Wofs: A Distributed Network File System Supporting Fast Data Insertion and Truncation

Cheng-Chia Wang and Yarsun Hsu
Department of Electrical Engineering,
National Tsing Hua University
Hsinchu, Taiwan
agigi@hpcc.ee.nthu.edu.tw, yshsu@ee.nthu.edu.tw

***Abstract*—Distributed file systems have become popular in recent years. However, they still lack functions for doing fast arbitrary data insertion and truncation. To solve the problem, we present Wofs, an object-based distributed network file system which supports fast arbitrary data insertion and truncation. Wofs splits a file into many small objects, stores these objects in remote file servers, and uses a special B+tree [1][4] to manage the metadata of these objects. Besides, Wofs uses the object-range locking policy to avoid data incoherence and improve performance.**

Keywords-file system; insert; truncate; object; B-tree

I. INTRODUCTION

With the development of high-speed networks, distributed file systems have become popular in recent years. More and more companies build their distributed file systems to provide commercial transactions and services.

Lately the concept of object-based storage has been brought up [7]. The main concept of object-based storage is to offload the space management component of an existing file system to the storage device itself. Application clients thus request for an object (or file) instead of many disk blocks. Some distributed file systems combined with object-based storage devices (OSDs), such as Ceph [14] and zFS [9], have also been presented.

But until now, there are still no file systems supporting fast arbitrary data insertion and truncation. In this paper we present Wofs, an object-based distributed network file system with support for fast arbitrary data insertion and truncation. Wofs splits a file into many small objects, stores these objects in remote file servers, and uses a special B+tree [1][4] to manage the metadata of these objects. Besides, Wofs uses the object-range locking policy to avoid data incoherence and improve performance.

II. RELATED WORK

As mentioned in Section I, fast arbitrary data insertion and truncation haven't been supported on general file systems, but have already appeared in database systems. EXODUS [2] is a classic model of object-oriented databases. But EXODUS is a local database system, not a distributed network file system.

B+tree is usually used to record the information of extents, or directory contents. Xfs [12] is a classic model of the file systems which utilize B+tree. It uses B+tree to record free extents, file extent maps, and directory contents. B+tree enables xfs to become more scalable and stable. However, xfs doesn't use B+tree to handle data insertion and data truncation. As far as we know, there has been no file system using B+tree to efficiently handle data insertion and data truncation so far.

Since there is still no real OSD manufactured and sold in stores, we modify two programs respectively called v9fs [13] and spfs [11] for the implementation of our Wofs. The two programs communicate with each other via the 9p protocol [6] to simulate OSDs. We also modify the 9p protocol to support data insertion and data truncation in Wofs.

III. DESIGN AND IMPLEMENTATION

A. The methods for data insertion and data truncation

Before we introduce the architecture of Wofs, we need to introduce how to do data insertion and data truncation in a file system first. As shown in Fig. 1, without supporting data insertion and data truncation like Wofs, a file system still can do data insertion by reading out all data after the specific offset, merging with to-be-inserted data, and then writing them back to the file. But this method for data insertion is very inefficient. The bigger the file is, the longer time the data insertion needs to take. Data truncation also needs to read out and write back data, so it has the same problem, too. So without a special mechanism, data insertion and data truncation can take a lot of time to do.

To solve this problem, we can just split a file into many fixed standard-sized chunks (objects) with a data structure managing them, and then do data insertion by just reading data out and writing back data as shown in Fig. 2. Thus, the amount we read out and write back is much smaller, and the time needed for data insertion will be much less, too. Data truncation works similarly. That is the main reason why we split a file into many objects in Wofs.

978-0-7695-4025-2/10 $26.00 © 2010 IEEE
DOI 10.1109/SNAPI.2010.13

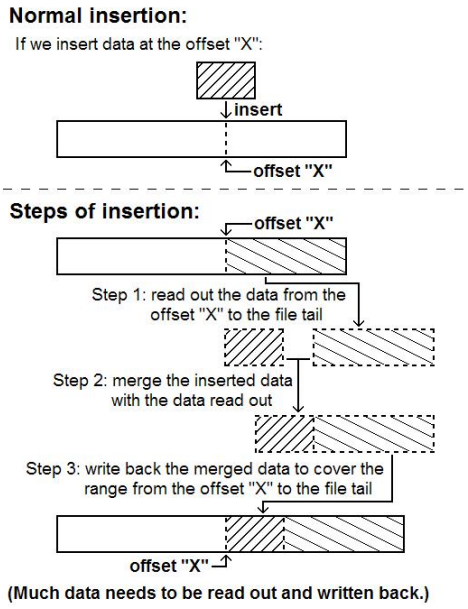

Figure 1. Normal insertion in common file systems.

