a also shows that as the file size increases, the performance advantage of iFlatLFS over other file systems decreases. This is because iFlatLFS optimizes I/O performance by reducing the I/O overhead such as the overhead of looking up metadata and data addresses. When the file size increases, the proportion of overhead in the whole duration of an I/O operation decreases, and consequently the advantage of iFlatLFS becomes less prominent.

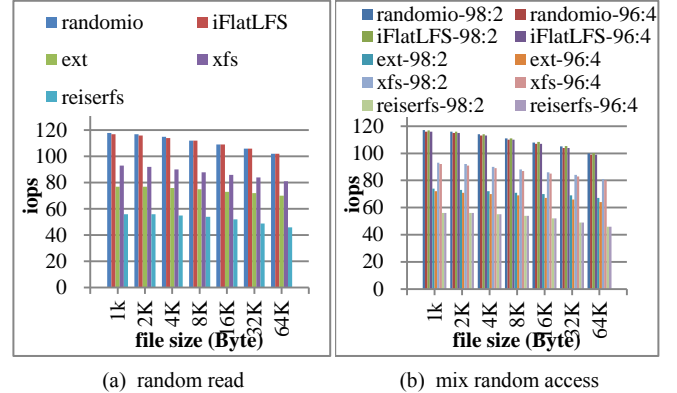(a) random read　　　　(b) mix random access

Fig. 8 Performance of Randomio, iFlatLFS, Ext4, XFS, ReiserFS

Another interesting observation is that Reiserfs obtains the worst performance, even worse than Ext4. The reason for this is explained as follows. Reiserfs can indeed optimize the performance of accessing small files in local computers. However, when Reiserfs is used as the file system on the data servers of a DFS, the files that Reiserfs receives from the upper DFS are the data blocks formed by the combined-block approach. Such a data block is relatively big (e.g., it is 64MB in TFS), which means that the files received and stored by Reiserfs are actually big. ReiserFS is engineered so that it can handle small files efficiently, but does not have good performance when handling big files. This is why Reiserf exihibits poor performance in the figure. This result suggests that although Reiserfs can optimize the performance in accessing small files in local computers, it does not show good performance in the context of DFS. This result once again shows the necessity of our work.

It can be seen from Fig. 10a that XFS used in Haystack delivers the higher performance than Ex4 and ReiserFS. This result is expected because XFS is configured in the experiments in the similar way as it is used in Haystack. However, the performance of XFS is still worse than that of iFlatLFS, which can deliver the near-optimal performance. This is because XFS is still designed following the principles of the traditional file systems, for example, using the hierarchical data management model. However, iFlatLFS is completely re-designed (e.g., using the flat data storage structure) to minimize the overhead for accessing massive small files in DFS. Consequently, the data addressing overhead in XFS is higher.

Fig. 8b shows the performance of iops when there is a mixture of reads and writes. In the legend of Fig. 8b, the suffix of 98:2 represents the random access mixing 98% reads and 2% writes, while the suffix of 96:4 represents the mix of 96% reads and 4% writes. It can be seen from Fig. 8b that in all cases, iFlatLFS can achieve near-optimal performance and significantly outperforms other file systems (by 54% on average in case of Ext4). These results are consistent with those observed in Fig.

8a. Again, similar to the results in Fig. 8a, in Fig. 8b the performance improvement of iFlatLFS over other file systems diminishes as the file size increases.

5.2 Performance of iFlatLFS-based TFS

The performance of TFS based on iFlatLFS and Ext4 file system is evaluated in a cluster with 4 nodes and a Cisco SR2024 24 ports 10/100/1000 gigabit switch. Each node has two 2.13 GHz Intel Xeon E5506 Quad processors, 4GB of memory, 2.5TB of SATA disks, and installs CentOS release 5.5 with the kernel 2.6.18-308.8.1.el5.plusxen for x86-64. In the total 2.5TB of disk space, 0.5TB is used by the operating system, 1TB by the Ext4 file system and 1TB by iFlatLFS.

The experiment was conducted in 3 steps: 1) writing a total 512GB of small files with different sizes; 2) randomly reading 1000 files from these small files written in the first step; 3) mixing read and write operations with the same ratio as in the experiments presented in Section 5.1.

