Metadata Distribution and Consistency Techniques for Large-Scale Cluster File Systems

Jin Xiong, Yiming Hu, *Senior Member, IEEE*, Guojie Li, Rongfeng Tang and Zhihua Fan

Abstract—Most supercomputers nowadays are based on large clusters, which call for sophisticated, scalable, and decentralized metadata processing techniques. From the perspective of maximizing metadata throughput, an ideal metadata distribution policy should automatically balance the namespace locality and even distribution without manual intervention. None of existing metadata distribution schemes is designed to make such a balance. We propose a novel metadata distribution policy, Dynamic Dir-Grain (DDG), which seeks to balance the requirements of keeping namespace locality and even distribution of the load by dynamic partitioning of the namespace into size-adjustable hierarchical units. Extensive simulation and measurement results show that DDG policies with a proper granularity significantly outperform traditional techniques such as the Random policy and the Subtree policy by 40% to 62 times. In addition, from the perspective of file system reliability, metadata consistency is an equally important issue. However, it is complicated by dynamic metadata distribution. Metadata consistency of cross-metadata server operations cannot be solved by traditional metadata journaling on each server. While traditional two phase commit (2PC) algorithm can be used, it is too costly for distributed file systems. We proposed a consistent metadata processing protocol, S2PC-MP, which combines the two phase commit algorithm with metadata processing to reduce overheads. Our measurement results show that S2PC-MP not only ensures fast recovery, but also greatly reduces fail-free execution overheads.

Index Terms—Distributed file systems, metadata management

1 Introduction

METADATA processing is a key issue for large-scale cluster file systems. Metadata is the data that describes the organization and structure of a file system, usually including directory contents, file attributes, file block pointers, organization and state information of physical space, etc. Metadata processing includes the maintenance of not only the namespace, but also file attributes and locations of file blocks. Although the amount of metadata is small, metadata operations account for over 60% of the operations in typical workloads [19]. Therefore, the efficiency of metadata processing will ultimately affect the overall file system performance.

Large clusters nowadays are at a scale of thousands to tens of thousands of nodes per cluster, with an order of magnitude more CPU cores. As their capacity and user requirements increase, so is the demand for high performance distributed file systems on these clusters. According to Dayal's statistical data [33] over 13 file systems from 5 supercomputing sites and two file systems from department file servers, the total number of files as well as directories of each file system is several millions to tens of millions. Moreover, the processes running on each CPU core issue I/O accesses independently, resulting in large numbers of concurrent accesses to the file systems. Therefore, file systems on large-scale clusters are required to handle tens of thousands of files' metadata per second. These requirements indicate that centralized metadata processing that relies on a single metadata server is impractical and it is necessary to use *decentralized metadata processing* [12, 14, 18, 25, 38] that utilizes a group of *metadata servers* (MDS).

Namespace partitioning and metadata distribution are the key issues in decentralized metadata processing that determine metadata processing throughput. Many existing systems use static partitioning [34, 30, 22] which relies on administrators to manually partition the namespace and distribute the metadata. This static method is obviously very difficult to maintain for large-scale clusters. Although a lot of systems use dynamic partitioning, they either emphasize on keeping namespace locality [7, 27, 25] or focus on equal distribution [35, 31, 32, 18], and hence neglect the tradeoff between the two aspects. As a result, these systems cannot make full use of all metadata servers' processing and storage resources. In contrast to these systems, we propose a metadata distribution policy that dynamically partitions the namespace into size-adjustable hierarchical units, and achieves a good balance between the namespace locality and equal distribution.

In addition, metadata consistency (the atomicity of a metadata operation) is complicated by cross-metadata server operations (known as *distributed operations*, see detailed discussions in Section 2.1.1). Individual journaling metadata modifications on each server, as in journaling

- *J. Xiong, G. Li and R. Tang are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China. E-mail: xj@ncic.ac.cn, lig@ict.ac.cn, rf_tang@ncic.ac.cn.*
- *Y. Hu is with the Department of Electrical & Computer Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221. E-mail: huyg@ucmail.uc.edu.*
- *Z. Fan is with the NetEase.com, Inc, Beijing 100084, China. He participated in this work when he was a graduate student at the Institute of Computing Technology, CAS. E-mail: fanzhihua@gmail.com.*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

file systems[24], cannot ensure consistency of distributed operations. To address this problem, existing systems either ensure metadata consistency by using the high-overhead two-phase commit (2PC) protocol at the sacrifice of performance [1, 11], or pursue high performance at the cost of relaxing the consistency model by allowing orphan objects[1] in the namespace [18, 29].

This paper makes two contributions. First, we present a dynamic metadata distribution policy called *dynamic dir-grain policy* (DDG) to improve the metadata throughput by using a triple-defined *distribution granularity* to dynamically partition the namespace and balance metadata distribution and the namespace locality. Our experiment results show that DDG policies with proper granularity setting outperform two traditional policies — the *Random* policy and the *Subtree* policy by 40% to 62 times under metadata intensive workloads.

Second, we propose a distributed metadata processing policy called S2PC-MP that deals with metadata consistency for distributed operations. By combining the two-phase commit protocol with metadata operations, S2PC-MP reduces the overhead of failure-free execution, and ensures quick recovery in the presence of metadata server failures or client failures. Our performance results show that only the creation and deletion throughputs decreased about 10%, and the read/write bandwidth and transaction throughputs were almost unaffected. Recovery can be completed within tens of seconds.

The rest of this paper is organized as follows. In Section 2, we summarize previous research work and techniques. We introduce the new metadata distribution policy in Section 3 and the S2PC-MP technique in Section 4. In Section 5, we evaluate the performance of the proposed solutions and compare them with those of two well-known policies. We conclude the paper in Section 6.

2 Related Work

Decentralized metadata processing (DMP) is necessary for large-scale distributed file systems to provide scalable performance. No matter metadata is stored on each metadata server's private storage [34, 30, 1, 18, 7] or on a storage system shared by all metadata servers [14, 25, 38], DMP partitions the namespace among the servers. Metadata distribution and metadata consistency are two key issues that not only affect the metadata performance but also determine file system reliability.

2.1 Metadata Distribution Policies

In essence, a metadata distribution policy first partitions the namespace into distribution units of smaller size, and then maps these units to the metadata server set.

2.1.1 Static Partitioning vs. Dynamic Partitioning

Metadata distribution policies can be divided into two types according to how the namespace is partitioned.

In *static partitioning*, the namespace is manually partitioned by administrators according to their experiences and target application requirements. Early distributed file systems such as NFS, Andrew[34], Sprite[30] and some recent distributed file systems such as CXFS[22] and StorageTank[14] use static partitioning.

In *dynamic partitioning*, the namespace is automatically partitioned and mapped, usually at the time of objects creation in the namespace. So in dynamic partitioning, there are *distributed metadata operations* in addition to *ordinary metadata operations*. A *distributed metadata operation* (or *distributed operation* for short) must be performed by collaboration of two or more metadata servers, while an *ordinary metadata operation* (or *ordinary operation* for short) can be performed by just one metadata server. For example, when create a new file *f1* under a directory *d1* who is managed by *server1*, a dynamic partitioning policy may decide that *f1* is managed by *server2*, and hence the create operation needs to request both *server1* and *server2*. File systems using dynamic partitioning include xFS[35], Slice[1], PVFS2[18], Kosha[7], DCFS[27], GIGA+[36], etc.

Obviously, dynamic partitioning greatly reduces the maintenance cost in these file systems. Static partitioning requires administrators to perform too much maintenance work, including namespace partitioning and mapping in the beginning, and constant re-adjusting during the whole life of the file systems.

Our DDG policy belongs to dynamic partitioning, and it does not require human intervention in namespace management.

2.1.2 Partition Granularity

Metadata distribution policies can be divided into three types according to their partition granularity.

In *subtree partitioning*, the namespace is partitioned into several subtrees which are then mapped to the metadata servers. The file systems employing static partitioning of namespace usually use subtree partitioning. Examples include NFS, Andrew[34], Sprite[30], CXFS[22] and StorageTank[14]. This type of partitioning is known as *static subtree partitioning*. Some file systems, such as Kosha[7], IFS[11], DCFS[27] and Ceph[25], also use *dynamic subtree partitioning*. Although dynamic subtree partitioning overcomes the management problem of the static subtree partitioning, both of them suffer from the problem that their distribution granularity is too large to balance both the resource usage and the workload among the servers.

In *single object partitioning*, the namespace is partitioned into individual objects (files or directories) which are then mapped to metadata servers through a random algorithm. xFS[35], HBA[31], G-HBA[32], PVFS2[18], etc. use this method. This approach can achieve better load balance than subtree partitioning because of the smallest partition granularity, but the major limitation of this method is that there are too many *branch points* (to be discussed in Section 3.1) and hence too many distributed metadata operations to achieve high metadata throughput.

In *directory segment partitioning*, each directory in the namespace is partitioned into fixed-sized segments through extendible hashing, and these segments are mapped to metadata servers. GIGA+[36] and SkyFS[37] use this method. The idea of partitioning a single directory is motivated by current requirements of very large di-

[1] We refer to files or directories in a namespace as objects in this paper.

rectories with millions of entries in each directory. Directory segment partitioning is targeted to solve the problem of a very large single directory. However, for many file systems in supercomputing centers, 90% directories have less than 64 entries [33]. For these cases, this method is less effective. An even bigger issue is that it is difficult to support POSIX *readdir* operation which is widely-used in supercomputing centers.