In Wofs, we allow a user to insert the data of any size at any offset of a file. We also allow a user to truncate any range of a file. But to maintain the performance of data insertion and data truncation, we have to limit the object size in Wofs.

To limit the object size after data insertion, we have to split this inserted object into many smaller objects if the size of the inserted object is beyond the standard object size. Just as shown in Fig. 2, after Step 3 of insertion, the inserted object "A" becomes too large, so it must be split into 3 new objects "L", "M" and "N" to limit the sizes of all objects. Also, after data insertion or data truncation, sometimes the size of some object will become smaller than the standard object size, like the new object "N" in Fig. 2. To support this, Wofs is designed to allow a file to have many objects with various sizes, but it needs a well-designed architecture to support that.

B. The architecture of Wofs

Wofs has three components: one centralized metadata server (MDS), several clients and several OSD servers. The OSD server is the remote storage server of Wofs used to store objects. They are connected by a network. The architecture of Wofs is shown in Fig. 3.

As implied by the name, the centralized metadata server of Wofs stores all metadata in Wofs and is inquired for the metadata by clients. To shorten the time needed for getting metadata in MDS, MDS stores all metadata in memory and just stores the metadata in the disk at some fixed times.

The clients communicate with MDS for metadata, and then use the metadata to access OSD servers. In essence, they are Linux kernel modules with network functions in Wofs.

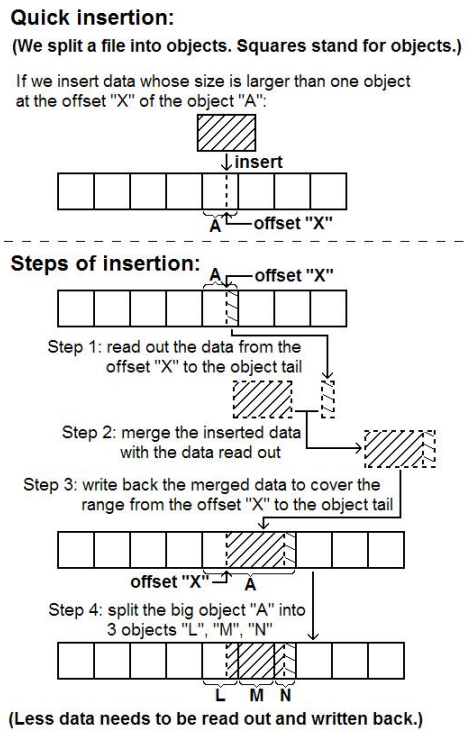

Figure 2. Quick insertion in Wofs.

The OSD servers are the remote storage servers of Wofs used to store objects. In Wofs, a file is split into many chunks, and we use OSD servers to store these chunks. In OSD servers, these chunks are called "objects". But because there is no real object-based disk commercially available, we use a program called "spfs" [11] to simulate an object-based device and call it an OSD server. So in fact, we use this program to control objects and store these objects in the form of files on ext3.

The communication among these three components in Wofs is shown in Fig. 3. In Wofs, MDS and OSD servers need not communicate with each other. They have no information needed to exchange with each other. Only clients exchange information with them to complete all work.

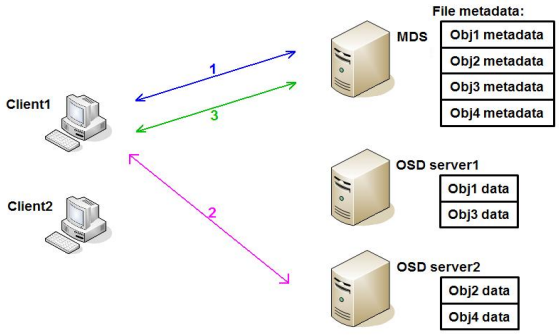

Figure 3. The communication for file access in Wofs.

C. The communication in Wofs

Wofs is a Linux file system, and we design it to support three file types: directories, files, and symbolic links. The content of a directory is the metadata of all files included in this directory, and the content of a symbolic link is the name or path of the directory or file it points at. Since the contents of directories and symbolic links are small, we see the contents as their metadata and just store them in MDS. Storing all data of directories and symbolic links in MDS is good for performance. Clients can know which file is included in a specific directory by just accessing MDS without accessing OSD servers. Besides, since MDS stores all metadata in its memory, we can more quickly get the contents of directories and symbolic links.