The results are shown in Fig. 9a and 11b. iFlatLFS-based TFS significantly outperforms Ext4-based TFS in both read only case (by about 45% on average) and the read-write case (by about 49% on average). These results once again indicate that iFlatFLS is able to optimize I/O performance.

By comparing with the results in Fig. 8, the performance of iFlatLFS-based (or Ext4-based) TFS is poorer than that of iFlatLFS (or Ext4). This is because accessing data in TFS incurs additional overhead, such as transferring data in the network and accessing metadata server. If a faster network product is used, the additional overhead is expected to be smaller.
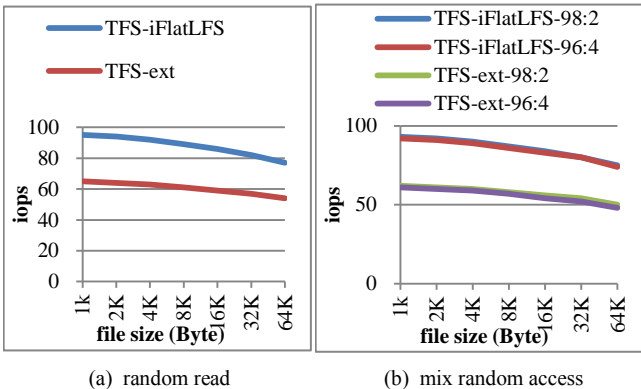

(a) random read (b) mix random access

Fig. 9 Performance of TFS based on iFlatLFS and Ext4

5.3 The impact of metadata consistency policy

In Section 5.1 and 5.2, the strong metadata consistency policy is applied. In this subsection, we present the experiment results for the weak consistency policy and show the impact of the consistency policy on performance.

Fig. 10 plots the performance of iFlatLFS-based TFS with different metadata consistency policies as the ratio of write requests to the total requests increases. In Fig. 10, in the weak consistency policy, the performance of the iFlatLFS-based TFS increases as the proportion of write requests increases. This result can be explained as follows. Under the weak consistency policy, a write operation returns after the data has been written into the system buffer. The file system periodically writes the new data into the disk, which is conducted in the background and may overlap with other file access operations. However, a read operation most likely involves a disk operation, because

they are random reads in the experiments and the cache effect has little positive impact. Therefore, a write operation typically spends less time than a read under the weak consistency policy. Moreover, when the file system write the data into disks periodically, many data are written together, which gives the underlying I/O system the opportunity to improve performance.

It can also be observed that under strong consistency, the performance of the iFlatLFS-based TFS decreases as the proportion of the write operations increases. This is because the new data are written into the disk synchronously under strong consistency. A write operation now needs two disk operations, while a read operation needs only one disk operation.

Fig. 10 also plots the performance of the Ext4-based TFS. It shows that the performance increases slowly as the proportion of writes increases. This can be explained by comparing (3) and (9). $T_{2:WriteIFData}$ in (9) is always smaller than $T_{2:ReadIFData}$ in (3) because a *mmap* write operation returns when the data is put into the memory cache while a *mmap* read will cause a disk I/O operation, if the required index data are not in memory.

Fig.10 also shows that under both policies, the performance of iFlatLFS-based DFS is better than that of traditional DFS in all write-to-read ratios. In the experiments, the performance of traditional DFS has counted in the effect of the cache. This result suggests that iFlatLFS-based DFS can always deliver better performance than traditional DFS. These experimental results are also consistent with the analysis by comparing (9) and (10).
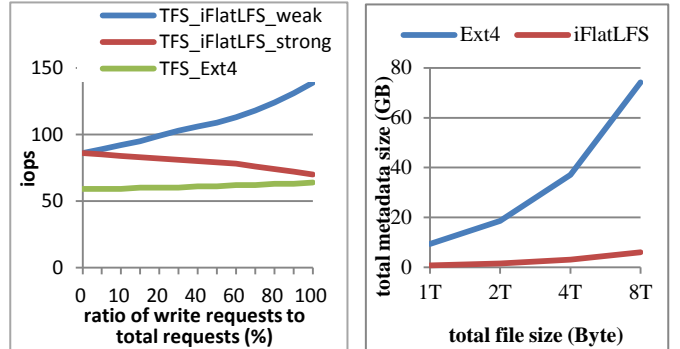