Different from the above methods, our DDG policy dynamically partitions the namespace into smaller hierarchical units. The novelty of DDG policy is that it provides a way to control distribution granularity, by which it can keep namespace locality for the majority of objects after namespace partitioning. Therefore, it can achieve better metadata throughput than subtree partitioning and single object partitioning. Moreover, DDG policy supports most POSIX file operations including *readdir*, and also partitions directories, hence can deal with both large and small directories. Therefore, it is a more general methodology than directory segment partitioning.

2.1.3 Mapping Method

In general, there are three methods to map each unit to a metadata server.

Manual mapping (NFS, Andrew[34], Sprite[30], etc.) relies on administrators to perform the mapping and uses a static table to maintain the mapping. Apparently, too much manpower and lack of flexibility are the main drawbacks of such a method.

Random mapping (xFS[35], PVFS2[18], HBA[31], ANU randomization[26], G-HBA[32], etc.) uses a random algorithm to automatically map units to servers. Additionally, xFS uses a table to maintain the mapping, while HBA uses a Bloom Filter-based method and significantly reduced the memory overhead. However, in the former three systems, the mapping cannot adjust automatically with the change of access loads or server configuration. On the contrary, ANU randomization also remaps units to servers according to observed request latencies, and G-HBA maintains additional information to support the server configuration changes.

Hashing-based mapping (LazyHybrid[6], GIGA+[36] etc.) uses a simple hash algorithm to automatically map units to servers. Such a simple hashing makes them less adaptable to the changes of server configurations. LazyHybrid eliminates hierarchical relationships of objects, resulting in a severe performance decline for some operations such as *readdir*, renaming a directory, and changing the mode or owner of a directory.

DDG policy uses random mapping. Our system retains the simplicity and even distribution of random mapping, and avoids the difficulty of maintaining the mapping by encoding each object's location (the metadata server ID) into its unique ID (inode number).

2.2 Metadata Consistency

A file system operation consists of a sequence of indivisible sub-operations, each of which modifies a single piece of metadata. Therefore, a file system operation is similar to a transaction and has ACID properties[10]. *Metadata consistency* refers to a metadata operation's result is either it successfully completed or never happened. This issue is important since a metadata operation usually modifies multiple data structures. In native file systems, inconsistency is mainly caused by metadata caching if different types of metadata are cached in different memory locations, and written back individually. In distributed metadata processing, inconsistency is caused not only by metadata caching but also by distributed operations. For example, the *create* operation creates a new file *f1* under a directory *d1*. If *d1* and *f1* are maintained by two distinct servers (S_1 and S_2 respectively), S_1 and S_2 modify different data structures respectively. If either S_1 or S_2 crashes, the result on this server may be lost, thus may lead to inconsistent metadata on these two servers. The metadata consistency issue that this paper intends to address is to make an entire metadata operation atomic across multiple metadata servers.

Journaling file systems[24] ensure metadata consistency by logging metadata updates. Many distributed file systems, such as NFS, Calypso[8], CXFS[22] and Lustre[5], have a single metadata server, and hence can rely on server-side journaling file systems to ensure metadata consistency. In addition, some distributed file systems, such as Sprite[2], StorageTank[14], GPFS[20] and GFS[17], have no distributed metadata operations, so they can ensure metadata consistency by loging metadata updates on individual servers.

Individual logging on each server cannot ensure consistency of distributed operations each of which requires collaboration of two or more servers. Two-phase commit (2PC)[9, 4, 23] in distributed transaction processing is used by some distributed file systems such as Slice[1] and IFS[11] in their metadata processing. However, the main concern of using 2PC in metadata processing is its high fail-free execution overhead introduced by two rounds of messages as well as multiple disk writes in a single commit. It is unclear from the literature that any optimization efforts were undertaken in these systems to reduce the high overheads of 2PC.

To avoid 2PC's high overhead and complexity, neither DiFFS[29] nor PVFS2[18] uses 2PC, although they have distributed metadata operations. However, both of them may have "orphan" objects after server failures and they ensure "orphan" objects will not be seen or accessed by users. They achieve high performance by relaxing the consistency model, allowing orphan objects.

Ursa Minor avoids 2PC by migrating the required metadata to a single metadata server and executing the operation on that server[38]. This approach simplified the implementation of multi-server operations. However, the performance overheads for multi-server operations are high due to metadata migration. Since its target environments have much few multi-server operations (less than 1%), the performance loss is acceptable.

We propose a novel scheme called S2PC-MP to deal with the high failure-free execution overhead of 2PC in metadata processing. By combining 2PC with each metadata operation instead of starting 2PC at the end of each operation, our scheme significantly reduces failure-free execution overhead by reducing the number of messages between collaborative servers, and by reducing the number of disk writes.

Some systems such as the GFS[39] use metadata replication to ensure metadata reliability and availability, and the primary copy protocol to make multiple replicas consistent. Both 2PC and primary copy serialize metadata at a single node for modification operations. However, metadata replication does not partition the namespace and GFS uses a centralized metadata management scheme. Through replication, GFS allows multiple servers to service read requests, but all write requests must be serviced by the master node which is a potential bottleneck for scalability. We believe that future applications need both metadata replication and metadata distribution. The former improves availability, while the latter improves scalability. We plan to integrate both methods in metadata management in our future work.

3 Dynamic Dir-Grain Policy

We designed a dynamic metadata distribution policy to fully exploit all metadata servers' processing and storage resources.

3.1 Namespace Locality

The hierarchical structure of file system namespace means that there is some kind of relationship among the objects in it. Many metadata operations, such as *lookup*, *create*, *remove*, *mkdir*, *rmdir* and *rename,* access or modify not only the target object itself, but also its parent directory. If the target and its parent are maintained by the same metadata server, then those operations can be performed by just this server. Otherwise, two or more servers are needed. We call an object in the namespace a *branch point* if its parent and it are maintained by two different metadata servers. A branch point is either a directory or a file. And we call the former a *branch directory* and the latter a *branch file*. Branch points break up the relationships of the objects and weaken the namespace locality.

Branch points are inevitable in decentralized metadata processing, otherwise all files and directories in the namespace have to be managed by the same metadata server. Inducing branch points is an intrinsic nature of metadata distribution. However, branch points weaken the namespace locality, causing more processing overheads.

Metadata throughput is the total number of operations processed by all the metadata servers per second. It is affected by the following two factors:

- *The degree of balance of objects distribution.* Very imbalanced distribution causes some metadata servers are too busy to process a large number of requests, while leaving some other servers mostly idle.
- *The number of branch points.* More branch points result in more distributed metadata operations, which are much more expensive than ordinary operations. Each distributed operation needs to interact with other metadata servers to complete the required operation. Moreover, distributed operations (such as file creation and deletion) which modify the namespace need 2PC to ensure metadata consistency. As discussed in Section 4.2, each distributed modification operation needs three network messages and four synchronous disk writes of metadata log, while an ordinary modification operation needs no network message and only one asynchronous disk write of metadata log. A distributed *lookup* needs an inter MDS message, while an ordinary *lookup* does not. Therefore, more branch points not only lead to longer latencies of the operations, but also reduce metadata throughput by causing heavier network traffic and more disk writes.

These two factors are often incompatible. For example, a round-robin policy that tries to evenly distribute objects across multiple servers is likely to generate a large number of branch points. To obtain optimal performance, it is important to seek a balance between them.

3.2 Distribution Granularity

The size of a distribution unit is known as *distribution granularity*. A smaller granularity results in more even distribution but more branch points, while a larger granularity leads to fewer branch points but less even distribution. The distribution unit of single object partitioning (discussed in Section 2.1.2) is a single object, so its granularity is 1, the smallest possible value. These policies emphasize on even distribution, at the cost of a large number of branch points. On the other hand, the distribution unit of subtree partitioning is a whole subtree, so its granularity is much larger. Subtree policies emphasize on keeping the namespace locality, which results in imbalanced distribution.

In order to find a compromise between the two factors, we introduce D-D-F granularity, which defines a hierarchical unit in the namespace by a triple <DirDep, DirWid, FileWid>. DirDep stands for directory depth, which defines the maximum number of levels of directories in a distribution unit. DirWid stands for directory width, which defines the maximum number of child directories[2] of each directory in a distribution unit. FileWid stands for file width, which defines the maximum number of child files of each directory in a distribution unit. Figure 1 shows an example of a distribution unit with a granulari-

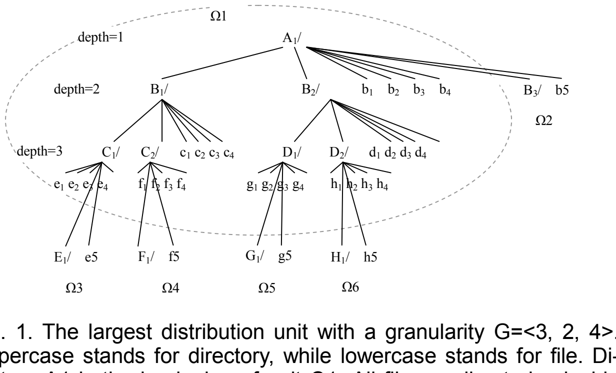


Fig. 1. The largest distribution unit with a granularity G=<3, 2, 4>. Uppercase stands for directory, while lowercase stands for file. Directory A1 is the beginning of unit Ω1. All files or directories inside the dashed circle belong to unit Ω1. Files and directories outside the dashed circle belong to other units.

[2] Child directories of a directory "P" are those directories whose parent is P. And child files of a directory "P" are those files whose parent is P.

ty G=<3, 2, 4>.

