The Journal of Supercomputing (2019) 75:7665–7689
https://doi.org/10.1007/s11227-019-02974-8

Effective metadata management in exascale file system

Myung-Hoon Cha[1] · **Sang-Min Lee**[1] · **Hong-Yeon Kim**[1] · **Young-Kyun Kim**[1]

Published online: 22 August 2019

Abstract
This paper presents an effective method of managing metadata in exascale file systems. In order to store exponentially growing numbers of files, numerous methods for distributing and managing metadata have been suggested and developed. However, these methods have not provided an appropriate solution for managing a very large amount of metadata because they do not overcome two significant challenges in exascale file systems: (1) nonlinear performance scalability and (2) performance degradation over time. We propose an effective metadata management model and high-performance metadata management system that not only overcome these limitations but also provide a foundation for managing exascale metadata in a distributed file system. The resulting implementation of our metadata management system is the core of EEFS, an exascale distributed file system by the Electronics and Telecommunications Research Institute. The evaluation results show that the critical challenges of existing metadata management technologies are overcome and particularly that the performance is not degraded even when the amount of accumulated metadata increases with time.

Keywords Metadata management · Exascale · Storage

✉ Myung-Hoon Cha
mhcha@etri.re.kr

Sang-Min Lee
sangmin2@etri.re.kr

Hong-Yeon Kim
kimhy@etri.re.kr

Young-Kyun Kim
kimyoung@etri.re.kr

[1] High Performance Computing Research Group, Infra/Core Software Technology Research Division, Electronics and Telecommunications Research Institute, Daejeon, Korea

1 Introduction

Petascale storage systems have been actively used for supercomputing, cloud storage and big-data analysis. However, the amount of data generated is currently expanding to the exascale, resulting in various technical issues, such as low scalable metadata generation performance. Managing exascale data requires breakthrough improvements in the capacity and performance capabilities of existing petascale-distributed file systems by a factor of 100 or more, but several significant challenges have not been overcome. These include nonlinear performance scalability and performance degradation over time. As a result, instead of directly managing exascale data, several petascale storage systems are built independently and run without integration. Given the considerable amount of data to be generated in the future IoT devices, it is essential to develop a highly scalable and high-performance file system that supports exascale storage space and consistent performance without degradation over time.

Thus far, exascale file systems have not solved the critical issues in metadata distribution and management, such as linear scalability of the metadata processing performance. Modern distributed file systems, as exemplified by HDFS [1] and Lustre [2], which have been applied successfully to both business and industry, use metadata management techniques based on a single metadata server. Because these file systems use only one metadata server, the storage capacity of metadata depends on the single metadata server's resource limits, i.e., the disk space or the amount of RAM.

In one study [3], a metadata storage scalability problem is illustrated using an HDFS-installed metadata server which stores metadata of 100 million files using 60 GB of RAM. The HDFS constraints that all metadata is loaded into the RAM of the metadata server force enough RAM capacity to accommodate all metadata. Because, however, the RAM size cannot be increased infinitely, the number of files that can be managed by the metadata server is very limited.

The greater problem arising due to the use of a single metadata server is the limit of metadata performance scalability, which relates to whether the performance of metadata operations, such as opening a file to process its metadata before I/O operations, is linearly scalable. In a typical distributed file system with one metadata server and many data servers, metadata operations cause a bottleneck [4] because they are processed by only one metadata server, and an increase in the number of data servers in this case has no effect on the performance. Thus, the number of file I/O operations that can be processed per unit of time is also not linearly scalable.

As mentioned above, regardless of how much the performance of the single metadata server of modern distributed file systems is improved, the metadata server does not have the potential to deliver both higher performance and a large enough capacity appropriate for exascale metadata management owing to its structural limitations. However, in order to overcome this issue, research on metadata distribution and management of distributed file systems with many metadata servers is currently extensive, and a variety of technical problems have been

discussed. For example, against all expectations, even if the number of metadata servers increases, the performance does not increase linearly; rather, it deteriorates as the metadata for files accumulates. In addition, the complexity caused by the increased number of metadata servers leads to instability in distributed file systems.

The contributions of this paper are threefold:

1. We propose a distributed directory-based metadata clustering method that provides not only a very good locality of reference but also has excellent potential to realize balanced utilization of all metadata servers.
2. A high-performance metadata management system that supports fast transactions is presented. This is a metadata-specific management system for quickly storing and retrieving inodes and namespaces. It performs simple, space-efficient metadata allocation and memory management, and supports high-speed inode fetch operations.
3. We address the performance degradation issue associated with metadata server clusters, which arises when the number of metadata servers in use increases and metadata accumulates. As a solution of this issue, we propose a performance degradation prevention method based on the efficient memory-buffer management of metadata servers.

The composition of this paper is as follows. Metadata server cluster technologies in large-scale distributed file systems are analyzed in Sect. 2. Our directory-based metadata clustering method and the implementation details of the high-performance metadata management system is presented, and our solutions regarding how to prevent the performance degradation in a metadata server cluster are described in Sect. 3. We evaluate the performance of the proposed method in Sect. 4 and then conclude this paper in Sect. 5.

2 Related works

Instead of centrally managing metadata in a single server in order to store exascale data such as social media service data of the type generated worldwide at every moment, changing the architecture of distributed file systems such that metadata is distributed and managed in many metadata servers is inevitable. Methods for metadata distribution and management are divided into three categories: First, canonical types based on subtree partitioning or hashing exist. Ceph [5, 6] and Gluster [7] are archetypes to which this technique is applied. Second, as a more practical approach, there are workload-specific, dedicated distributed file systems that simply restrict the types of metadata operations to lower the complexity of the system. Hence, they operate effectively with a particular category of operations. Haystack [8], f4 [9] and TAO [10] are representative examples of distributed file systems specific to Facebook workloads. Third, recent attempts have been made to use a distributed DBMS as an engine for managing metadata [11, 12]. Through the use of a distributed

DBMS, located outside of the distributed file system, the complexity of implementation is reduced but there are limitations such that both the functions and performance of distributed file systems depend on the interface and performance of the underlying distributed DBMS. In addition, there are various results based on modifications of other metadata management structures [13–17]. These include attempts to reduce metadata storage by optimizing the metadata management scheme [16].

In this section, the current status and limitations of each of these three categories of methods are discussed.

2.1 Canonical management of metadata

The canonical methods of distributing and managing metadata are further divided into two categories: subtree partitioning and hashing. In subtree partitioning, metadata is distributed in a unit known as a subtree which has large granularity, with the aim of providing great locality of reference. For instance, if there is a need to check the permissions of each component of a path while traversing it, the costs of these operations are offset by access patterns in which those files in the same directory continue to be used. However, future performance depends on how the namespace is initially partitioned, and the load distribution among metadata servers becomes increasingly uneven over time [18–20].

There are also modifications [18] of subtree partitioning that relocate partitioned subtrees dynamically according to the load variation and move the corresponding metadata. With such a method, it is very difficult to decide when to relocate metadata and how to move metadata efficiently.

Hashing distributes metadata to metadata servers according to the hash function, thereby intending to have an evenly distributed metadata. However, because the location of the metadata is determined by the hash function, it is unrelated to the directory hierarchy. As a result, the locality of reference is not preserved. In addition, when a metadata server cluster changes, the output range of the hash function also changes. This change causes the subsequent migration of metadata, which places a considerable burden on the long-term management of the metadata server cluster [18–20]. The difficulty of changing the metadata server cluster is illustrated by Gluster, a leading distributed file system based on hashing. It was implemented to send broadcast lookups to all servers every time a file is created. This is done to avoid false negatives arising from changes of its cluster.

Managing metadata in a canonical way introduces critical problems such as limited file creation performance and instability in operations. In this paper, we propose a metadata management system which resolves the constraints of existing distributed file systems.