Fig. 10 Performance of TFS based on Ext4 and iFlatLFS with different consistency policies

Fig. 11 The size of metadata in iFlatLFS and Ext 4 under different total file sizes

5.4 Metadata size

Fig. 11 plots the size of metadata in iFlatLFS and in the Ext4 file system as the total file size increases. In the experiments, the file sizes range from 1K to 64K with the average size of 16KB. The data block size in Ext4 is 64MB. In Fig. 11, the metadata size increases at a very modest rate in iFlatLFS, while the metadata size in Ext4 increases much more sharply as the total file size increases. This is because the metadata in iFlatLFS only contain index items, each of which is of fairly small size, while the metadata in Ext4 contain inode data, index data and address data, which have to consume much bigger disk space. Metadata can be seen as the storage overhead of a file system. Therefore, this result indicates that iFlatLFS incurs much less storage overhead than Ext4. In fact, since the size of metadata in iFlatLFS is so small, it enables the entire metadata to be loaded into the main memory, and therefore effectively eliminates the I/O operations involved in retrieving metadata.

Fig.12 plots the size of metadata in iFlatLFS and in the Ext4 file system as the average file size increases from 4KB to 64

KB. In these experiments, the total file size is fixed to be 1 TB. As can be seen from this figure, the total metadata size decreases as the average file size increases in both iFlatLFS and Ext4. This can be explained as follows. When the average file size increases and the total file size is fixed (1TB), the total number of files decreases. Consequently, the total number of index items decreases, since each small file has an index item (which is stored in the index files in Ext4, and stored in the metadata area in iFlatLFS). Thus the total metadata size decreases.

It can also been seen from Fig. 12 that as the average file size increases, the metadata size in Ext4 and iFlatLFS gradually approaches to different figures. The curve of Ext4 approaches to about 8GB, while the curve of iFlatLFS to about 0. This can be explained as follows. In iFlatLFS, the index items are the only metadata. Each file has an index item of 12 Bytes. In theory, when the file size is so big that the whole 1TB space has only one file, the total metadata size is then 12 Bytes, which should be the theoretical lower bound of the total metadata size. In Ext4, however, the metadata includes inode data, index data and address data. An address pointer of 8 Bytes is needed for each 1KB data. Then 1TB data will have 8GB address data, which is fixed no matter the number of files in the disk. On the other hand, the size of index data decreases as the number of files decreases. This is why the metadata size of Ext4 gradually approaches to 8GB in the figure.
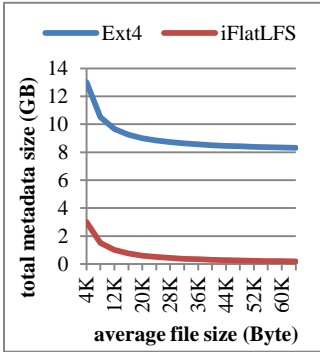

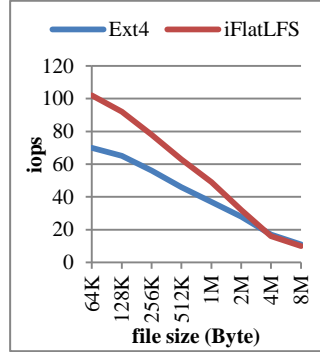Fig. 12 The size of metadata in iFlatLFS and Ext4 under different average file sizes

Fig. 13 Critical point of file size for iFlatLFS

5.5 Critical point of file size

As shown in Fig. 8, the performance advantage of iFlatLFS over Ext4 diminishes as the file size increases. We have also conducted the experiments to identify the critical point of file size, i.e., the file size beyond which iFlatLFS does not outperform Ext4 any more. The experimental results are presented in Fig. 13. The experimental settings are the same as those in Section 5.1. It can be seen that As can be seen from this figure, the performance advantage of iFlatLFS over Ext4 decreases as the file size increases from 64KB to 2MB. iFlatLFS becomes inferior to Ext4 when the file size is 4MB. Besides the reasons discussed in Section 5.1, another reason why the iFlatLFS advantage decreases may be because that the predictive prefetching and caching techniques in the Ext4 file system can lead to substantial performance improvement as the file size increases.