But files in Wofs are different. Because the data of a file is usually big, we need to store file data in disks. In Wofs, the data of a file is split into many objects and stored in OSD servers, but the metadata of that file is stored in MDS. The metadata of a file includes the information about which OSD servers objects are stored in. So clients need to access MDS, and then access OSD servers. If we choose to read this file, we cannot allow other clients to modify this file simultaneously. So in Wofs, MDS also takes charge of locking the metadata to avoid data incoherence.

As shown in Fig. 3, Wofs does communication three times to accomplish one file access. Clients do the 1st and the 3rd communication with MDS, and do the 2nd communication with OSD servers. At the 1st communication, a client delivers some information to tell MDS the file and the file range it wants to access. And then, MDS will find out which objects reside at that file range and lock the metadata of these objects. Then, MDS sends clients some information about which OSD servers these objects are stored in. At the 2nd communication, clients use this information to access correct OSD servers to get correct object data. At the 3rd communication, clients tell MDS the file access is done, and MDS will unlock or modify the metadata of these objects.

Whether this file access is read, write, insertion, truncation or deletion, MDS locks the metadata of objects at the 1st communication and unlocks it at the 3rd communication. During the whole period of file access, the metadata of the accessed objects is locked and protected. So we can ensure that data incoherence won't occur in Wofs.

D. MDS

As mentioned in Section III.B, MDS stores all metadata in memory to accelerate the inquiry of metadata. To manage all metadata efficiently, we use a simple hierarchical data structure to record the file hierarchy of Wofs. As shown in Fig. 4, the root of this hierarchical data structure records all system metadata about this file system. Then the root points to the metadata of the top directory. Finally, by traversing the hierarchical data structure, we can find the metadata of files we want to access.

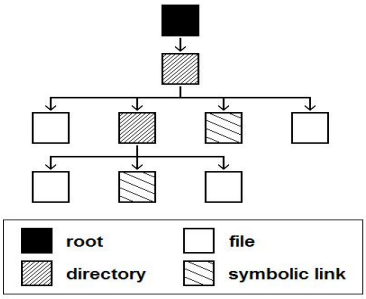

Figure 4. The hierarchical data structure used to record the file hierarchy of Wofs.

In this hierarchical data structure, the metadata of directories, files and symbolic links records different contents. The metadata of a directory records the memory address of the metadata of the directories or files belonging to this directory. The metadata of a symbolic link records the name or path of the directory or file it points at. And the metadata of a file is a data structure called bptree used to manage the metadata of all objects belonging to this file.

As mentioned in Section III.A, splitting a file into many chunks to do data insertion and data truncation is a practical way to reduce the time consumption. But since Wofs supports insertion and truncation, some objects will be added, deleted or truncated, so the metadata of these objects will be added, deleted or modified. In terms of the data structure, it is difficult to record and manage the object metadata efficiently.

For example, as shown in Fig. 5, if we use an array to manage the metadata of objects, this array will need another linked list to record the metadata of objects whose size is not a "standard size". Since Wofs limits the maximum of the object size to maintain the performance of insertion and truncation as mentioned in Section III.A, we call this limited object size "standard size" and call the object whose size is not a standard size "nonstandard object" for convenience. If a file has no nonstandard objects, we can find out the specific object based on a specific offset by just dividing the offset by the standard size. But if a file has one or more nonstandard objects, dividing the offset by the standard size gives the wrong object.

As shown in Fig. 5, if we want to know which object the offset 2.3MB points at, we can get the object index of this object by calculating the quotient of dividing 2.3 by 1 ignoring the remainder. This is because there is no nonstandard object in front of the offset 2.3MB. But if we want to know which object the offset 4.7MB points at, dividing 4.7 by 1 will get 4, but the offset 4.7MB doesn't point at the object whose object index is 4 actually, because there is one nonstandard object in front of the offset 4.7MB. We can get the right object only by consulting the linked

list. So to find out the specific object based on a specific offset, generally, the array needs to consult its linked list from the head of this linked list until the object is found.