2.2 Workload-specific management of metadata

In the development of distributed file systems, there are major research trends that address the issue with practical approaches such as simplifying the types of metadata operations and reducing the complexity of implementing the system, with the

aim of avoiding the difficulties of large-scale metadata management by sacrificing POSIX compliance. Workload-specific distributed file systems developed to accept every moment of social media service traffic, such as Facebook's Haystack, f4, TAO and LinkedIn's Ambry [21], represent such research trends. These systems focus on simple workloads with no updates, group large blobs of immutable data into partitions or volumes and provide a minimal API.

Haystack is used as Facebook's photograph storage system, which relieves the system complexity by supporting only limited operations, i.e., read, write and delete. Haystack focuses on storing data that is read many times without being modified once it is saved. By keeping multiple photographs in a single file and storing them, Haystack basically manages very large files. The cost of reading metadata from disks is reduced by allowing the metadata lookup to be performed in memory [8].

As Facebook grows, its storage system has evolved into a hybrid type where hot data is processed by Haystack and warm data is stored in dedicated storage called f4. Data that is above the threshold is migrated from Haystack to f4, which also stores blobs but uses erasure coding instead of triple replication as used in Haystack [9].

In the overall architecture of Facebook, while the large immutable collections of data, called blobs, are stored in Haystack and f4, the handle to such data is stored in a separate graph store called TAO. Facebook initially stores the social graph in MySQL and uses Memcache [22] as its cache, but Memcache is soon replaced by a read-optimized graph data store which is a direct implementation of a graph-aware cache [10].

LinkedIn's Ambry, another system for storing blobs, supports only three types of operations: put, get and delete. Because a very large blob can cause a load imbalance, such a blob is split and managed in chunk units. Special metadata blobs are used to manage these chunks [21].

As illustrated above, there are many constraints that must be resolved when providing both POSIX semantics and distributed transactions, but they can be relaxed by supporting a limited set of operations and only a specific workload when developing large-scale distributed file systems. However, these specialized distributed file systems are not suitable when there is a change in the workload or when new applications are to be deployed. In this paper, we propose methods that fully support POSIX semantics and that are appropriate for clustering metadata servers in a large distributed file system, overcoming all of the aforementioned disadvantages.

2.3 Other variations on metadata management

Various methods of metadata management schemes have also been studied. First, the use of distributed DBMS as an engine for managing metadata is increasing [11, 23, 24]. The StoreAll [12] file system, implemented for the purpose of managing metadata, is an application which runs on the LazyBase [25] distributed DBMS. Because one characteristic of LazyBase is to collect and upload updated data in batches, this asynchronous update mechanism can give rise to inconsistent reads of old data values. Thus, scanning for updates that have not yet been applied is required to read the most recent data values [25]. In addition, the greater the freshness of the

data, the higher the processing cost. For this reason, the use of this system is limited to specialized applications rather than to general forms.

Second, there is a method which reduces the amount of metadata by sharing the metadata of multiple files via a deduplication method [16]. However, even if the amount of metadata is reduced, there still remains to be solved the fundamental problem of having to use a large number of metadata servers, because a single metadata server cannot overcome exascale memory constraints.

Third, to avoid distributed transactions, all relevant metadata can be moved and processed on a specific metadata server before the transaction processing step [17]. Whenever, however, a metadata operation is performed, the high cost of migrating metadata to a single server and the degraded concurrency issues that arise during the process of coordinating such migrations must be addressed.

Fourth, there have been attempts to reduce the cost of the path name resolution by using a hash-based distribution of metadata while also copying the namespace to all metadata servers [13]. In addition, there have also been techniques proposed to manage the access control list (ACL) for each instance of metadata instead of the namespace, although metadata is distributed based on hashing [15]. These approaches aim both to provide a uniform distribution of metadata and to reduce the cost of the pathname resolution, but a serious burden arises when either updating the replicated namespace or managing ACLs whenever the namespace is changed. Furthermore, there remains a subsequent processing burden when the metadata server configuration is changed in hashing-based metadata server cluster.

Fifth, instead of storing the location information of fixed-size pieces of a file, referred to as chunks, permanently in the metadata server, the chunk location information can be queried by all of the chunk storage servers and the acquired information can then be stored in the memory of the metadata server when the server initially starts [14]. This is suggested simply to manage the synchronization between metadata servers and chunk storage servers so that they do not need to be synchronized with each other. As a result, there is no need to manage and implement the entire system via a difficult process for the purpose of processing various synchronization cases that occur frequently, such as resuming a failed storage server in a system composed of a large number of storage servers. Nevertheless, this approach is linked to the problem of no longer being able to extend the scalability of a single metadata server.

3 Fast and reliable metadata management system

When managing exascale metadata, it is very challenging to solve the linear scalability issues of the storage space and performance, as well as the problem of performance degradation over time. As a solution to these problems, we propose (1) a model that provides good locality of reference while keeping metadata servers in balance, (2) a metadata management system that efficiently supports transactions as a base for the fast execution of the model and finally (3) a method for solving a sudden performance drop due to interference among metadata servers in a metadata server cluster on which the metadata management system runs.

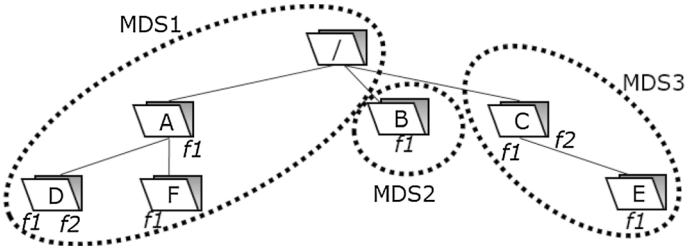
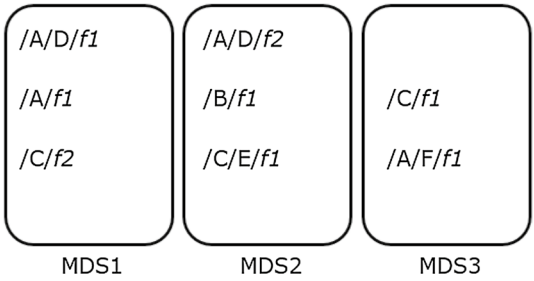


Fig. 1 Subtree-based metadata distribution

Fig. 2 Hashing-based metadata distribution



3.1 Effective metadata management based on the directory distribution

As shown in Fig. 1, when metadata is distributed and managed on a per-subtree basis, it is advantageous to maintain the locality of reference because the unit being distributed is very large while the metadata is not evenly distributed across the metadata servers. If therefore the subtree managed by a specific metadata server grows, the performance of such a metadata server can detrimentally affect the performance of the entire cluster. On the other hand, hashing-based management evenly distributes metadata, as shown in Fig. 2, but it is difficult to maintain the locality of reference. In addition, when a new metadata server is added or an existing server is removed, a significant amount of metadata movement occurs during a subsequent process of handling the modification to the hash function.

In this paper, we propose a directory-based distribution as a means of enabling an even distribution while maintaining sufficiently good locality of reference. Figure 3 illustrates the concept of the directory-based distribution, which uses a directory as the metadata distribution unit. By making the degree of metadata distribution much finer than that of a subtree, it becomes possible to maintain the balance among metadata servers relatively evenly and, therefore, the metadata server cluster can continue to operate in a balanced manner. Such a balance among metadata servers is a property that cannot be achieved with the subtree approach. Furthermore, though a directory-based distribution can have somewhat less locality than its subtree-based counterpart, it still preserves relatively large-scale locality such that a high level of locality of reference, which can never be provided by a hashing-based method, is guaranteed.