5.6 Comparing different data storage systems

The experiments presented so far compare iFlatLFS with other file systems in accessing small files. This section compares TFS-iFlatLFS with other types of distributed storage systems.

TFS-iFlatLFS is the combined-block-storage DFS. The divided-block-storage DFSes and the NoSQL (Not only SQL) database systems (a type of key-value storage systems) are other two popular data storage systems. In this section, TFS-iFlatLFS (version of 1.4) is compared with HDFS [5] (Hadoop version of 2.4.1, a divided-block-storage DFS) and Cassandra [37] (version of 2.0.9, a key-value storage system) in terms of system throughput (defined as the amount of data that can be accessed by the system in a time unit) over a wide range of data sizes.

The hardware platform and test steps used in this section are the same as those in Section 5.2. The experimental results of random read are plotted in Fig. 14. The results show that iFlatLFS-based TFS does not always have performance advantages over Cassandra and HDFS. TFS-iFlatLFS outperforms Cassandra when the size of the data accessed by each request is bigger than 9.8KB, while it outperforms HDFS when the data size is less than 2.1MB. The reasons for these can be explained as follows. iFlatLFS-based TFS aims to optimize the performance for accessing massive number of seldom modified small files. It does not have the optimization measures present in HDFS or Cassandra for accessing big files or tiny data. Our experimental records show that in iFlatLFS-based TFS, when the data size is small, the iops (IO operations per seconds) value remains almost unchanged and the performance increases almost linearly as the data size increases. But this observation gradually deviates as the data size becomes bigger. This is because as the data size increases, the proportion of the data accessing time in the whole I/O operation duration (i.e., data accessing time plus disk accessing overhead) increases. The value of 2.1M shown in Fig. 14a appears to become a threshold data size, beyond which the data accessing time outweights the disk accessing overhead and consequently, the iops value decreases as the data size increases. Therefore, the increasing rate of throughput slows down as the data size increases. In HDFS, however, when the data size is bigger, the prefetching and write-back techniques, which is absent in iFlatLFS, begins to play an increasingly more important role and counerbalance the loss due to the disk accessing overhead. As the result, the throughput achieved by HDFS increases faster than that by TFS-iFlatLFS. The trend of the throughput curves in Fig. 14b can be explained by the similar reasons.
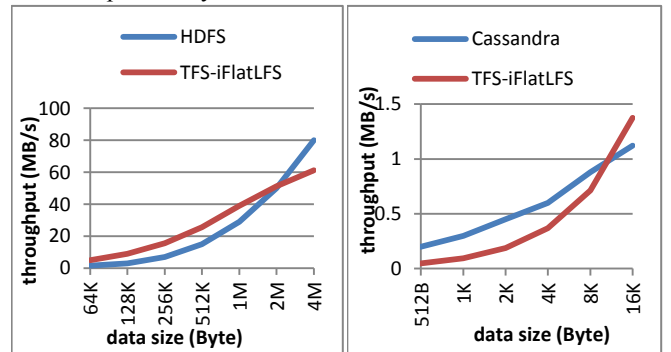
(a) iFlatLFS-based TFS vs. HDFS

(b) iFlatLFS-based TFS vs. Cassandra

Fig. 14 The random read performance comparation among Cassandra, iFlatLFS-based TFS and HDFS

The comparison results for mix random access are plotted in Fig. 15. The results are plotted as columns instead of lines in this figure, because otherwise some curves would overlap each other. The results observed from Fig. 15 are consistent with

those observed in Fig. 14.