The existing partitioning policies described in Section 2.1.2 can also be viewed as special cases of certain D-D-F granularities. For example, the granularity of single object partitioning is <1, 1, 1>. And the granularity of directory segment partitioning is <1, m, n> where m+n equals to segment size. The root subtree policy uses two different granularities: <1, 1, 1> for root and <∞, ∞, ∞> for other directories. Kosha's subtree policy also uses two granularities: <1, 1, 1> for objects in levels higher than or equal to the distribution level, and <∞, ∞, ∞> otherwise.

3.3 Distribution Algorithm

Our distribution policy (DDG) assigns a metadata server to an object when the object is created. Suppose *D* is a directory in unit Ω. Suppose that *D*' is a directory created most recently under *D*, and *D*' belongs to unit Ω'. And suppose that *f* is a file created most recently under *D*, and *f* belongs to unit Ω''. The values related with *D* include:

- *dir_depth*: the depth of D in unit Ω;
- *dir_mds*: the metadata server that maintains unit Ω';
- *dir_num*: the number of child directories of D in unit Ω';
- *file_mds*: the metadata server that maintains unit Ω'';
- *file_num*: the number of child files of D in unit Ω''.

For example, for A1 in Figure 1, its dir_depth=1, its dir_mds=MDS(Ω2), its dir_num=1, its file_mds=MDS(Ω2), and its file_num=1. For C1 in Figure 1, dir_depth=3, its dir_mds=MDS(Ω3), its dir_num=1, its file_mds=MDS(Ω3), and its file_num=1.

Suppose the distribution granularity is <DirDep, DirWid, FileWid>, and a user wants to create a new object *X* under directory *P*. Five conditions should be taken into account when assigning a metadata server for object *X*.

1. If *X* is a file, and its parent *P*'s file_num +1 ≤ FileWid, then assign P's file_mds to *X*.
2. If *X* is a file, and its parent *P*'s file_num +1 > FileWid, then choose a metadata server MDS_i according to certain random function, and assign MDS_i to *X*.
3. If *X* is a directory, and its parent *P*'s dir_depth +1 ≤ DirDep, and *P*'s dir_num +1 ≤ DirWid, then assign *P*'s dir_mds to *X*.
4. If *X* is a directory, and its parent *P*'s dir_depth +1 ≤ DirDep, and *P*'s dir_num +1 > DirWid, then choose a metadata server MDS_j according to certain random function, and assign MDS_j to *X*.
5. If *X* is a directory, and its parent *P*'s dir_depth +1 > DirDep, then choose a metadata server MDS_k according to certain random function, and assign MDS_k to *X*.

After assigning the metadata server for the new object *X*, DDG also modifies the five values of *P* and *X* accordingly. The pseudo code of the DDG algorithm is depicted in Figure 2.

The performance of DDG policy depends on the values of D-D-F, and the proper values of D-D-F depend on the namespace characteristics of the target file system, access workloads, and the metadata servers' processing ability. Our evaluation in Section 5 arrived at a set of reasonable values for DDF based on our data sets and system configuration. These values are applicable for similar environments. Since in different systems, the namespace structures, access workloads and metadata servers are very different, there are no such values of D-D-F that are suitable for all environments. Therefore, a preferable way is to adjust the values of D-D-F adaptively by DDG policy itself with the growth of the namespace, and changes of access workloads and server configuration. However, current DDG does not support such adaptation and the values of D-D-F can only be re-adjusted by administrators.

```
If (X is a file) then
    If (P.file_num + 1 <= G.FileWid) then
        Choose P.file_mds to be X's server
        P.file_num ← P.file_num+1
    else
        randomly select a server $MDS_i$ to be X's server
        P.file_mds ← $MDS_i$
        P.file_num ← 1
    endif
else if (X is a directory) then
    if (P.dir_depth + 1 > G.DirDep) then
        randomly select a server $MDS_j$ to be X's server
        P.dir_mds ← $MDS_j$
        P.dir_num ← 1
        X.dir_depth ← 1
        X.dir_mds ← $MDS_j$
        X.dir_num ← 0
        X.file_mds ← $MDS_j$
        X.file_num ← 0
    else if (P.dir_num + 1 <= G.DirWid) then
        Choose P.dir_mds to be X's server
        P.dir_num ← P.dir_num + 1
        X.dir_depth ← P.dir_depth + 1
        X.dir_mds ← P.dir_mds
        X.dir_num ← 0
        X.file_mds ← P.file_mds
        X.file_num ← 0;
    else
        randomly select a server $MDS_k$ to be X's server
        P.dir_mds ← $MDS_k$
        P.dir_num ← 1
        X.dir_depth ← 1
        X.dir_mds ← $MDS_k$
        X.dir_num ← 0
        X.file_mds ← $MDS_k$
        X.file_num ← 0
    endif
endif
```

Fig. 2. The DDG Algorithm.

4 Reliable Distributed Metadata Processing Protocol

Since 2PC is the basis of our protocol, we first give a brief introduction to 2PC in this section.

4.1 Two-Phase Commit Protocol (2PC)

Atomic commitment protocols are proposed to ensure consistency of distributed transactions especially in the presence of failures[4]. And 2PC protocol [9, 4, 23] is the simplest and most popular atomic commitment protocol. The sites involved in a distributed transaction usually play two different roles. The site that initiates the transaction is called the *coordinator*, while the remaining sites are called *participants*. Both the coordinator and all participants write down their progress by writing log records to stable storage in the course of commitment.

The first phase of 2PC is called *voting phase*, in which the coordinator sends a VOTE-REQ message to all participants to ask them if they agree to commit. Meanwhile it writes the "Start-2PC" record in its log. And a participant writes the "Yes" record in its log if it votes Yes, and sends a VOTE-Yes message back to the coordinator. Otherwise, it writes the "Abort" record and sends a VOTE-No message back. The participants that vote Yes then wait for the next message from the coordinator.

The second phase is called *commitment phase*. After the coordinator has collected the reply messages (either VOTE-No or VOTE-Yes) from all participants, it then decides Commit by writing the "Commit" record in its log if all votes are VOTE-Yes, and then sends COMMIT-REQ messages to all participants. Otherwise, the coordinator decides Abort by writing the "Abort" record in its log and sends ABORT-REQ messages to all participants that vote yes. A participant writes the "Commit" record in its log if it receives "COMMIT-REQ". Otherwise, it writes the "Abort" record in its log. And the participant sends an acknowledgement message to the coordinator.

Suppose there is just one participant besides of the coordinator, in normal case when no failure occurs in the course of transaction commitment, 2PC needs four messages, and five disk writes of log records. In 2PC protocol, both the coordinator and participants that vote yes wait messages. The coordinator waits votes and acknowledgement messages from all participants, while the participants wait the VOTE-REQ and the final decision from the coordinator. The waiting message will never arrive if failures occur in the course of commitment. To address this problem, 2PC protocol uses timeout mechanism and the cooperative termination protocol [4].

4.2 S2PC-MP Protocol

Since operations that need more than two metadata servers are very scarce, S2PC-MP only deals with operations that need exactly two metadata servers[3]. For convenience, any distributed operation can be denoted by executing a sub-operation (*Sub-op1*) on the first metadata server, then executing another sub-operation (*Sub-op2*) on the second metadata server. The former is called a "*coordinator*", which receives the metadata request from a client, while the latter is called a "*participant*", which receives the metadata request from the coordinator. In S2PC-MP, *create*, *remove*, *mkdir* and *rmdir* operations are split among the two servers in the way showed in Table 1.

The S2PC-MP protocol is depicted in Figure 3(a). The key idea here is to combine request processing with commitment to reduce the processing latency. It is unnecessary for the coordinator to send the request to the participant if Sup-op1 fails. The coordinator will send the request to the participant only when Sup-op1 succeeds, and this request indicates that the coordinator has voted "yes". And as a comparison, the metadata processing directly based on 2PC without any improvement is depicted in Figure 3(b).

In S2PC-MP, the coordinator first completes Sup-op1, and writes the result record into its log. If the Sub-op1

TABLE 1
SPLIT OF OPERATIONS

Operations	Sub-op1 (Coordinator)	Sub-op2 (Participant)
create	Insert a new entry in the parent directory, and update parent inode	Allocate an inode and fill it, set the flag that indicates it is a regular file
remove	Remove the entry in the parent directory, and update parent inode	Decrease the nlink of the file, if nlink reaches 0 and open-count is 1 then free the inode of the file
mkdir	Insert a new entry in the parent directory, and update parent inode	Allocate an inode and fill it, set the flag that indicates it is directory, and create a native file to store the entries for the new dir
rmdir	Delete the entry in the parent directory, and update parent inode	Decrease the nlink of the dir, if nlink reaches 0 and open-count is 1 then free the inode of the dir, and remove the corresponding native file