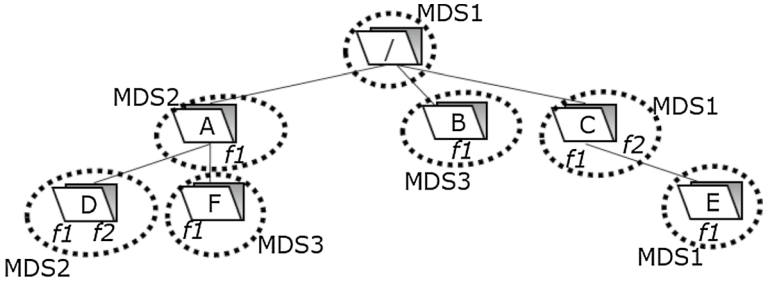


Fig. 3 Directory-based metadata distribution

The directory-based distribution model consists of (1) clients that request metadata operations, (2) metadata servers that receive and process metadata operations, (3) a directory placement algorithm that uniformly distributes directories across metadata servers and (4) metadata processing algorithms that obey POSIX semantics while recognizing the directory-based distribution and placement.

The client sends metadata servers a request to create, query, modify or delete metadata. Metadata is distributed and stored across metadata servers in accordance with the directory-based distribution model. A new directory is created in a suitable metadata server by the directory placement algorithm that chooses the best metadata server to be used. The metadata server that receives the client's request processes metadata operation considering the directory-based distribution semantics. Figure 4 shows a representative architecture using this directory-based distribution model.

For the purpose of laying the foundation of a directory-based distribution, each metadata server provides both inode storage and multi-purpose storage, as shown in Fig. 5. When a directory is placed on a specific metadata server, all of the inodes for files or subdirectories that directly belong to the directory, as well as the inode representing the directory itself, are placed on that metadata server. In particular, each directory entry for directory also contains 'metadata server address' information which indicates the location of the metadata server that owns the directory.

For example, if a directory A is determined to be created on the metadata server 2 (MDS2), the creation of A and its children belonging to A, such as a file f1, proceed as follows. The inode of the directory A is made by allocating one inode resource

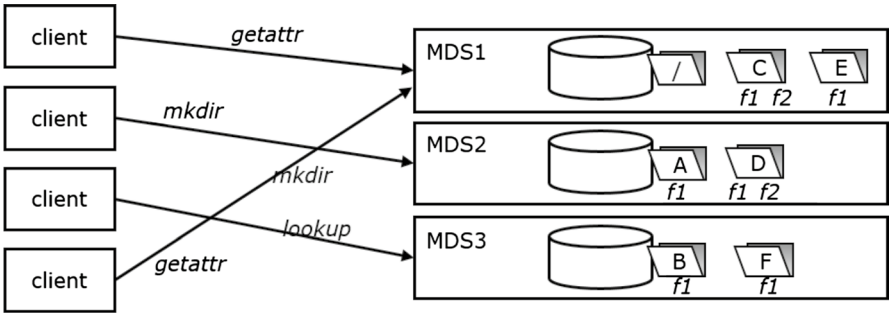


Fig. 4 Architecture of a metadata server cluster using the directory-based distribution model

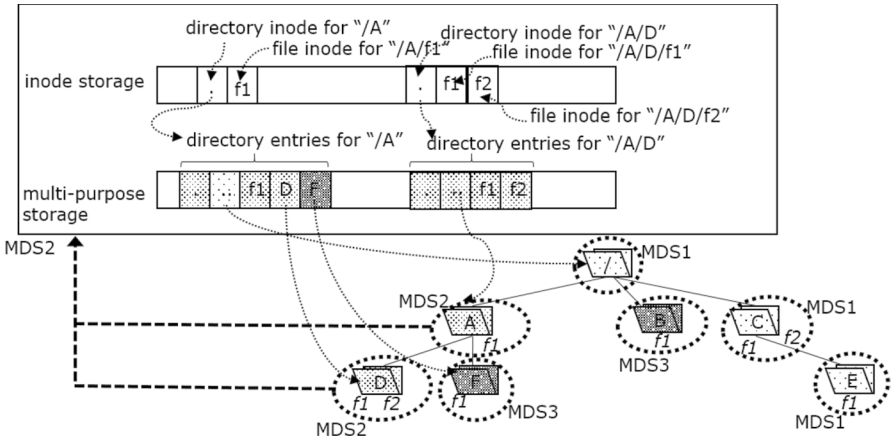


Fig. 5 Directory-based metadata distribution structure

from the inode storage of MDS2. The multi-purpose storage then stores information about '.' and '..' to represent the directory entries for A. If you want to create a file f1 as a child of the directory A, the inode of the file 1 is made by allocating an inode resource from the inode storage of MDS2, which is the same metadata server as the parent directory A, and the entry for f1 is added to the multi-purpose storage where the entries for the directory A are already stored. The results of this process can be seen in Fig. 5. Section 3.2 contains more information on inode storage and multi-purpose storage.

In performing metadata operations that recognize a directory-based distribution model, most metadata operations, e.g., file search and creation operations, are performed as a single transaction on a specific single metadata server because files and symbolic links are stored and managed on the same metadata server that contains their parent directory. However, it is also possible to have metadata operations whose transactions are processed across multiple metadata servers because, in this case, the parent directory may use a storage different from that of the child directory. Most metadata operations that recognize the directory distribution are processed by transactions with one or two metadata server accesses, but there are also rare cases in which some metadata operations require four or more metadata server accesses. Table 1 shows the classification according to the access count of metadata servers when processing metadata operations as transactions.

In *Type*-(*A*) of Table 1, metadata operations are processed as a single transaction, but *Type*-(*B*) and *Type*-(*C*) operations require distributed transactions. Considering an actual workload pattern [20] that generates a large number of file I/O operations, it is very rare for metadata operations to be processed as distributed transactions because *Type*-(*A*) metadata operations such as *getfilelayout*, *getattr*, *setattr* and *creat* account for 99% of all metadata operations. As a result, with so much metadata being managed, running a metadata processing protocol that minimizes distributed transactions and handles a directory-based distribution makes the predicted disadvantages of lower locality than that of the subtree much less meaningful.

Table 1 Categorizing metadata operations based on the total number of metadata servers accessed

Type	Metadata operation type	Total number of metadata servers accessed
Type-(*A*)	creat, unlink, setattr, getattr, lookup, readlink, symlink, getfilelayout	1
Type-(*B*)	mkdir, rmdir, hardlink, lookup (remote dir)	2
Type-(*C*)	rename	≥4

Algorithm 1 is the directory creation algorithm that performs the directory-based distribution and placement operations.

Algorithm 1 Make_directory(P, N)

Input: P – the identification number of parent inode at which the new directory will be made
N – the name of the directory to be made

Begin Procedure

```
1:  static int idx = 0;
2:  int target_mds, i;
3:  If  MDSLIST.total_number > 0  Then
                              /* MDSLIST: list of all metadata servers */
4:    i = idx++ % MDSLIST.total_number;
5:    target_mds = MDSLIST.mds[i].mds_ipaddress;
6:  Else If
7:    target_mds = 0;
8:  End If
9:      /* local mds: indicates the metadata server itself performing the current
              Make_directory(P, N) operation   */
10: If the local mds is not target_mds Then
11:       send RPC call for making new directory inode in P directory
              to target_mds;
12:       I = receive the number of newly created inode at remote mds;
13: Else If
14:       Begin_transaction();
15:       If the local mds is not target_mds Then
16:             make new directory entry having name N linking new inode
                  number I on local mds;
17:       Else If
18:             make new directory inode on the local mds;
19:             I = get the number of newly created inode at local mds;
20:             make new directory entry having name N linking new inode
                  number I on local mds;
21:       End If
22:       End_transaction();
23: End If
```

End Procedure

In Algorithms 1, lines 3–8 present the process of selecting the metadata server to be used as the target for a new directory: With a list of all metadata servers prepared, the next metadata server which is placed after the metadata server in which the most recent directory is created in the traversal order of the list is used as the target where a new directory will be created. If a directory is created on the very same metadata server that initially received the directory creation request, the new directory's inode and directory entries are processed as a single transaction and created on that metadata server. However, if a directory is created on a remote metadata server other than that which originally received the directory creation request, the local metadata server on which the first request arrived sends a second request to create an inode for the new directory to a remote metadata server, and if the second request succeeds, the local metadata server creates a directory entry with which the newly created directory's inode is connected.