These results show that there is a data size range within which TFS-iFlatLFS represents a better storage solution compared with other two storage solutions such as Cassandra and HDFS. These results are reasonable since the divided-block-storage solution is designed for storing big files, while the key-value storage solution is designed for storing the data with very small sizes. These results also suggest that when big files, small files and tiny data all exist, there may not be a storage system that can achieve the best performance for the whole spectrum of data. It would be ideal to integrate these different storage systems as a hybrid data storage system. The final section of this paper tries to open the discussions in this aspect and proposes the possible design of such a hybrid data storage system.
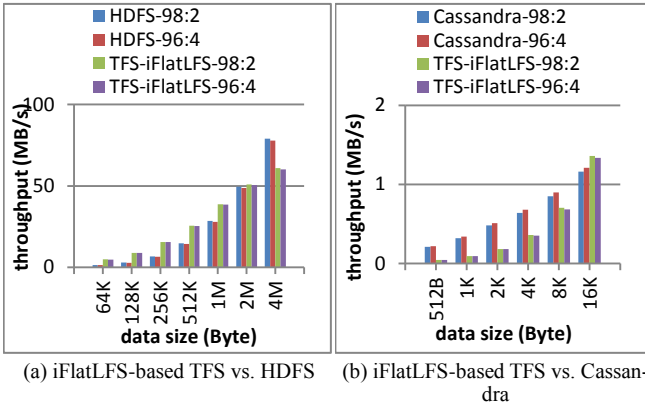

(a) iFlatLFS-based TFS vs. HDFS (b) iFlatLFS-based TFS vs. Cassandra

Fig. 15 The mix random access performance comparation among Cassandra, iFlatLFS-based TFS and HDFS

6 DISCUSSIONS: INTEGRATING DIFFERENT DATA STRORAGE SYSTEMS

At present, there are three types of prevalent large-scale distributed data storage systems: the divided-block-storage DFSes, the combined-block-storage DFSes and the NoSQL (Not only SQL) database systems. The divided-block-storage DFSes, such as GFS and HDFS, are usually used to store big files. But these DFSes can not deliver the ideal performance when handling small files. The main aim of designing combined-block-storage DFSes, such as Haystack and TFS, is to solve the problem of accessing massive numbers of small files efficiently. Furthermore, the NoSQL database systems, such as Dynamo [36], Cassandra [37], MongoDB [38], HBase [39] and BigTable [40], are mainly designed for storing the data of tiny size.

The experimental results presented in Section 5.6 are consistent with the above design objectives. iFlatLFS can improve the performance of accessing massive numbers of small files with the KB-level size in the combined-block-storage DFSes. For the files with MB-level or bigger size, HDFS (the divided-block-storage DFS) can achieve better performance, while for the tiny data with the byte-level size Cassandra (the NoSQL database system) is a better solution. Therefore, it is ideal to integrate different data storage systems that have different ranges of "expertise" and use them to handle the data with different characteristics, such as the size of the data being accessed. We call this a hybrid data storage system.

There are some existing works in the literature [41, 42] to integrate the individual storage Clouds. RACS [41] is a proxy that applies the RAID-like techniques at the cloud storage level.

Namely, RACS stripes user data across multiple providers, and transparently spreads the storage load over many providers. DepSky [42] is designed for improving the availability, integrity and confidentiality of the data which are stored on diverse clouds. DepSky is implemented as a software library in the clients and offers a uniform store interface. The requests of the clients are finally sent to the backend clouds by DepSky.

Although there are some similarities in principles between RACS/DepSky and the hybrid storage system that we want to achieve. RACS/DepSky cannot be used in our scenario. First, the backend of RACS/DepSky is individual storage clouds while the backend of our hybrid storage system is the individual data storage systems. Their interfaces are different. Second, RACS/DepSky mainly focus on maintaining the data availability at cloud level. Therefore they apply the data replication techniques, aiming to avoid vendor lock-in and better tolerate provider outages or failures. Each cloud contains all data set, but only stores a part of all replicas of a data item. However, our hybrid storage system mainly aims to achieve good data accessing performance by making use of the "expertise" of individual data storage systems. A backend data storage system will only store a part of all data set in the hybrid storage system, but store all replicas of a data item.