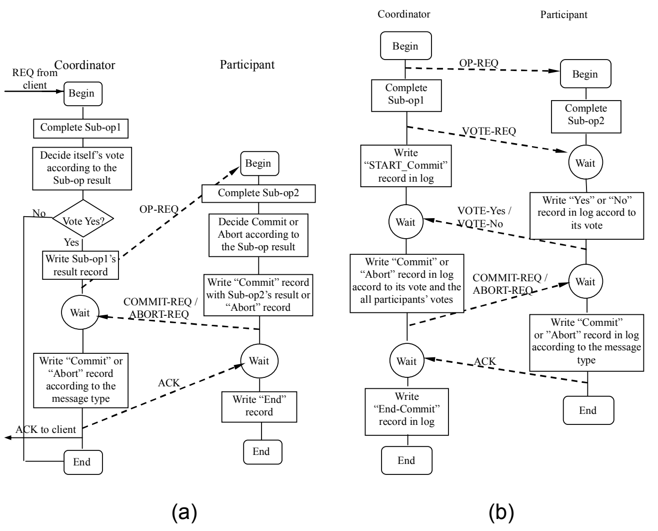

Coordinator
Participant
REQ from client
Begin
Complete Sub-op1
Decide itself's vote according to the Sub-op result
Vote Yes?
No
Yes
Write Sub-op1's result record
OP-REQ
Complete Sub-op2
Decide Commit or Abort according to the Sub-op result
Write "Commit" record with Sub-op2's result or "Abort" record
COMMIT-REQ / ABORT-REQ
Wait
Write "Commit" or "Abort" record according to the message type
ACK
Write "End" record
ACK to client
End
Coordinator
Participant
Begin
OP-REQ
Complete Sub-op1
Complete Sub-op2
VOTE-REQ
Write "START_Commit" record in log
Wait
Write "Yes" or "No" record in log accord to its vote
VOTE-Yes / VOTE-No
Write "Commit" or "Abort" record in log accord to its vote and the all participants' votes
COMMIT-REQ / ABORT-REQ
Write "Commit" or "Abort" record in log according to the message type
ACK
Write "End-Commit" record in log
End
(a)
(b)

Fig. 3. Metadata processing protocols: (a) The S2PC-MP protocol, (b) 2PC-base metadata protocol without improvement.

succeeds, it sends the operation request (OP-REQ) to the participant. This request not only tells what operation the participant should perform, but also implies that the coordinator has voted "yes". Therefore, the participant can decide whether the whole operation should be "committed" or "aborted" according to the results of Sub-op2. Then it sends its decision ("commit" or "abort") as well as the result through the message "Commit-REQ" or "Abort-REQ" to the coordinator. The coordinator writes "COMMIT" or "ABORT" record in its log according to this message, then sends an "ACK" message to the participant. After receiving the "ACK" message, the participant writes an "End" record in its log, and the whole processing is now completed.

For a distributed operation, S2PC-MP needs only three messages between the two servers, compared with five messages needed by Figure 3(b). Although the two sub-operations can be processed in parallel in Figure 3(b), the gain of such parallel is very limited, because both Sub-op1 and Sub-op2 typically cost only little processing time. Therefore, by reducing messages of each distributed operation, S2PC-MP not only reduces the response time of them, but also improves metadata throughput by greatly

[3] Operations that may require more than two metadata servers are *link* and *rename*. Howerver, both of them seldom occur in typical workloads [19, 21]. So they are handled in a different way.

reducing network traffic.

Moreover, the message waiting mechanism in S2PC-MP is different from that in the 2PC. In 2PC, the server does nothing when it is waiting for a specific message, and it ends the waiting if the message arrives or it time-outs. In order to better utilize the server's processing and storage resources, in S2PC-MP the server will process other requests during waiting by putting the half-completed request into a *waiting queue*. When the message which it has been waiting arrives, it will take the half-completed request out of the waiting queue, and continue processing it. This optimization is shown in Figure 4.

If some metadata server crashes, some waiting messages may never arrive, resulting in some requests staying in the waiting queue forever. Therefore, S2PC-MP needs to handle these waiting requests in its failure recovery protocol, which will be discussed in Section4.4.

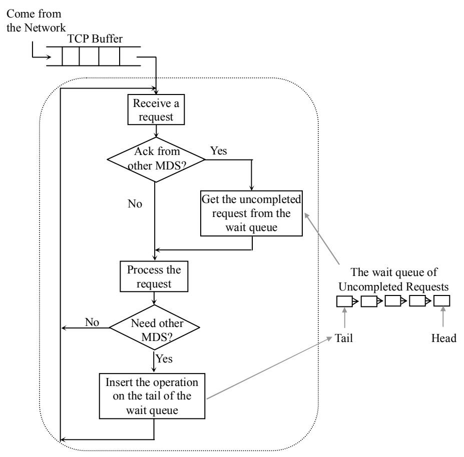

Come from the Network
TCP Buffer
Receive a request
Ack from other MDS?
Yes
No
Get the uncompleted request from the wait queue
Process the request
Need other MDS?
No
Yes
Insert the operation on the tail of the wait queue
The wait queue of Uncompleted Requests
Tail
Head

Fig. 4. Message Waiting Mechanism in S2PC-MP.

4.3 Metadata Log

4.3.1 Log Records

By combining operation result records with commitment progress records, there is a total of four types of records in S2PC-MP protocol. They are

- *Operation result record*. It not only contains the result of correspond operation, but also indicates the beginning of a distributed operation.
- *Commit record*. For the coordinator, it also indicates the end of the operation.
- *Abort record*. For the coordinator, it also indicates the end of the operation.
- *End record*. It is the end of the operation, and only appear in the participant.

For failure-free execution of a distributed operation, there are two records for each distributed operation in the coordinator's log file. They are (1) Sub-op1's result record, and (2) Commit or Abort record. And there are also two records in the participant's log file. They are (1) Commit record with Sub-op2's result, or Abort record, and (2) End record. Each of the above records should be written through to disk.

During recovery of a distributed operation, the system needs to know the state of the operation on the collaborated metadata servers. As a result, we need to carefully design the format of the log records such that the collaborated metadata servers can recognize the related log records of the same operation easily. In our system, each operation is uniquely identified by an ID which is the coalescence of the coordinator's ID and the operation's sequence number assigned by the coordinator. Therefore, we can easily match the log records on different metadata servers by operation IDs.

4.3.2 Write-back of Log Records

Every modification, either an ordinary operation or a distributed operation, has its log records. And these records must be written back to a log file on disk before the corresponding metadata are written back to disk.

Each log record for a distributed operation must be written to disk synchronously. However, motivated by improving performance, log records for ordinary operations are written back to disk asynchronously. In S2PC-MP, each metadata server keeps a log record buffer in its memory. Log records are first written into this buffer, and then the buffer is written back to disk as a whole when the following three types of events occur.

1. The time to write back is come, which is known as periodical write back.
2. The log buffer is full, which is known as forced write back.
3. The last record written into the buffer is a log record of a distributed operation, which is known as protocol write back.

Thus, if a metadata server crashes or powers off, the records in its log buffer will be lost. However, these records correspond to ordinary operations. Their loss will not affect the recovery of metadata consistency. Therefore, the metadata on all servers will recover to a consistent state. Although results of some latest ordinary operations may be lost, the effect is the same as that of other file systems using metadata journaling technology. One solution to avoid data loss is to keep the log buffer on a nonvolatile storage device, such as NVRAM.

4.3.3 Pruning of Log Records

Like other file systems using metadata journaling, the size of the log file is fixed, 50MB by default. And the log file is written sequentially. Log records of an ordinary operation can be pruned when related metadata are written from the metadata cache to the disk. On the other hand, log records of a distributed operation can be pruned when related metadata are written back to disk and the end record (the "Commit" or "Abort" record for coordinator, and the "End" record for participants) is presented in the log file. In S2PC-MP, log records are periodically pruned when metadata are periodically written back to disk or when the log file is full. For the latter case, the log buffer must be written back before metadata written back.

However, in the course of pruning the log file, metadata servers may encounter some distributed operations whose end records have not been written back in log file yet. These log records are preserved and moved to the head of the log file.

4.4 Recovery Protocol

The recovery process starts when the failure detection subsystem confirms a crash. After the failed metadata server reboots, it informs all other metadata servers to go into the recovery state, in which they do not accept any new coming requests and write their log buffer to disk. The server then reads its log file, and recovers according to log records. The result of any ordinary operation is recovered by its records, while the result of any distributed operation is recovered according to steps described in Section 4.4.1 and Section 4.4.2 accordingly.

Although all metadata servers must stop processing client requests when a failed metadata server restarts, the period of pause processing is very brief, typically tens of seconds, as showed in Section 5.2.2. The S2PC-MP recovery protocol enables fast failure recovery.

4.4.1 Recovery for the Coordinator

If the current record in the log is an operation result record, and the rebooted server is the coordinator of the distributed operation, the server tries to search for other records of this operation in the log file.

If it finds the Commit record of this operation, this means the operation was committed finally. It then writes the operation result to disk, and sends an "ACK" message to the participant. Receiving the "ACK" message, the participant will look up the related request in the waiting queue and delete it from the queue if finding it, and write the End record in the log file.

If it finds the Abort record of this operation, this means the operation was abort finally. So it will not write back the result. And it also sends an "ACK" to the participant.

If it finds neither of above records, it does not know what the final decision is. So it sends a "Decision-REQ" message to the participant to ask for the final decision, and waits for the answer by blocking itself. After receiving the "Decision-REQ", the participant will look up the records of this operation and send a reply message back. If the participant answers that it has the Commit or Abort record, then the rebooted server writes the Sup-op1's result to disk or ignores Sup-op1's result, respectively, and sends an "ACK" to the participant. If the participant answers that it has only Sub-op2's record and no any other records, then the rebooted server ignores Sub-op1's result because neither results has been written to disk. If the participant answers that it has no records of this operation, then the rebooted server ignores Sub-op1's result because the participant never handled related request.

4.4.2 Recovery for the Participant

If current record is an operation result record, and the rebooted server is the participant of the distributed operation, it tries to search for other records of this operation in its log file.