In a situation with a large number of metadata servers running, a new metadata server can be added to the cluster of metadata servers and existing metadata servers can be removed. In the hashing-based model, this situation has a significantly negative effect, such as the migration of already deployed metadata, whereas the directory-based distribution model does not have this problem.

Algorithm 2 shows the procedure used to obtain the metadata of a file. This algorithm uses the inode identification number to find the target metadata server in the list of metadata servers and then obtains the desired metadata from the target metadata server. Due to a lack of space, the details of other algorithms are omitted.

Algorithm 2 Getattr (I)

Input: I – the identification number of inode to be retrieved

Begin Procedure

1: *Begin_transaction();*
2: *extract the indexing number from the inode identification number;*
3: *retrieve inode directly from the inode storage using the indexing number;*
4: *End_transaction();*

End Procedure

The following effects can be realized by using the directory-based distribution model: (1) The most frequently used metadata operations, such as file retrieval, file creation and file deletion, are processed by a single metadata server. (2) Metadata operations such as directory browsing, directory creation and directory deletion involve at most two metadata servers. (3) The directory-based metadata distribution enables metadata to be evenly stored across multiple metadata servers. (4) Even though the metadata server cluster changes, expensive post-processing steps such as metadata re-migration are not required.

3.2 High-performance metadata management system with transaction support

In this paper, we propose a high-performance metadata management system whose special feature is that it processes transactions very rapidly to support the efficient management of metadata. This system's architecture consists of (1) inode storage, (2) multi-purpose storage and (3) log storage, as shown in Fig. 6.

Inode storage is a table that stores file inodes and directory inodes, and each inode is represented by a fixed-size record. The position of a record in the table serves as the identifier of the record, and this position is used as the inode identifier. Inode storage, therefore, can store inodes without indexes and inodes can be retrieved instantly. Because, intrinsically, inode storage is a big array that stores inodes, where each element of the array is one inode and its size is 128 bytes, a specific inode can be retrieved immediately with the inode number as a key, which has O(1) complexity.

Multi-purpose storage can store various types of data structures. It is mainly used as space for storing the contents of a directory. In this case, pairs of <*inode identifier, name*> of child entries belonging to a specific directory are collected and stored. The space of multi-purpose storage is managed in fixed-size extent units, and the extent size is relatively larger than that of the record in inode storage. Directory contents are stored initially in one extent, and if the extent space then becomes insufficient, a new extent is allocated and used. The contents of the extent are managed as B + trees. Because, in essence, multi-purpose storage is also a big array that stores extents, where each element of the array is one extent with a size of 4 KB, a specific extent can be retrieved immediately with the extent number as a key, which has O(1) complexity too. In summary, as the retrieval complexity at the level of a single metadata server consisting of inode storage and multi-purpose storage is O(1) and directories are balanced across metadata servers, each metadata is chosen and run evenly on average.

Log storage is a write-ahead log for transaction processing which consists of a log flush thread and two log buffers, as shown in Fig. 7. One log buffer is used as the source of the flushing operation, which is performed by the log flush thread, and the other log buffer accepts the logging contents that are appended at that moment. When the flushing of a log buffer is completed, the role of the log buffer is alternately switched, and a group commit especially is provided to increase the flushing efficiency.

The metadata management system runs a plurality of threads dedicated to processing metadata operations, and it provides strict two-phase locking as a means of concurrency control. In addition, as a buffer management policy, NO-STEAL/NO-FORCE is supported.

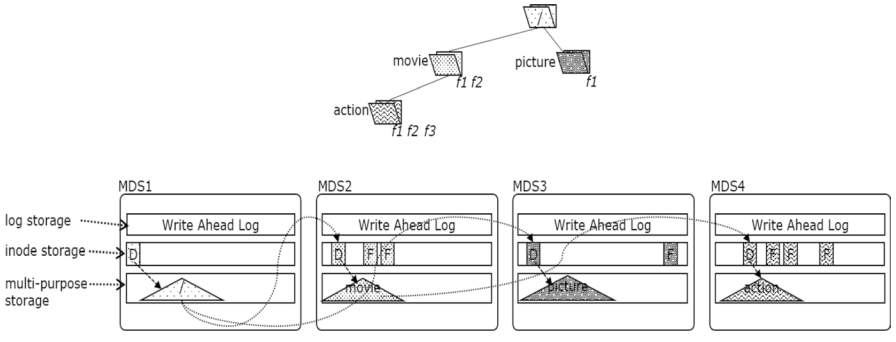


Fig. 6 Architecture of the metadata management system

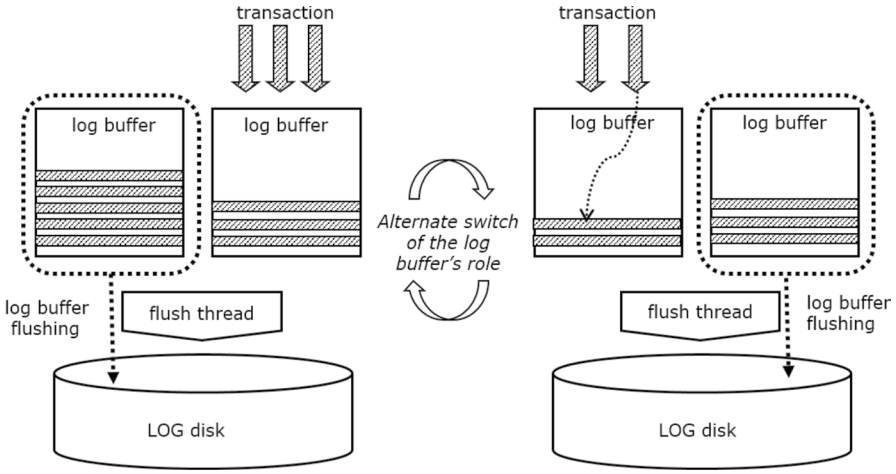


Fig. 7 Architecture of log storage

There are several advantages of operating a dedicated metadata management system for each metadata server. First, the caching efficiency of each metadata server's memory is maximized because the number of inodes that can be cached in the metadata server's memory is maximized by managing one inode as a very small unit of 128 bytes, resulting in metadata information being retrieved more quickly compared to that when using other file systems.

Second, the amount of metadata that can be stored on each metadata server is also maximized, resulting in the minimization of the number of metadata servers.

For example, Fig. 8a illustrates an inefficient use of space in a typical distributed file system where only one file metadata is stored in a 4 KB page. However, as shown in Fig. 8b, the metadata management system introduced here can contain information pertaining to 32 inodes in a single 4 KB page. Suppose a 4 TB hard disk is mounted per metadata server; at least 1125 metadata servers are required to store 1 trillion files under the mechanism shown in Fig. 8a, but at least 32 metadata servers are needed when using the case shown in Fig. 8b. Such an efficient caching and storage mechanism for large-scale metadata is critical when building an exascale file system.