In this section, a hybrid storage system is proposed to serve the above purpose. This section only aims to show that it is possible to integrate iFlatLFS and other storage approaches, i.e., the combined-block-storage DFSes and the NoSQL database systems, in a hybrid storage system. The detailed implementation of this hybrid storage framework is beyond the focus and scope of this paper. We plan to carry out the implementation work in the future. We also hope the proposed hybrid storage system can open valuable discussions and constitute a basis for further research work in this topic.

6.1 The architecture of the hybrid storage system

The architecture in the hybrid storage system is illustrated in Fig. 16. In this figure, the clients are the same as the client in Fig. 1, i.e., the application server that provide someservices to the users. Different from Fig. 1, the client does not access the DFS directly. Instead, a storage proxy sits between the clients and different backend storage systems (e.g., iFlatLFS-based TFS, HDFS and Cassandra). The storage proxy is responsible for classifying the incoming data accessing requests and dispatching the requests to the suitable backend storage systems.
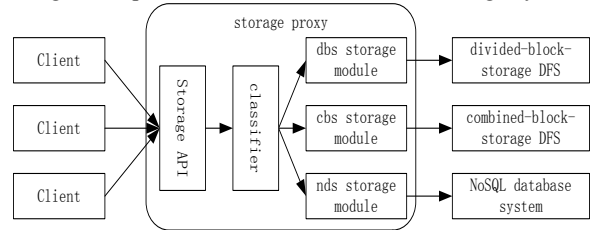

Fig. 16 The architivecutre of the hybrid storage systems

The storage proxy not only classifies the client requests, but also deals with the heterogeneity of the backend data storage systems. The storage proxy is composed of the storage API, the classifier and a number of storage modules. Each storage module acts as an agent for a backend storage system. The backend data storage systems are transparent to the clients. Namely, the storage proxy provides a set of uniform storage APIs that can be

invoked by the clients and the clients do not have to be aware of the heterogeneity of the backend storage systems.

The process of storing the data in the hybrid storage system is as follows. First, the clients invoke the generic storage APIs provided by the storage proxy. Then, the classifier classifies the incoming data and invokes the corresponding storage module to forward the data to the backend data storage system. Finally, the data storage system stores the data into its own data servers.

6.2 Storage APIs and the classifying approach

The storage APIs provided by the storage proxy can include *read*, *write*, *update*, *delete* and so on. Each storage module implements these APIs for the corresponding backend storage system in the following way. A storage module encapsulates the data to be stored into the storage format required by the corresponding backend storage system and then invoke the corresponding interface of the backend data storage system.

Theoretically, the classifier can classify the data using deffirent attributes, such as data type, data freshness, and client attributes, and so on. As shown in the experimental results in Section 5.6, these backend data storage systems manifest better performance in certain ranges of data size. Therefore, in order to improve the overall performance of the hybrid storage system, the storage proxy can classify the data according to their sizes. In practice, we can benchmark the ideal range of data size for each backend storage system deployed in the hybrid storage system in the similar way as that in the experiments presented in Section 5.6. The storage proxy can then classify the data to be stored accordingly.

7 Conclusions and Future Work

When developing efficient distributed file systems (DFS), one of the challenges is to optimize the storage and access of massive numbers of small files for Internet-based applications. Previous work mainly focuses on tackling the problems in traditional files systems, which generate too much metadata and cause low file-access performance on data servers. In this paper, we focus on optimizing the performance of data servers in accessing massive numbers of small files and present a lightweight file system called iFlatLFS. iFlatLFS directly accesses raw disks and adopts a simple metadata scheme and a flat storage architecture to manage massive numbers of small files. New metadata generated by iFlatLFS consume only a fraction of total space used by the original metadata based on traditional file systems. In iFlatLFS, each file access needs only one disk operation except when updating files, which rarely happens. Thus the performance of data servers and the whole DFS can be improved greatly. This paper finally proposes a hybrid storage system to integrate different storage systems, each of which represents a better solution for different ranges of data sizes. The proposal aims to open discussions and constitute a basis for further research work in this topic. Future work is planned towards the following three directions: 1) redesigning the metadata server to improve its performance because in iFlatLFS the metadata server now contains much fewer metadata; 2) extending iFlatLFS so that it has the capability to intelligently cache hotspot data in applications; 3) implementing the hybrid storage architecture proposed in this paper.