If it finds the End record, this means the operation is committed or aborted and there must be a Commit or Abort record too. So it writes Sub-op2's result to disk if it has the Commit record, or ignores Sub-op2's result if it has the Abort record.

If it finds the Commit record, but does not find the End record, then it sends a "Commit-REQ" to the coordinator, writes Sub-op2's result to disk, and waits for the coordinator's "ACK" message by blocking itself. After receiving the "Commit-REQ", the coordinator looks up the request related to the operation in the waiting queue and deletes it if finding it, writes the Commit record in the log file and sends an "ACK" to the rebooted server. When the rebooted server receives the "ACK" message, it completes the recovery of this operation and goes on to process the next record.

If it finds the Abort record, but does not find the End record, or it finds only Sup-op2's record, but does not find the other records, then it sends an "Abort-REQ" to the coordinator, and waits for the coordinator's "ACK" message by blocking itself. Afterwards the coordinator and the reboot server do the same as the above case.

4.3.3 Additional Steps

After all the records of the rebooted server's log file are properly handled according to Section4.4.1 and Section4.4.2, additional steps are required to recover other metadata servers' state.

The rebooted server will send a "Recover-END" message to all other servers to indicate that it has completed recovery. And it waits for response messages from all other servers. When it receives them, it goes into normal state and can handle client requests now.

When receiving the "Recover-END" message, the other servers should first do the following two things. (1) They parse their log files, and undo the result of those operation result records which has not any other records, and the peer server of which is the failed server. In this case, the failed MDS is the participant of these operations. However, the participant crashed before handling the requests about these operations. So its log file has no records about these operations, while the other MDSs have only Sup-op1 result records in their log files. This step is to undo these Sub-op1s' results. (2) They search their waiting queues for the requests that are sent to the failed server, and delete them from the queue.

After completing the above two steps, these servers send a response message to the rebooted server. And then they exit the recovery state and go back to the normal state, and continue to process requests from clients.

4.5 Validation

In S2PC-MP, every distributed operation is processed by a sequence of steps in a time order. As showed in Figure 3(a), these steps are: (1) the coordinator performs Sub-op1. The completion of this step is showed by Sub-op1's result record in the coordinator's log file; (2) the participant performs Sub-op2. The completion of this step is showed by Sub-op2's result record in the participant's log file; (3) the participant makes the final decision. The completion of this step is showed by either the "Commit" record or the "Abort" record in the participant's log file; (4) the coordinator commits or aborts according to final decision. The completion of this step is showed by either the "Commit" record or the "Abort" record in the coordinator's log file; (5) the participant completes the operation. The completion of this step is showed by the "End" record in the participant's log file.

It is easy to prove that if a distributed operation follows S2PC-MP showed in Figure 3(a), the two servers can reach to consistent results through the recovery protocol in Section4.4, whichever server crashes in the middle of the operation. Therefore, the S2PC-MP as well as the recovery protocol can ensure metadata consistency. We omit the proof of this conclusion, because it is beyond the scope of this paper.

5 Performance Evaluation

In this section, we evaluate the performance of the DDG policy and the S2PC-MP protocol using detailed simulation and actual measurements.

5.1 Performance of DDG

We evaluate the metadata processing performance of DDG from three aspects. We first conducted extensive simulation study to investigate its performance under different granularities, and compare them with two baseline policies, the *Random* policy and the *Subtree* policy. We then evaluate its scalability as the number of metadata servers increases. Finally, to validate our simulation results, we implemented DDG in a real system, and conducted detailed and extensive measurements.

The DDG and the two baseline policies are implemented in the same simulator (*simu_mds*, see Section 5.1.3) and the real system (DCFS2, see Section 5.1.5). The only difference between them is the way to select a metadata server for a new object. In the *Random* policy, each MDS chooses an MDS for a newly created object in a round-robin fashion. And in the *Subtree* policy, the MDS that maintains the root directory chooses a MDS for each object created in the root directory in a round-robin fashion, and objects in directories other than the root directory are assigned to the MDS that maintains their parent directory.

5.1.1 The Test Namespaces

We used two namespaces in our performance tests. One of them, denoted by *Namespace1*, is a snapshot of the home directory on the NFS server of our laboratory. The snapshot, taken in early 2005, contains 698,320 files and 25,864 directories. Because *Namespace1* is from a real environment, it can represent the file systems of the similar environments. However, *Namespace1* is too small compared with current and future file systems. Therefore, we generated a larger namespace, denoted by *Namespace2*, with similar characteristics as *Namespace1* by an automatic tool. It has 1,010,000 files and 34,721 directories. The characteristics of these two namespaces are showed in Fig. 5.

The two namespaces have similar characteristics, including: (1) Over 70% directories and files are located in several continuous levels; (2) Over 92% directories have 0 to 4 child directories; (3) Over 86% directories have 0 to 32 child files, except that the numbers of child files in *Namespace1* are distributed over a much wider range. Moreover, the distribution of directory sizes in these two namespaces matches with recently published data[33] over 13 file systems from 5 supercomputing sites and two file systems from department file servers. Dayal found that 90% directories have less than 32 entries (including both child files and child directories) for 11 file systems[33].

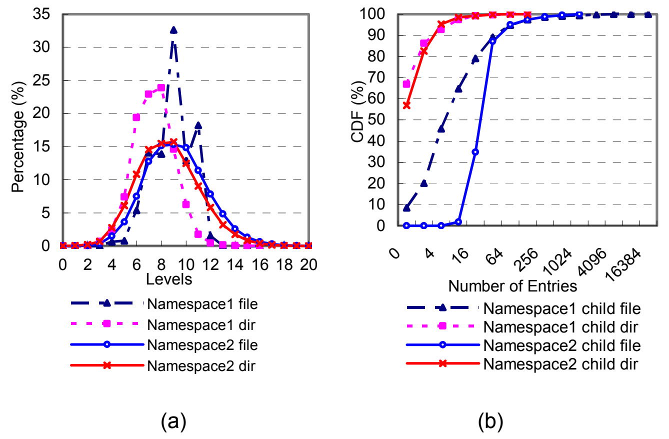


Fig. 5. (a) Percentage of files and directories in each level for *namespace1* and *namespace2*. (b) Cumulative distribution function of number of child directories and child files of each directory.

5.1.2 Benchmark Programs

We used two benchmark programs to evaluate our designs. *Macrobenchmark1* is a metadata-intensive benchmark, which takes a namespace as the input. It first recreates the hierarchy and all the files according to the input namespace. It then checks the attributes of all files and directories through the *stat* system call. Finally it deletes all the files, as well as the whole hierarchy. The time spent in each phase is measured. *Macrobenchmark1* allows multiple test processes running on multiple nodes, and each process deals with a disjoint part of the namespace.

Macrobenchmark2 is another multi-process program simulating a real environment with plenty of concurrent processes performing a large number of I/O operations. Each of these processes randomly selects files in the namespace and performs certain operations on these files, so they may access the same file or same directory. The

TABLE 2
OPERATIONS COMPOSITION OF *MACROBENCHMARK2*

Operations	Percentage (%)	Operations	Percentage (%)
Create	5.55	Stat	9.98
Mkdir	1.71	Readdir	7.70
Open	35.45	Remove	1.07
Read	25.67	Rmdir	0.04
Write	12.83		

TABLE 3
TOTAL NUMBER OF OPERATIONS AND THE AMOUNT OF DATA OF *MACROBENCHMARK2*

Number Of Clients	Total number Of Operations	Total Write Size (MB)	Total Read Size (MB)
1	243398	1205	3437
2	486933	2344	6364
3	730842	3803	9309
4	974430	4964	10992
5	1218173	5651	15010
6	1461699	6959	18610
8	1949044	13931	23049

benchmark is both metadata-intensive and data-intensive, generating a lot of file read and write operations. The percentages of each type of operations issued by *Macrobenchmark2* are given in Table 2, which reflects workloads similar to SPECsfs3.0 [21]. *Macrobenchmark2* also takes a namespace as the input, and consists of 3 phases: the warm-up phase, the testing phase and the post-processing phase. In the warm-up phase, it rebuilds the hierarchy according to the input namespace. In the test phase, it issues a set of test operations specified in a description file. In the post-processing phase, it calculates and outputs analysis statistics. The number of operations issued by all client nodes, as well as the total amount of data read or written by all client nodes, is listed in Table 3.

5.1.3 Simulation Results

In order to compare the performance of different distribution policies, we implemented the two baseline policies (*Random*, *Subtree*) as well as DDG in a metadata server simulator (*simu_mds*), which is a user-level process. We use *n* processes to simulate *n* metadata servers. The simulation experiments ran *Macrobenchmark1*, which took both *Namespace1* and *Namespace2* as the data sets. We simulated a system with 16 metadata servers and 32 client nodes. In these tests, each *simu_mds* has an inode cache that keeps 128k inodes and a dentry cache that keeps 128k directory entries. Both of them use the LRU algorithm for cache replacement.

The results in Figures 6 to 12 were obtained by running *simu_mds* on two nodes, with all clients on one node and all metadata servers on the other node. Each node has 2 quad-core processors (Intel Xeon E5335 @2.0GHz), 8 GB memory, and 3 146GB SCSI disk (Seagate ST3146707LC). The operating system is CentOS4.4. The two nodes are interconnected by a Gigabit Ethernet network.