3.3 Resolving interference among metadata servers

The use of the directory-based distribution mechanism increases the performance of metadata operations in proportion to the number of metadata servers. However, if hard disks are used as the storage media of metadata servers, the performance of the entire metadata server cluster will fluctuate heavily as the number of metadata servers increases. For example, if a metadata server cluster is continuously requested to create metadata, the initial performance of the metadata creation is not maintained, and the performance soon fluctuates up and down. As the number of metadata

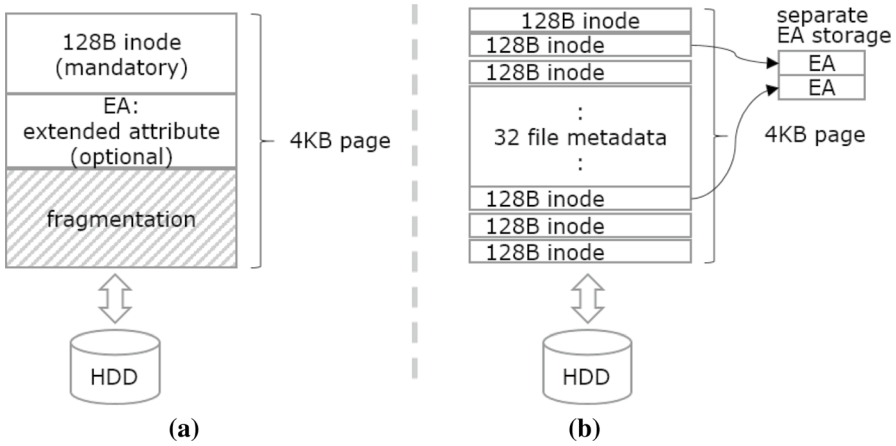


Fig. 8 Metadata storage layout based on a 4 KB page versus a 128B record

servers increases, this phenomenon worsens. In this paper, we refer to this as the problem of performance interference among metadata servers.

As a background knowledge before explaining the performance interference problem, the processing of metadata operations as transactions on metadata servers will be described. While metadata operations essentially create metadata or change them in inode and multi-purpose storage, the changes that are made to process these metadata operations as transactions are initially written into log storage and then reflected in inode storage and multi-purpose storage. However, in order to process metadata operations at a high speed, the contents to be reflected in inode storage and multi-purpose storage are not immediately written to the disk but are reflected first in the memory buffer managed by the operating system. This is done to collect the results of multiple operations to the greatest extent possible so as to perform high-speed processing at the same time, without dispersing expensive disk writes, and to maintain the consistency of the metadata. Dirty data reflected in the memory buffer is flushed to disks at a fixed cycle by the flushing thread of the operating system, such as *pdflush* of Linux, or when a certain memory condition is met.

Figure 9 illustrates a situation in which a large-scale flush operation occurs while log write operations continue to be done to process each metadata operation as a transaction. In a of Fig. 9, because a large flush does not occur in the operating system of the metadata server, the log write operations are not disturbed; as a result, the metadata operations are processed at a high speed. In b of Fig. 9, on the other hand, all dirty data in the memory buffer, corresponding to inode and multi-purpose storage, are flushed to disks at one time such that the log write operations slow down

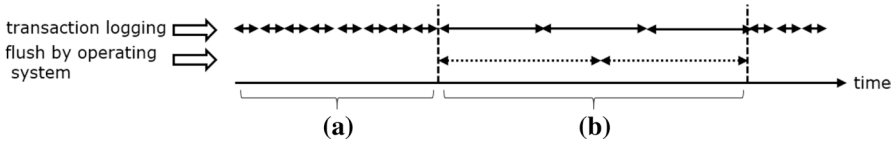


Fig. 9 Metadata disk usage over time when the operating system uses a flush mechanism

due to this bottleneck. In this paper, b of Fig. 9 is termed the processing delay zone. Because a delay during log write operations means that the corresponding metadata operations are also delayed, running a disk-based metadata server in which a large amount of metadata is written causes frequent delays on the metadata server.

The effect of the *pdflush*-like flushing mechanism by an individual metadata server is a critical issue that cannot be ignored in clustering metadata servers. The higher the number of metadata servers, the more interference of performance the metadata servers will encounter due to the reasons mentioned above. Thus, in an environment where a very-large-scale metadata processing is required, such as at the exascale, effective techniques are needed to minimize the performance interference caused by multiple metadata servers running simultaneously. Detailed tests and analyses of the causes of these phenomena are described in Sect. 4.

In this section, we introduce a new memory-buffer-manager algorithm as a solution to the performance interference problem among metadata servers. This algorithm efficiently manages the temporary memory buffer before storing metadata in inode and multi-purpose storage, and it is run on each metadata server. The memory buffer is an area where resources of inode and multi-purpose storage are kept before being flushed to disk. One memory buffer is divided into a plurality of fixed-size blocks, and multiple resources like inodes are kept temporarily in a block. The use of the memory-buffer manager is intended to prevent the performance degradation of the entire metadata server cluster by using each memory-buffer manager's unique behavior based on Algorithms 3 and 4.

Each memory-buffer manager periodically performs the *Manage_Block* function, as described in Algorithm 3. The *Manage_Block* function manages a block which is an in-memory management unit for metadata storage, i.e., inode and multi-purpose storage. It is a key algorithm in the memory-buffer manager mechanism, performing two key functions as described in Algorithm 4: (1) The blocks to be written are grouped into clustered block groups in consideration of minimizing the block writing cost. (2) These clustered block groups are written in short intervals.

Algorithm 3 Memory_Buffer_Manager(*L, fd, cd, m, b, fi, ci*)

Input: *L* – the buffer with the active blocks of inode and multi-purpose storage
fd – the flushing duration of a block
cd – the caching duration of a block
m – the max number of writing blocks
b – the max boundary of writing blocks
fi – *flushing* interval
ci – checking interval

Begin Procedure
1: **Repeat**
2: ***Manage_Block****(L, fd, m, b, fi, cd);*
3: *Sleep for interval ci;*
4: **Until** *Memory_Buffer_Manager receives stop signal;*
End Procedure

Algorithm 4 Manage_Block(*L, fd, m, b, fi, cd*)

Input: *L* – the buffer with the active blocks of inode and multi-purpose storage
fd – the flushing duration of a block
m – the max number of writing blocks
b – the max boundary of writing blocks
fi – *flushing* interval
cd – *the caching duration of a block*

Begin Procedure

```
1:  N is an empty list of blocks
2:  For each block x such that x ∈ L Do
3:    If x is dirty Then
4:      If current_time > (the last flushing time of x + fd) Then
5:        Add x to N;
6:      End If
7:    End If
8:  End For
9:  Sort the blocks of N according to the block position;
10: T is an empty list of blocks
11: spos := NULL;
12: For each block x such that x ∈ N Do
13:   If spos is NULL Then
14:     spos := the position of x;
15:   End If
16:   If (the position of x > spos + b) or (the number of blocks in T >= m) Then
17:     Flush the blocks of T at one time;
18:     Initialize T;
19:     spos := NULL;
20:     Sleep for interval fi;
21:   End If
22:   Add x to T;
23: End For
24: If T is not empty Then
25:   Flush the blocks of T at one time;
26: End If
27: For each block x such that x ∈ L Do
28:   If x is not dirty Then
29:     If current_time > (the caching time of x + cd) Then
30:       Remove x from L;
31:     End If
32:   End If
33: End For
```

End Procedure

The effect of the above algorithm is to have the memory-buffer manager replace the operating system's *pdflush*-like flushing mechanism so that the large flush is split into

smaller flushes, as shown in Fig. 10. Transaction logging, therefore, progresses steadily without interruptions, and the metadata server's processing delay is eliminated such that the corresponding metadata operation continues to be processed immediately.

The most dominant algorithm, Manage_Block(), has the feature of sorting out the dirty blocks of the blocks managed by the memory-buffer manager. The complexity varies depending on the sorting algorithm. In the implementation of this paper, the quick sort algorithm is used, and if the number of dirty blocks is k, then the sorting complexity is O(k log k). Thus, the complexity of Manage_Block() is bound to O(k log k).

The parameter values of Algorithms 3 and 4 depend on the memory size of the metadata server, the load imposed on the metadata server and the I/O performance of the metadata server. Taking these factors into account, we can optimize the parameter values so that the metadata operations per seconds have the best performance.

4 Analysis and evaluation

In this section, the performance of the proposed metadata management system on which the directory-based distribution model is applied is evaluated when many metadata servers are run. In addition, the metadata performance interference problem is illustrated through tests and analyses, after which the test results pertaining to how the memory-buffer manager affects the solution to the interference problem are presented.