REFERENCES

[1] N. Agrawal, W. Bolosky, J. Douceur and J. Lorch. A five-year study of file-system metadata. In Proceedings of the 5th USENIX Conference on File and Storage Technology (FAST'07). , San Jose, CA, USA Feb. 13-16, 2007.

[2] Dutch T. Meyer, William J. Bolosky. A Study of Practical Deduplication. In Proceedings of the 9th USENIX Conference on File and Storage Technology (FAST'11) , San Jose, CA, USA, Feb. 15-17, 2011

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10), Vancouver, Canada, Oct. 2010

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In 19th Symposium on Operating Systems Principles, Lake George, New York, 2003

[5] S. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST'10), Nevada, May 3-7, 2010

[6] Liu Jiang Bing Li Meina Song. THE optimization of HDFS based on small files. The 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT), Oct. 2010

[7] Xuhui Liu, Jizhong Han, et al. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. IEEE Cluster'09, doi:10.1109/CLUSTR.2009.5289196. New Orleans LA, Sep. 2009

[8] Mackey, G. Sehrish, S. Jun Wang. Improving metadata management for small files in HDFS. IEEE Cluster'09, doi:10.1109/CLUSTR.2009.5289133. New Orleans LA, Sep. 2009

[9] Alibaba. TFS Project. http://code.taobao.org/p/tfs/src/

[10] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier. The new Ext4 filesystem: current status and future plans. Proceedings of the Linux Symposium (PDF). Ottawa ON, CA: Red Hat Jan. 15, 2008

[11] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference, pages 1–1, Berkeley, CA, USA, 1997.

[12] Borislav Djordjevic, Valentina Timcenko. Ext4 file system performance analysis in linux environment. Proceedings of the 11th WSEAS international conference on Applied informatics and communications. Wisconsin, USA 2011

[13] Steve D. Pate UNIX Filesystems: Evolution, Design, and Implementation. Wiley. ISBN 0-471-16483-6. 2003

[14] R. Buyya, M. Pathan and A. Vakali. Content Delivery Networks, ISBN 978-3-540-77886-8, Springer, Germany, 2008.

[15] Cade Metz, Google File System II: Dawn of the Multiplying Master Nodes, http://www.theregister.co.uk/2009/08/12/google_file_system_part_deux/, Auguest 12, 2009

[16] Hadoop Archive Guide, http://hadoop.apache.org/mapreduce/docs/r0.21.0/hadoop_archives.html, Aug. 17, 2010

[17] Sequence File, http://wiki.apache.org/hadoop/SequenceFile, Sep. 20, 2009

[18] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In ATEC '96: Proceedings of the 1996 annual

conference on USENIX Annual Technical Conference, pages 1–1, Berkeley, CA, USA, 1996.

[19] FastDFS, http://code.google.com/p/fastdfs/

[20] Alibaba Group, http://www.alibaba.com

[21] Randomio, http:// www.arctic.org/~dean/randomio

[22] Facebook online social network, http://www.facebook.com

[23] Amazon online shopping, http://www.amazon.com

[24] Taobao online shopping, http://www.taobao.com

[25] F. Schmuck et al., GPFS: A shared-disk file system for large computing clusters, in Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST '02, 2002.

[26] I. F. Haddad, PVFS: A parallel virtual file system for linux clusters, Linux J., vol. 2000, Nov. 2000.

[27] Esmet J, Bender M A, Farach-Colton M, et al. The TokuFS streaming file system[C]//Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems, HotStorage. 2012, 12: 14-14.

[28] Vrable M, Savage S, Voelker G M. BlueSky: a cloud-backed file system for the enterprise[C]//Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST'12). USENIX Association, Berkeley, CA, USA. 2012: 19-19.

[29] Hendricks J, Sambasivan R R, Sinnamohideen S, et al. Improving small file performance in object-based storage[R]. CARNEGIE-MELLON UNIV PITTSBURGH PA PARALLEL DATA LABORATORY, 2006.