The normalized throughputs (which use the throughput of *Subtree* as the baseline) of file creation and deletion under different granularities are showed in Figures 6 to 8. *Random* has the lowest throughput, because it has the largest number of branch points. *Subtree* has the fewest number of branch points, which means it has very few metadata log writes. The creation throughput of *Subtree* is over 8 times that of *Random*, and its deletion throughput is over 6 times that of *Random*. However, since the metadata is very unevenly distributed among the metadata servers in *Subtree*, some individual metadata servers have a large number of disk reads and writes because the number of objects on it is much larger than the metadata cache. DDG policies with proper granularity can achieve a better balance between distribution and branch points, and hence their creation and deletion throughputs are 1.4 to 7 times that of *Subtree*, and 1.5 to 63 times that of *Random*.

As Figure 6 shows, for DDG policies, the throughputs of both creation and deletion increase with FileWid. Smaller FileWid results in more branch points, and hence more disk writes for metadata log and more messages among metadata servers. DDG outperforms *Subtree* until FileWid is larger than or equal to 4096 for *Namespace1* and 512 for *Namespace2* respectively. From the profiling data, we found that in *Namespace1*, about 93.45% of files are in directories with less than 4096 child files. Similarly, in *Namespace2*, about 89.25% of files are in directories with less than 512 child files. These results indicate that from the perspective of file creation and deletion throughput, it is preferable to set *FileWid* to the value such that over 90% of files that are in the directories each of which has no more than *FileWid* number of files.

In Figure 7, we can see that for *Namespace1*, DirWid has less effect on throughputs than FileWid. Since branch points consist of branch files and branch directories, the proportions of branch directories to total branch points is affected by both FileWid (which determines the number of branch files) and DirWid (which determins the number of branch directories). Becasue FileWid was set to 2048, the branch directories account for only 1/4 to 1/8 of total branch points in *Namespace1*, and hence the number of branch points changes little with different DirWid values. On the other hand, due to setting FileWid to 2048, in *Namespace2* the branch directories account for 100% of branch points. So DirWid has some obvious effect on throughputs for *Namespace2*. Throughputs increase with the increase of DirWid, but decline slightly when DirWid is larger than or equal to 32, because uneven distribution causes more disk reads and writes. These results indicate that from the perspective of file creation and deletion throughput, DirWid is not as important as FileWid, and a value around 16 is preferable for the namespaces in which 90% directories are in the directories that have no more than 32 child directories.

As showed in Figure 8, throughputs first increase slightly until DirDep gets 6, and then decline steeply. This is because that a larger DirDep value, like *Subtree*, may cause much skewed metadata distribution among meta-

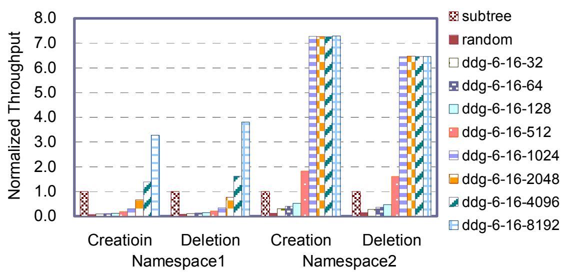


Fig. 6. Normalized metadata throughput for file creation and deletion with different FileWid (*macrobenchmark1*)

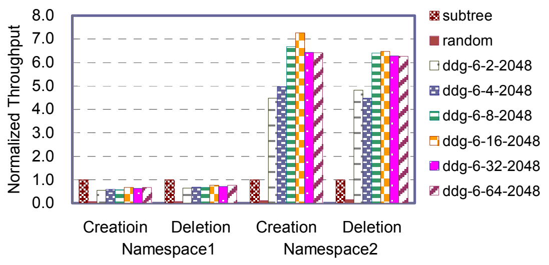


Fig. 7. Normalized metadata throughput for file creation and deletion with different DirWid (*macrobenchmark1*)

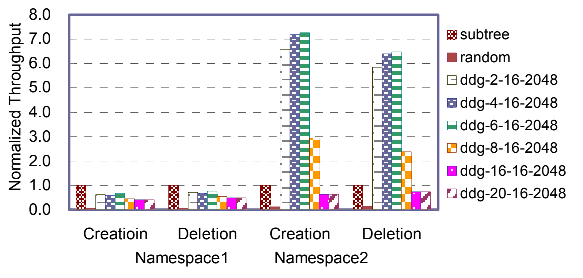


Fig. 8. Normalized metadata throughput for file creation and deletion with different DirDep (*macrobenchmark1*)

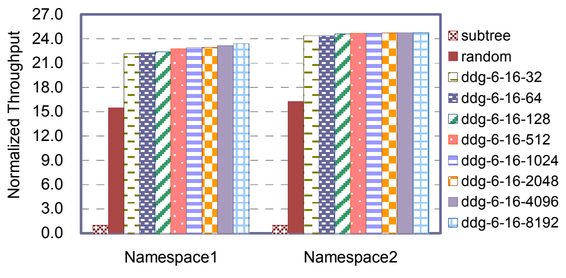


Fig. 9. Normalized throughput for file stat operation with different FileWid (*macrobenchmark1*).

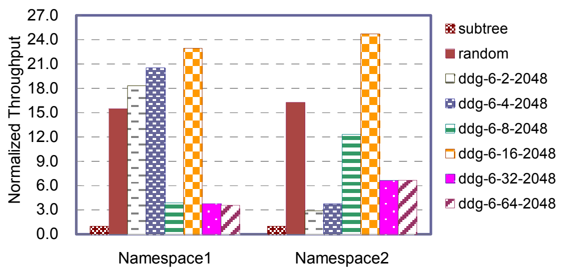


Fig. 10. Normalized throughput for file stat operation with different DirWid (*macrobenchmark1*).

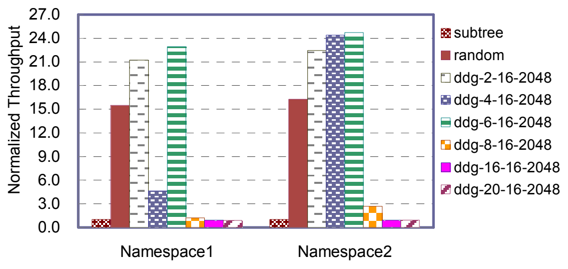


Fig. 11. Normalized throughput for file stat operation with different DirDep (*macrobenchmark1*).

data servers, resulting in extensive disk reads and writes on individual metadata servers. These results indicate that small DirDep values such as 2-6 are preferable.

The throughputs of file stat are showed in Figures 9 to 11. These figures show very different results with that of file creation and deletion. Since file stat is a read only operation, there is no 2PC log, as a result, *Subtree* and DDG with large DirDep perform worst. And benefited from even metadata distribution, *Random*'s throughput is over 15 times that of *Subtree*. DDG policies with proper granularity outperform *Random* by about 10% to 50%, because they balance the distribution and namespace locality and hence have 3 to 10 times less inter MDS messages than *Random*. And DDG policies with proper granularity can achieve 3 to 24 times performance improvement for file stat over *Subtree*.

Figure 9 shows that stat throughputs are not sensitive to FileWid. This is very different from file creation and deletion. We believe the reason is that each metadata server has a large metadata cache, which can hold 128k objects, and the distribution skew caused by FileWid is not beyond limit of the cache.

As Figures 10 and 11 show, DDG has very poor stat performance for both large DirWid and DirDep, because of uneven metadata distribution. This is similar with the file creation and deletion. When choosing DDG values, it is very important to keep this in mind.

5.1.4 Scalability Analysis

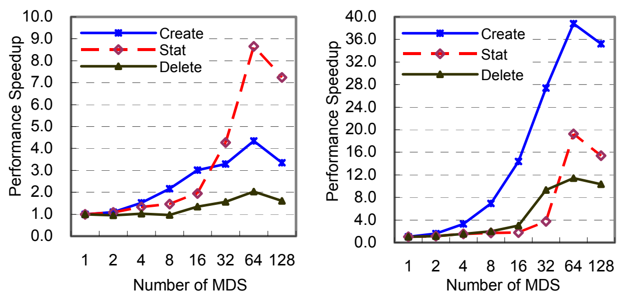
The scalability of metadata processing is an important consideration for a cluster file system designed for large systems. For distributed metadata processing, scalability lies in two aspects: (1) Server scalability, which is how the system performance under fixed workloads changes as the number of metadata servers increases; (2) Client scalability, which is how the system performance changes as the number of clients, hence the workload, increases.

In this section, we discuss the server scalability of DDG found in the simulation experiments. Figure 12 shows the throughput speedups of ddg-2-16-2048 with 128 client processes and 1 to 128 metadata servers. Note that the 128 client processes were constantly feeding requests to servers at a very high rate, so the aggregated request rate was equivalent to that generated by a much larger group of client nodes in a real system, where clients do not perform file I/O constantly. As shown in the Figure 12, performance speeds up well for *Namespace2*, but does not speed up well for *Namespace1*. Through profiling, we found that *Namespace1* has over 42 thousand branch files, while *Namespace2* has no branch files. As a result, *Namespace1* has a large number synchronous log writes than *Namespace2*, which counteracts some of the gains by adding more MDS. Through profiling, we also found that for *Namespace1*, metadata distribution is more uneven with the increase of the number of metadata servers, which also impairs its speedup.