4.1 Experimental setup

The metadata management system proposed in this paper is implemented in EEFS, an Exascale File System developed by the Electronics and telecommunications research institute in Korea. EEFS consists of (1) clients providing POSIX interfaces based on low-level FUSE, (2) metadata management servers where the metadata management system of this paper is applied and (3) data servers that store data. The storage area controlled by EEFS is mounted on all client machines prior to the running of the tests in this chapter. The operation requested at the mount point is, therefore, interpreted by the metadata management system of EEFS and processed as a metadata operation so that its performance can be measured.

The test environment is as follows. One-to-sixty-four Dell PowerEdge R530 servers with Intel Xeon CPUs were used as metadata management servers. Each metadata management server uses a 6 TB Seagate hard disk as its storage medium, and 10 G Ethernet is used as the network among the servers. The operating system of each machine is CentOS 7.0, and our metadata management system was installed on each metadata server. Client machines also use 1–64 machines with the same specifications.

To generate POSIX operations on the mount points of the distributed file system, we implemented a metadata workload generator, the structure of which is shown in Fig. 11.

Fig. 10 Metadata disk usage over time when our *memory-buffer manager* mechanism is applied

Multiple agents of the workload generator are started on each client machine, and all of the agents wait for a control command from one controller. Only one controller is run in the workload generator cluster, and it can enable all agents of the workload generator to start or stop repeatedly sending requests for a specific metadata operation, such as metadata creation, to all EEFS mount points. Given that each agent acts as a single thread and considering that multiple agents can be run per client machine, a large number of metadata operations can be requested at the same time. Once a specific type of metadata operation is started, it is repeated until the specified number is reached.

A full path consisting of a deep directory hierarchy is used when requesting metadata operations. For example, if an agent issues a POSIX operation such as "creat(path, 0666)," a very long absolute path, such as "/mnt/posix1/hostname/1/0000000000/0000000004/00-00-00-04@etri.re.kr/U[B@174fa0ef11953158889485628922-283.dat" is randomly generated and assigned to the first argument of *creat*(). If there is no directory belonging to the absolute path, commands to make such directory components are repeatedly sent to the metadata server cluster, after which a command to create the last file component of the path is transmitted to the metadata server cluster.

If a large number of clients exist and numerous agents per clients are run, a workload generator can be a very appropriate tool for a large-scale stress test on a distributed file system because a great number of POSIX operations containing

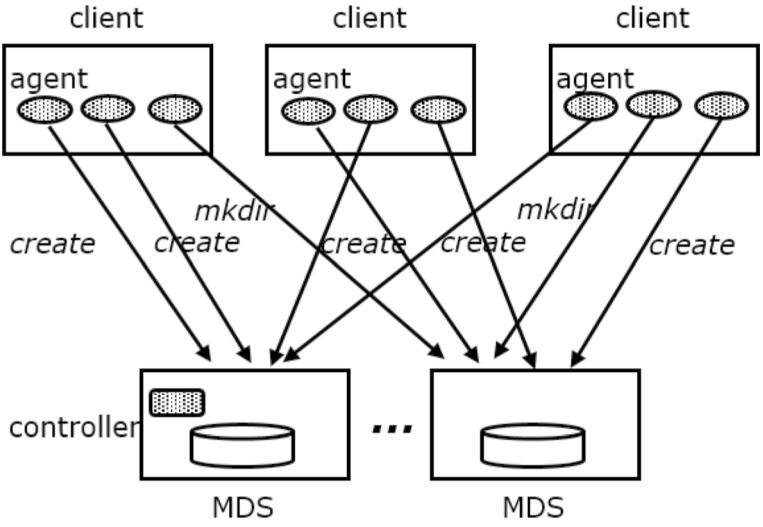


Fig. 11 Architecture of the metadata workload generator

a deep hierarchical path, which is randomly generated, can be sent to the file system.

4.2 Metadata operation performance

We installed the proposed metadata management system software on metadata servers and measured the performance of metadata operations while changing the number of metadata servers. The test results are shown in Fig. 12.

Figure 12a shows the metadata generation performance per second when a *creat*() request whose argument has a random path is continuously sent to the metadata servers. As the number of metadata servers increases, the maximum performance also increases proportionally, whereas the average performance does not. Identical phenomena are observed in the cases of *unlink*() and *mkdir*() shown in Fig. 12c, d respectively. However, in the case of *open*(), in which metadata is neither generated nor changed, both the maximum and the average performances increase in proportion to the number of metadata servers, as shown in Fig. 12b.

This phenomenon arises due to the metadata performance interference problem, which is analyzed in Sect. 4.3.

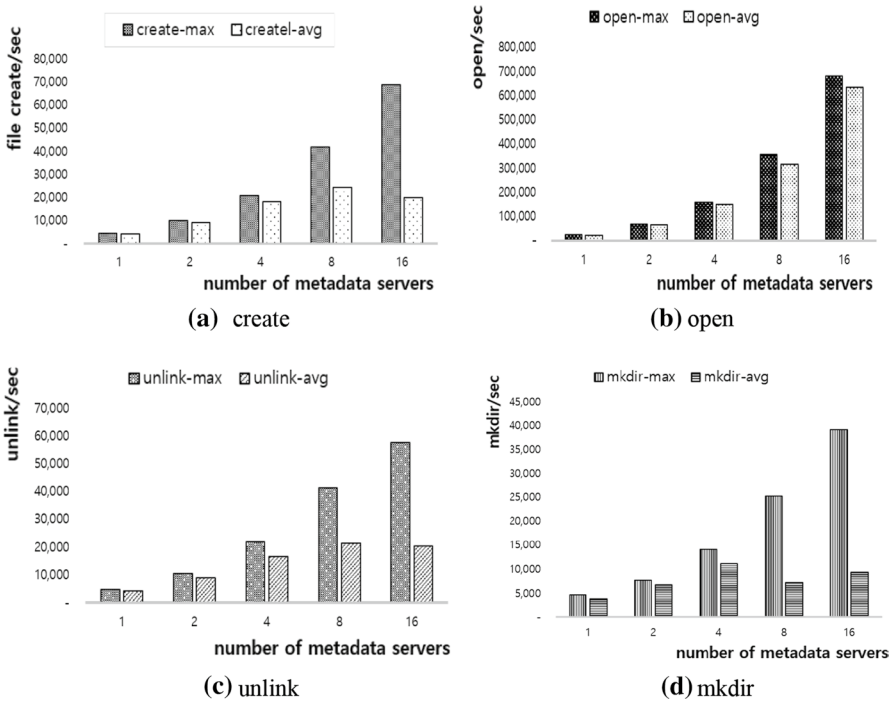


Fig. 12 Performance measured in operations per second

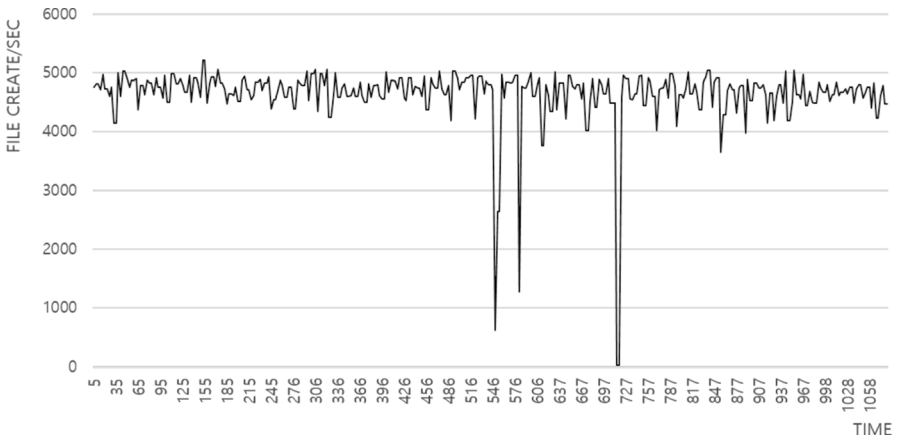


Fig. 13 Performance changes over time on a single metadata server

4.3 Metadata performance interference and analysis results

As preparation for identifying cases of metadata performance interference, we repeatedly created metadata on a single metadata server. This workload was chosen because it is the dominant pattern in the real workload [20]. The test result is shown in Fig. 13, indicating the general performance of 4600 create/s. However, there are some significantly degraded sections.