[30] Zhang B, Zuo Y Y, Zhang Z C. Research and Improvement of the Hot Small File Storage Performance under HDFS[J]. Advanced Materials Research, 2013, 756: 1450-1454.

[31] Zhang Q, Feng D, Wang F. Metadata Performance Optimization in Distributed File System[C]//Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on. IEEE, 2012: 476-481.

[32] K. K. Ramakrishnan , Prabuddha Biswas , Ramakrishna Karedla, Analysis of file I/O traces in commercial computing environments, ACM SIGMETRICS Performance Evaluation Review, v.20 n.1, p.78-90, June 1992.

[33] Wallace G, Douglis F, Qian H, et al. Characteristics of backup workloads in production systems[C]//Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST'12). 2012.

[34] Harter T, Dragga C, Vaughn M, et al. A file is not a file: understanding the I/O behavior of Apple desktop applications[J]. ACM Transactions on Computer Systems (TOCS), 2012, 30(3): 10.

[35] Mason C. Journaling with reisersfs[J]. Linux Journal, 2001, 2001(82es): 3.

[36] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store[C]//ACM SIGOPS Operating Systems Review. ACM, 2007, 41(6): 205-220.

[37] Apache Cassandra project. http://cassandra.apache.org/

[38] Chodorow K. MongoDB: the definitive guide[M]. " O'Reilly Media, Inc.", 2013.

[39] George L. HBase: the definitive guide[M]. " O'Reilly Media, Inc.", 2011.

[40] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 4.

[41] Abu-Libdeh H, Princehouse L, Weatherspoon H. RACS: a case for cloud storage diversity[C]//Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010: 229-240.

[42] Bessani A, Correia M, Quaresma B, et al. DepSky: dependable and secure storage in a cloud-of-clouds[J]. ACM Transactions on Storage (TOS), 2013, 9(4): 12.

[43] Yang XJ, Liao XK, Lu K et al. The TianHe-1A supercomputer: Its hardware and software[J]. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 26(3): 344–351 May 2011. DOI 10.1007/s11390-011-1137-4

[44] Liao X, Xiao L, Yang C, et al. Milkyway-2 supercomputer: system and application[J]. Frontiers of Computer Science, 2014, 8(3): 345-356.

[45] Pang Z, Xie M, Zhang J, et al. The TH Express high performance interconnect networks[J]. Frontiers of Computer Science, 2014, 8(3): 357-366

Songling Fu received the BS degree in the department of electronic science and technology from Harbin Institute of Technology, Harbin, China, in 2001, and received the MS and PhD degree of computer science and technology from National University of Defense Technology, Changsha, China, in 2003 and 2014, respectively. His research interests include parallel and distributed computing, high-performance computer systems, operating systems, cloud computing.
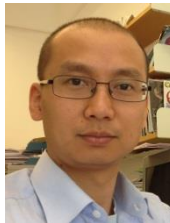
Ligang He received the Bachelors and Masters degrees from the Huazhong University of Science and Technology, Wuhan, China, and received the PhD degree in Computer Science from the University of Warwick, UK. He was a Post-doctoral researcher at the University of Cambridge, UK. In 2006, he joined the Department of Computer Science at the University of Warwick as an Assistant Professor, and then became an Associate Professor. His areas of interest are parallel and distributed computing, high performance Computing.

Chenlin Huang received the BS, MS and PhD degree of computer science and technology from National University of Defense Technology, Changsha, China, in 1998, 2001 and 2005, respectively. His research interests include parallel and distributed computing, high-performance computer systems, operating systems, cloud computing.

Xiangke Liao received the BS degree in the department of computer science and technology from Tsinghua University, Beijing, China, in 1985, and the MS degree of computer science and technology from National University of Defense Technology, Changsha, China, in 1988. He is currently a full professor and the dean of school of computer science, National University of Defense Technology. His research interests include parallel and distributed computing, high-performance computer systems, operating systems, cloud computing, and networked embedded systems. He is a member of the IEEE and the ACM.

Kenli Li received the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2003. He is currently a full professor of computer science and technology at Hunan University and associate director of National Supercomputing Center in Changsha. His major research includes parallel computing, grid and cloud computing, and DNA computing.