5.1.5 Results on the Real System

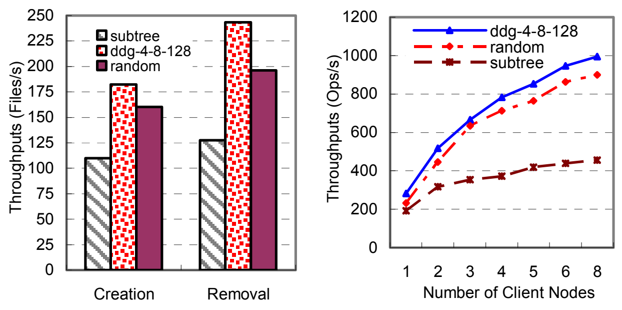
In addition to simulation experiments, we also implemented the three policies in DCFS2, and measured their performance on a real system. DCFS2 is a cluster file system we designed for very large-scale, supercomputer-class clusters. It consists of a storage space manager, a group of metadata servers, a group of IP-addressable storage servers (called *SuperNBD servers*) for file data storage, and a large number of client nodes.

The tests were done on 17 nodes of a very large cluster, each of which has two AMD Opteron™ 242 processors,



(a) *Namespace1* (b) *Namespace2*

Fig. 12. Scalability of metadata servers (*Macrobenchmark1*)



(a) *Macrobenchmark1* (b) *Macrobenchmark2*

Fig. 13.Throughputs on the real system.(*Namespace1*)

2GB of memory and a 37GB Seagate ST373307LC SCSI disk, running Turbolinux 3.2.2. The nodes were interconnected by a Gigabit Ethernet. Because of the space limitation, we only present the results of DDG-4-8-128, and compared them with those of *Random* and *Subtree*.

Our goal is to measure the performance, interactions, and data distributions between multiple metadata servers. As a result, while this is a relatively small test platform, we configured DCFS2 to run six metadata servers, one storage space manager, two *SuperNBD* servers and eight client nodes. While we did not run the experiments on a full production system with thousands of nodes, we believe that our results are still valid for large systems. We are only interested in metadata server performance, and the six metadata servers (which can be further scaled up, as indicated in the above discussion) can easily support a large number of client nodes and disks. In these tests, each MDS used about 800MB memory as its metadata cache.

Figure 13(a), showing the results of *Macrobenchmark1*, confirms that DDG significantly outperforms the baseline systems. For file creation, DDG is 66% and 15% faster than *Subtree* and *Random*, respectively. For file deletion throughput, DDG is 91% and 24% higher than those of *Subtree* and *Random* respectively.

Figure 13(b) compares the aggregate throughputs of all clients running *Macrobenchmark2* on *Namespace1*. While this benchmark is both metadata-intensive and data-intensive, DDG still shows the highest throughputs, about 100% higher than that of *Subtree*, and 10% higher than that of the *Random*.

Through careful analyses [28], we found that although *Subtree* issued the fewest number of requests, the distribution of these requests was very unbalanced, resulted in the lowest aggregate throughput. *Random*, on the other hand, produced the most balanced the metadata distribution. However it also generated 50% more branch points and 27% more requests than DDG did. DDG achieved the best performance by striking a right balance between the number of requests and the balance of distribution.

5.2 Performance of the S2PC-MP

The S2PC-MP was implemented in DCFS2. We evaluated the performance of the protocol by measuring the overhead of writing log records required by 2PC, as well as the efficiency of recovery from a metadata server crash.

The test platform consisted of 20 nodes, each of which is the same as those described in Section5.1.5. DCFS2 was configured with two metadata servers, one storage space manager, one *SuperNBD* server and 16 client nodes.

5.2.1 Overhead of Metadata Logging

We measured the cost of metadata logging of S2PC-MP by comparing the read/write bandwidth, the file creation throughput, the file deletion throughput and the transaction throughput with and without writing logging.

Iozone [22], running in the cluster mode and the stonewall mode, was used to measure the read and write performance. The total amount of data read/written was 528MB, and the data size of each read/write operation was 1MB. As shown in Figure 14, except in a few cases, there was no obvious different between the aggregate read/write bandwidth with and without writing logging. DCFS2 successfully minimized the logging overheads through many optimization methods, such asynchronous writing log records; asynchronous updating of file size and *mtime* in write operations, and no updating of *atime* in read operations.

Postmark [17] was used to measure the file creation throughput, the file deletion throughput and the transaction throughput, and results are shown in Figure 15. Each client node ran a postmark process. They created or deleted 48,000 files in 48 directories, and performed 96,000 transactions. In the cases of file creation and deletion throughputs, turning on write logging decreased performance by only 10%. For the transaction throughput, there was no obvious difference between turning the write logging on and off.

5.2.2 Efficiency for Recovery

For S2PC-MP, the time to recover from a metadata server failure is independent of the file system size, which may be large and in PB-scale. The recovery time is determined by the size of the log file, and by the number of uncompleted distributed operations in the log file, whose results need to be restored by communicating with other metadata servers.

Table 4 shows the recovery time for different log file sizes. In all these tests, each log file had 721 uncompleted distributed operations, regardless of its size. As a result, we can isolate the effect of the log file sizes to the recov-

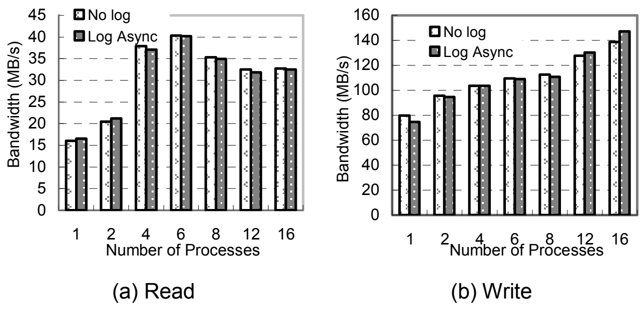
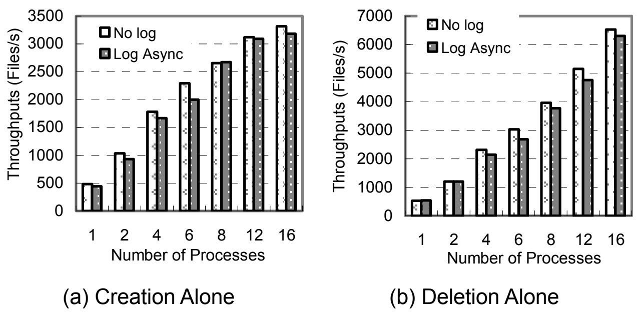


Fig. 14. Read and write bandwidth.



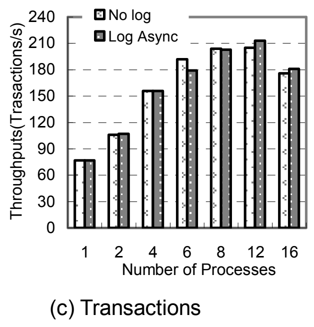


Fig. 15. Throughputs by PostMark.

ery time. As expected, the recovery time was within tens of seconds, and was proportional to the log file's size.

Table 5 compares the recovery times of different numbers of distributed operations in the log file, while fixing the log file size at 10MB. It is clear that the recovery process is very efficient. When the number of uncompleted distributed operations increases 10 times, the recovery time increased only 1 time.

6 Conclusion

Most supercomputers nowadays are based on large clusters. As their capacity and user requirement increase, so is the demand for high performance distributed file systems on these clusters. While traditional metadata management techniques such as single metadata server or simple distribution algorithm maybe adequate for small to medium systems, large scale high performance cluster systems call for more sophisticated decentralized metadata processing technology.

This paper presents a set of solutions to two fundamentally important issues in decentralized metadata processing: (1) how to distribute the objects in the namespace across metadata servers, which affects the metadata throughput, and (2) how to keep the cross-metadata server operations consistent in the presence of server failures, which determines the reliability and availability of the file system.

TABLE 4
Recovery time for different size of log files

Log File Size (MB)	Number of Log Records	Recovery Time (s)
10	71998	4.14
20	143607	7.87
30	215332	11.65
40	287087	15.73
50	359044	19.26

TABLE 5
Recovery time for different number of uncompleted distributed operations

Number of Log Records	Number of Uncompleted Distributed Op.	Percentage （%）	Recovery Time (s)
71763	0	0	4.12
71998	720	1	4.14
72262	1446	2	4.39
72508	2178	3	4.64
72760	2912	4	4.89
73019	3655	5	5.33
74338	7440	10	9.26

Our Dynamic Dir-Grain (DDG) policy dynamically partitions the namespace into hierarchical units according to a triple-defined distribution granularity, and achieves a good balance between maintaining namespace locality and equal distribution, which is important to achieve high metadata throughput. Extensive simulation and measurement results show that DDG policies with proper granularity significantly outperform the *Random* policy and the *Subtree* policy by 40% to 62 times.

We proposed a technique called S2PC-MP for consistency of cross-server operations. It reduces overhead in normal processing by combining commit operations with metadata operations, and can quickly recover metadata consistency after any server crashes. Our performance results show that the creation and deletion throughputs decreases only about 10% and recovery can be completed within tens of seconds.

We implemented DDG algorithm and S2PC-MP protocol in DCFS2, a large scale cluster file system for supercomputers. However, these methods can also be used in other cluster file systems which provide a hierarchical namespace to users.

Acknowledgment

The authors wish to thank Prof. Ninghui Sun, Prof. Dan Meng, Dr. Sining Wu, Dr. Jie Ma, and many other colleagues for their discussion and helpful suggestions. The authors wish to thank the Scientific Computing Platform of ICT for providing its namespaces to us. This work was supported in part by the Natural Science Foundation of China under grant numbered 60970025, and the National High-Technology Research and Development Program of China under grant numbered 2006AA01A102, 2009AA01Z139 and 2009AA01A129. Yiming Hu is supported by National Science Foundation under grant No. CCF-0541103.