Figure 14 shows the result when measuring the metadata generation performance on eight metadata servers where directories are distributed by the directory-based distribution model. Peak performance of 43,000 create/sec is measured, but this performance level is not maintained and the performance degradations occur frequently. Compared to the results of the single metadata server shown in Fig. 13, it is confirmed that more serious performance degradation occurs in Fig. 14.

Henceforth, if one of the metadata servers flushes, we analyze whether clients experience poor performance of metadata services. In a situation with m metadata servers, the probability P that a metadata server doing a flush is not selected i times in a row as the target metadata server for a request sent by a client is given by Eq. (1).

$$P(i) = \left(\frac{m-1}{m}\right)^{i} \tag{1}$$

Suppose the number of requests per second made from clients is k; in this case, the probability Q that a metadata server that is flushing is not selected i times in a row as the metadata server to process k requests per second made from the clients is given by Eq. (2).

$$Q(i) = \left(\frac{m-1}{m}\right)^{ki} \tag{2}$$

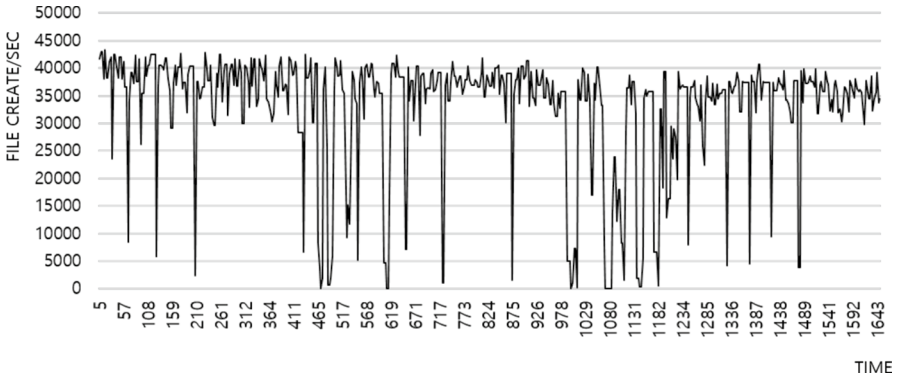


Fig. 14 Performance changes over time on eight metadata servers

According to Eq. (2), the greater the number of requests to metadata servers, the sooner a client's metadata request is always delivered to a metadata server that is flushing.

In other words, even when only one metadata server is flushing in a large cluster of metadata servers, a client's request is delivered to the metadata server after only a few requests. The meaning of this phenomenon is as follows.

When a metadata server enters a processing delay zone, the processing of metadata operations sent to the metadata server is delayed. Thus, a client who sent a metadata request to such a metadata server cannot immediately receive the result of the request and must wait for a significant amount of time. In contrast, metadata requests sent to other metadata servers that are not in a processing delay zone are immediately processed.

If a client sends metadata requests to all metadata servers with equal probabilities, the requests that were sent to the metadata server that did not enter the processing delay zone are answered immediately, and the client that receives the processing results can afford to generate new metadata requests. However, the newly generated requests become more likely to go to the metadata server that has entered the processing delay zone and is now delaying the processing of the previously received requests. As a result, despite the fact that most of the metadata servers have no processing delay, some of the operations newly requested by clients are repeatedly delivered to the metadata server that is in the processing delay zone such that the situation deteriorates further and such clients cannot request new operations in proportion to the amount of the already delayed operations whose results are being waited for. Thus, the quality of the metadata service is seriously degraded.

Each metadata server performs a periodic flush, and if there are n metadata servers performing such a flush, the probability R that a request of a client will be forwarded to a metadata server that is flushing is expressed by Eq. (3), where x represents how many times metadata servers that are flushing is not selected x times in a row as the target of a metadata request.

$$R(x) = \sum_{x=0}^{i} \left(1 - \frac{n}{m}\right)^{x} \times \frac{n}{m} \tag{3}$$

As n approaches m, Eq. (3) converges to 1. That is, the greater the number of metadata servers that are performing a flush, the closer the probability of accessing such metadata servers is. Because each metadata server flushes periodically, the frequency of a metadata server entering a processing delay zone becomes much greater as the number of metadata servers increases. As a result, the problem of the performance of the entire metadata server cluster becoming degraded occurs more frequently.

If the period of a large flush that blocks the I/O is T, the probability S that a metadata server maintains a normal state in which flushing does not occur while requests for metadata operations, starting at an arbitrary time t, are repeated i times is expressed by Eq. (4).

$$S(i) = \left(1 - \frac{1}{T}\right)^{i} \tag{4}$$

According to Eq. (4), even if requests of clients are repeated but T is sufficiently large, metadata servers are more likely to remain in a normal state. In conclusion, to avoid performance interference among metadata servers, the key is to reduce the probability of a large amount of flushing that may cause a service failure.

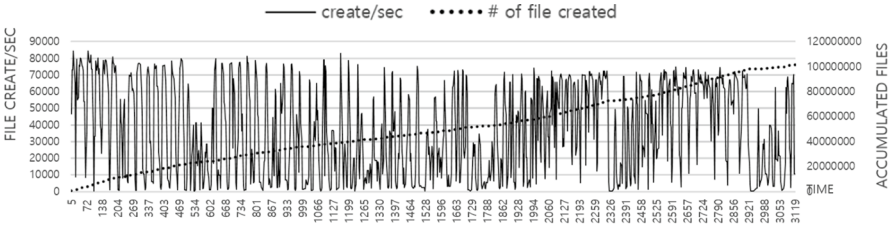
4.4 Test results of the metadata management system with the memory-buffer manager

Based on the analysis results in Sect. 4.3, we applied our memory-buffer-manager algorithm to the metadata management system to prevent large flushes from occurring. Figure 15 shows the test results. Each test in Fig. 15 served to examine how much the performance interference among metadata servers is reduced in a situation in which requests to generate metadata are continuously thrown to the metadata servers.

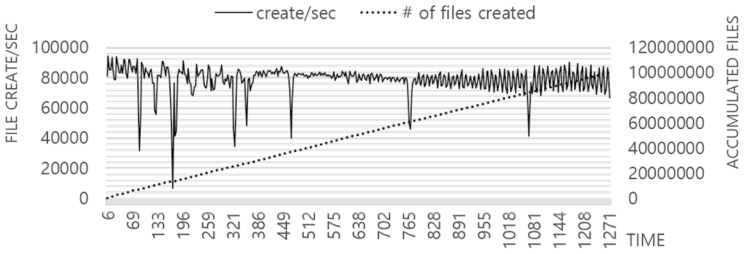
With 16 metadata servers, as shown in Fig. 15a, it takes 3119 s to create 100 million instances of metadata without the memory-buffer manager and, in this case, it can also be seen that the performance outcomes fluctuate greatly. However, when the memory-buffer manager is applied, it takes 1274 s, as indicated in Fig. 15b, achieving roughly a 2× performance improvement. In addition, the fluctuation is also minor.

The results tested on a more expanded scale are similar to those shown above. Figure 15c shows that it takes 10,092 s without the memory-buffer manager to create 1 billion instances of metadata from 64 metadata servers, but the presence of the memory-buffer manager reduces this time to only 4845 s for the same amount of metadata, as shown in Fig. 15d.

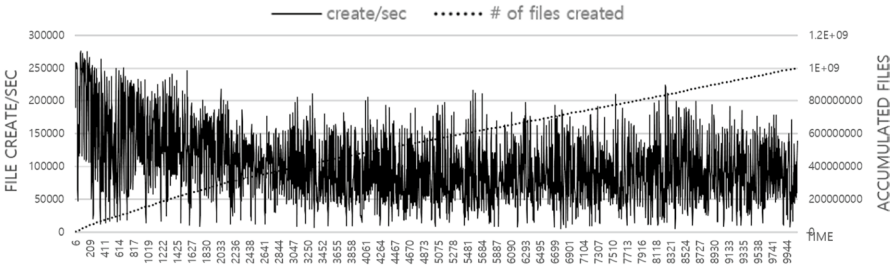
The small fluctuations shown in both Fig. 15b, d imply that with the help of the memory-buffer manager, the performance interference among metadata servers can be greatly reduced and thus a very large cluster of metadata servers can be run stably, achieving consistent performance.



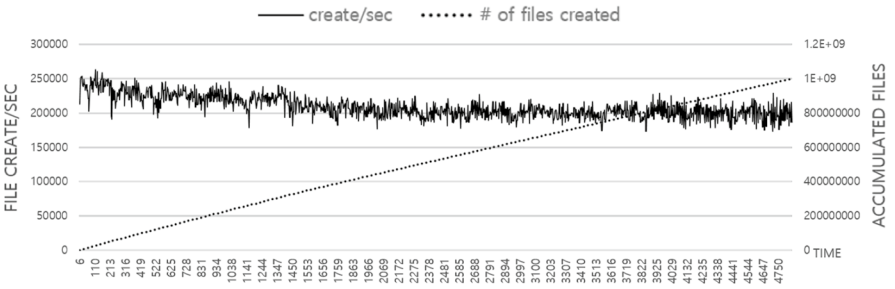
(a) Performance changes over time on 16 metadata servers without the memory-buffer manager



(b) Performance changes over time on 16 metadata servers with the memory-buffer manager



(c) Performance changes over time on 64 metadata servers without the memory-buffer manager



(d) Performance changes over time on 64 metadata servers with the memory-buffer manager

Fig. 15 Comparison of test results without or with the memory-buffer manager

5 Conclusion

Because a very large number of files, such as those on the exasacle, are beyond the manageable limits of a single metadata server, many techniques involving the use of a large-scale metadata server cluster have been presented, but existing methods have numerous problems.

In this paper, first we proposed directory-based distribution model which ensures excellent locality while maintaining a sufficient balance among servers. Second, an effective architecture of a high-performance metadata management system was proposed and implemented, enabling efficient metadata placement and memory management while also enabling the rapid processing of metadata. Finally, we found that the performance capabilities of a cluster of metadata servers are degraded due to interference among metadata servers as the number of metadata servers increases. On the basis of this fact, the cause of the performance degradation was analyzed, and a memory-buffer manager algorithm was proposed and tested to resolve this performance interference problem.

The three methods presented here enable the realization of metadata processing performance proportional to the scale of the metadata server cluster in use, allowing the efficient management of significantly large amounts of metadata on the exascale.

Acknowledgements This work was supported by Institute for Information and communications Technology Promotion (IITP) Grant funded by the Korea government (MSIP) (No. 2015-0-00262, Management of Developing ICBMS (IoT, Cloud, Bigdata, Mobile, Security) Core Technologies and Development of Exascale Cloud Storage Technology).

References

1. Konstantin S, Hairong K, Sanjay R, Robert C (2010) The hadoop distributed file system. In: Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST'10), pp 1–10
2. Oracle (2010) Lustre 2.0 operations manual. Oracle corporation. https://docs.oracle.com/cd/E19527-01/821-2076-10/821-2076-10.pdf. Accessed June 2017
3. Konstantin S (2010) HDFS scalability: the limits to growth. USENIX; login 35(2):6–16
4. Sadaf RA, Hussein NEH, Kristopher H, Neil S, Fabio V (2011) Parallel I/O and the metadata wall. In: Proceedings of the 6th Workshop on Parallel Data Storage (PDSW'11), pp 13–18
5. Sage AW (2007) Ceph: reliable, scalable, and high-performance distributed storage. Doctoral dissertation, University of California
6. Sage AW, Scott AB, Ethan LM, Darrell DEL, Carlos M (2006) Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), pp 307–320
7. Redhat (2018) Architecture. Redhat, Inc. http://gluster.readthedocs.io/en/latest/Quick-Start-Guide/Architecture. Accessed October 2018
8. Beaver D, Kumar S, Li H, Sobel J, Vajgel P (2010) Finding a needle in Haystack: Facebook's photo storage. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10), pp 47–60
9. Muralidhar S, Llyod W, Roy S, Hill C, Lin E, Liu W, Pan S, Shankar S, Sivakumar V, Tang L, Kumar S (2014) f4: Facebook's warm BLOB storage system. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14), pp 383–398

10. Bronson N, Amsden Z, Cabrera G, Chakka P, Dimov P, Ding H, Ferris J, Giardullo A, Kulkarni S, Li H, Marchukov M, Petrov D, Puzar L, Song Y, Venkataramani V (2013) TAO: Facebook's distributed data store for the social graph. In: Proceedings of USENIX Annual Technical Conference (USENIX ATC'13), pp 49–60
11. Alexander T, Daniel JA (2015) CalvinFS: consistent WAN replication and scalable metadata management for distributed file systems. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15), pp 1–14
12. Johnson C, Keeton K, Morrey III C, Soules C, Veitch A, Bacon S, Batuner O, Condotta M, Coutinho H, Doyle P, Eichelberger R, Kiehl H, Magalhaes G, McEvoy J, Nagarajan P, Osborne P, Souza J, Sparkes A, Spitzer M, Tandel S, Thomas L, Zangaro S (2014) From research to practice: experiences engineering a production metadata database for a scale out file system. In: Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14), pp 191–198
13. Xiao L, Ren K, Zheng Q, Gibson G (2015) ShardFS vs. IndexFS: replication vs. caching strategies for distributed metadata management in cloud storage systems. In: Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC'15), pp 236–249
14. Ghemawat S, Gobioff H, Leung S (2003) The Google file system. In: Proceedings of ACM Symposium on Operating Systems Principles (SOSP'03), pp 29–43
15. Brandt S, Miller E, Long D, Xue L (2003) Efficient metadata management in large distributed storage systems. In: Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'03), pp 290–298
16. Zhang S, Catanese H, Wang A (2016) The composite-file file system: decoupling the one-to-one mapping of files and metadata for better performance. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16), pp 15–22
17. Sinnamohideen S, Sambasivan R, Hendricks J, Liu L, Ganger G (2010) A transparently-scalable metadata service for the ursa minor storage system. In: Proceedings of USENIX Annual Technical Conference (USENIX ATC'10)
18. Weil S, Pollack K, Brandt S, Miller E (2004) Dynamic metadata management for petabyte-scale file systems. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)
19. Xiong J, Hu Y, Li G, Tang R, Fan Z (2011) Metadata distribution and consistency techniques for large-scale cluster file systems. IEEE Trans Parallel Distrib Syst 22(5):803–816
20. Cha M, Kim D, Kim H, Kim Y (2017) Adaptive metadata rebalance in exascale file system. J Supercomput 73:1337–1359
21. Noghabi S, Subramanian S, Narayanan P, Narayanan S, Holla G, Zadeh M, Li T, Gupta I, Campbell R (2016) Ambry: LinkedIn's scalable geo-distributed object store. In: Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16), pp 253–265
22. Memcachee (2018) https://memcached.org. Accessed July 2018
23. Thomson A, Diamond T, Weng S, Ren K, Shao P, Abadi D (2014) Fast distributed transactions and strongly consistent replication for OLTP database systems. ACM T Database Syst 39(2):11–49
24. Ren K, Thomson A, Abadi D (2014) An evaluation of the advantages and disadvantages of deterministic database systems. Proc VLDB Endow 7(10):821–832
25. Cipar J, Ganger G, Keeton K, Morrey III C, Soules C, Veitch A (2012) LazyBase: trading freshness for performance in a scalable database. In: Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12), pp 169–182