REFERENCES

[1] D.C. Anderson, J.S. Chase, and A.M. Vahdat, "Interposed Request Routing for Scalable Network Storage," *ACM Trans. Computer Systems (TOCS)*, vol. 20, no. 1, pp. 25-48, Feb. 2002.

[2] M. Baker and J. Ousterhout, "Availability in the Sprite Distributed File System," *ACM SIGOPS Operating Systems Review*, vol. 25, no. 2, pp. 95-98, Apr. 1991.

[3] M. Baker, "Fast Crash Recovery in Distributed File Systems," PhD. dissertation, Dept. of Computer Science, Univ. of California, Berkeley, Calif., 1994

[4] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publication Company, 1987

[5] P.J. Braam, "The Lustre Storage Architecture," White Paper, Cluster File Systems, Inc., Oct. 2003

[6] S.A. Brandt, L. Xue, E.L. Miller, and D.D.E. Long, "Efficient Metadata Management in Large Distributed File Systems," *Proc. 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST2003)*, pp. 290–298, Apr. 2003.

[7] A.R. Butt, T.A. Johnson, Y. Zheng, and Y.C. Hu, "Kosha: A Peer-to-Peer Enhancement for the Network File System," *Proc. 2004 IEEE/ACM High Performance Computing, Networking and Storage Conference (SC'04)*, Nov. 2004.

[8] M. Devarakonda, B. Kish, A. Mohindra, "Recovery in the Calypso File System", *ACM Trans. Computer Systems*, vol. 14, no. 3, pp. 287-310, Aug. 1996.

[9] J. Gray, "Notes on Data Base Operating Systems," *An Advanced Course: Operating Systems*, R. Bayer, R. M. Graham, and G. Seegmüller, eds, Lecture Notes on Computer Science Series, vol. 60, Berlin: Springer-Verlag, pp. 393-481, 1978.

[10] T. Haerder, A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys (CSUR)*, vol. 15, no. 4, pp. 287-317, Dec. 1983.

[11] M. Ji, E.W. Felten, R. Wang, and J. P. Singh, "Archipelago: An Island-Based File System for Highly Available and Scalable Internet Services," *Proc. 4th USENIX Windows Systems Symposium*, Aug. 2000.

[12] C. Karamanolis, L.Liu, M. Maholingam, D. Muntz, and Z. Zhang, "An Architecture for Scalable and Manageable File Services," Technical Report HPL-2001-173, HP Labs, Jul. 2001

[13] J. Katcher, "Postmark: A New File system Benchmark," Technical Report TR-3022, Network Appliance, Corp., Sunnyvale, 1997

[14] J. Menon, D.A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM Storage Tank — A Heterogeneous Scalable SAN File System," *IBM Systems Journal*, vol. 42, no. 2, pp. 250-267, Apr. 2003.

[15] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale Storage Cluster — Delivering Scalable High Bandwidth Storage," *Proc. 2004 IEEE/ACM High Performance Computing, Networking and Storage Conference (SC'04)*, Nov. 2004.

[16] W.D. Norcott, "Iozone File System Benchmark," 2005, available at http://www.iozone.org/docs/IOzone_msword_98.pdf

[17] K.W. Preslan, A. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. Van Oort, D. Teigland, M. Tilstra, and M. O'Keefe, "Implementing Journaling in a Linux Shared Disk File System," *Proc. 8th NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the 17th IEEE Symposium on Mass Storage Systems*, pp. 351-378, Mar. 2000.

[18] PVFS2 Development Team, "Parallel Virtue File System, Version 2," Sept. 2003, available at http://www.pvfs.org/pvfs2/

[19] D. Roselli, J. Lorch, and T. Anderson, "A Comparison of File System Workloads," *Proc. 2000 USENIX Annual Technical Conference*, pp. 41-54, Jun. 2000.

[20] F. Schmuck, and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," *Proc. First USENIX Conference on File and Storage Technologies (FAST02)*, Jan. 2002.

[21] SFS3.0 Documentation Version 1.0. Standard Performance Evaluation Corp. (SPEC), Warrenton, VA

[22] L. Shepard, and E. Eppe, "SGI InfiniteStorage Shared Filesystem CXFS™: A High-Performance, Multi-OS SAN File System from SGI," White Paper, Silicon Graphics, Inc. Jun. 2004

[23] M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems (2nd edition)*, Prentice-Hall, Inc., 1999

[24] U. Vahalia, *UNIX Internals: The New Frontiers*, Prentice-Hall, Inc., 1996.

[25] S.A. Weil, K.T. Pollack, S.A. Brandt and E.L. Miller, "Dynamic Metadata Management for Petabyte-scale File Systems," *Proc. 2004 IEEE/ACM High Performance Computing, Networking and Storage Conference (SC'04)*, Nov. 2004.

[26] C. Wu and R. Burns, "Handling Heterogeneity in Shared-Disk File Systems," *Proc. International Conference for High Performance Computing and Communications (SC'03)*, Nov. 2003.

[27] J. Xiong, S. Wu, D. Meng, N. Sun and G. Li, "Design and Performance of the Dawning Cluster File System," *Proc. 2003 IEEE International Conference on Cluster Computing (Cluster2003)*, pp. 232-239, Dec. 2003.

[28] J. Xiong, R. Tang, S. Wu, D. Meng and N. Sun, "An Efficient Metadata Distribution Policy for Cluster File Systems," *Proc. 2005 IEEE International Conference on Cluster Computing (Cluster2005)*, Sep. 2005.

[29] Z. Zhang and C. Karamanolis, "Designing a Robust Namespace for Distributed File Services," *Proc. 20th IEEE Symposium on Reliable Distributed Systems*, pp. 162-171, Oct. 2001.

[30] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System", *IEEE Computer*, vol. 21, no. 2, pp. 23-36, Feb. 1988.

[31] Y. Zhu, H. Jiang, J. Wang, and F. Xian, "HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems", *IEEE Trans. On Parallel and Distributed Systems*, vol. 19, no. 6, pp. 750-763, June 2008.

[32] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and Adaptive Metadata Management in Ultra Large-scale File Systems", *Proc. 28th International Conference on Distributed Computing Systems (ICDCS'08)*, pp. 403-410, June 2008

[33] S. Dayal, "Characterizing HEC Storage Systems at Rest", Technical Report of Parallel Data Laboratory at Carnegie Mellon University, CMU-PDL-08-109, July 2008

[34] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West, "The ITC Distributed File System: Principles and Design", *ACM SIGOPS Operating Systems Review*, vol. 19, no. 5, pp. 35-50, Dec. 1985.

[35] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless Network File Systems", Proc. *15th ACM Symposium on Operating Systems Principles*, pp. 109-126, Dec. 1995

[36] S. Patil, and G. Gibson, "GIGA+: Scalable Directories for Shared File Systems", Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-08-110, October 2008.

[37] J. Xing, J. Xiong, N. Sun, and J. Ma, "Adaptive and Scalable Metadata Management to Support A Trillion Files", *Proc. of the SC'09*, Nov. 2009.

[38] S. Sinnamohideen, R. Sambasivan, J. Hendricks, K. Liu and G. Ganger, "A Transparently-Scalable Metadata Service for the Ursa Minor Storage System", *2010 USENIX Annual Technical Conference (USENIX ATC'10)*, Jun. 2010

[39] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003, pp. 29-43

Jin Xiong received her BE degree from Sichuan University, China in 1990, ME degree from the Institute of Computing Technology (ICT), the Chinese Academy of Sciences (CAS) in 1993, and PhD degree from the Graduate University of Chinese Academy of Sciences in 2006. She is currently an associate professor at ICT, CAS. She is a member of IEEE CS. Her research interests include cluster file systems, high performance I/O systems and data management.

Yiming Hu received the PhD degree in electrical engineering from the University of Rhode Island in 1998. He received the BE degree in computer engineering from the Huazhong University of Science and Technology, China. He is an associate professor of computer science and engineering at the University of Cincinnati. His research interests include computer architecture, storage systems, peer-to-peer systems, operating systems, and performance evaluation. He has published papers in journals such as the IEEE Transactions on Computers and IEEE Transactions on Parallel and Distributed Systems (TPDS), and in conferences such as the ISCA, HPCA, SIGMETRICS, USENIX, FAST, IPDPS, ICPP, and ISLPED. He is a recipient of a US National Science Foundation CAREER Award. He is a senior member of the IEEE. He severs in the editorial board of IEEE TPDS.

Guojie Li received his BS degree from Peking University, China in 1968, ME degree from University of Science & Technology of China in 1981, and PhD degree in EE from Purdue University, USA in 1985. He was elected as a member of Chinese Academy of Engineering (CAE) in 1995 and a fellow of the Third World Academy of Sciences (TWAS) in 2002. He currently serves as professor and director of the Institute of Computing Technology, the Chinese Academy of Sciences, president of China Computer Federation, Editor-in-Chief of Journal of Computer Science and Technology, director of Information and Electronics Division of CAE, and vice director of IT technology and emerged industries division of the Advisory Committee for State Informatization. His current research interests include high performance computer architecture, and MPP systems for bio-information processing.

Rongfeng Tang is currently a PhD candidate at the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include cluster file systems and network storage systems.

Zhihua Fan received his BS degree from Tsinghua University, China in 2001, ME degree from the Graduate University of Chinese Academy of Sciences in 2008. He is currently working at NetEase.com, Inc.