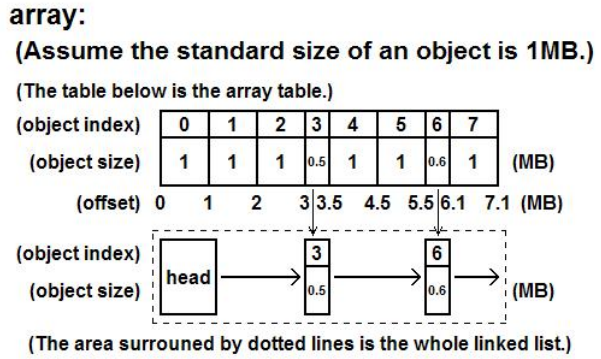

Figure 5. The data structure "array". An array contains 2 parts: 1 array table and 1 linked list. The array table records all object metadata and the linked list just records the metadata of nonstandard objects.

But there comes a problem. With more nonstandard objects, the linked list will get longer to record more metadata of nonstandard objects, and then it becomes less efficient to find the object. Moreover, when an object in the middle of a file is deleted, the corresponding metadata also needs to be deleted and the metadata in the rear of this array also needs to be moved forward, which means this array needs to be modified heavily and that takes a lot of time.

In comparison, as shown in Fig. 6, we can see the bptree records all metadata of objects in the same way whether these objects are standard or nonstandard, which means that Wofs has stable performance. The bptree is a kind of B+tree, so unbalanced sub-trees will not be generated, which can maintain good performance. To look for a particular object based on the specific offset quickly, the bptree clusters some objects into a group, and records the offsets of these objects. Then, the bptree continues to cluster these small groups into a bigger group, and records the offsets of these small groups. The bptree repeats this action until the root group is generated. Usually, a large file has a higher bptree depth.

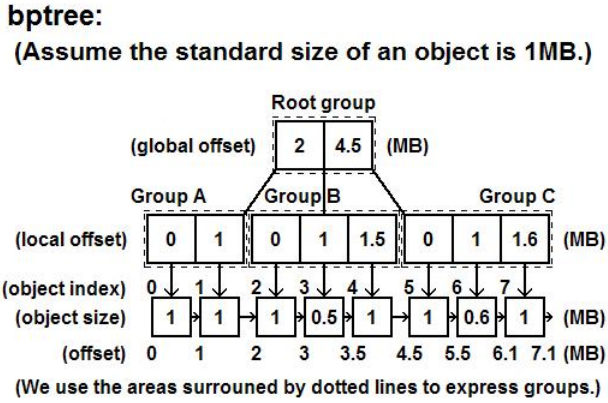

Figure 6. The data structure "bptree". Each group is a small table used to record the local offset, and the squares in the bottom of the figure are used to record the object size for each object. (The object arrangement in this figure is the same as Fig. 5, but the data structures used to record them are different.)

And after the bptree is built, we can start to find out which object a specific offset points at by comparing this offset with the local offsets recorded in groups from top to bottom. For example, as shown in Fig. 6, if we want to know which object the offset 3.2MB points at, we can compare the number 3.2 with the local offsets recorded in the root group. Because 3.2 is less than 4.5 but greater than 2, we need to choose the middle branch and go down to Group B. When we go down to Group B, the number 3.2 has to be subtracted by 2 and we have to use the difference of 1.2 to compare with the local offsets recorded in Group B. Since 1.2 is less than 1.5 but greater than 1, finally we can find the offset 3.2 MB points at the object whose object index is 3.

The subtraction in the process of finding objects is the concept of local offsets in bptree. The design of recording local offsets is very important to bptree. With this, when some object in the middle of a file is deleted, we can just modify the metadata of affected objects and some local offsets in bptree. This is the main advantage of using a bptree to manage the object metadata. And in Section IV, we will compare the performance of the bptree with the array.

E. Client

A client in Wofs takes charge of accessing file data from MDS and OSD servers. In essence, a client in Wofs is a Linux kernel module modified from v9fs [13] which is mentioned in Section II. It receives a user request from user space of Linux and accesses MDS and OSD servers according to this request. And at last, it replies the user with the data one wants or some other information one wants to know. Because the virtual file system of Linux doesn't support insertion and truncation, and Linux does not provide system calls for insertion and truncation, we need to modify the Linux kernel to support insertion and truncation.

F. OSD server

As mentioned previously, in Wofs, OSD servers store all objects of files in the form of normal files on ext3, and the filename of an object is its object index. The objects belonging to the same file are stored in the same directory, and the name of this directory is the inode number of this file. Since all contents of directories and symbolic links are stored in MDS, unlike files, directories and symbolic links store nothing in OSD servers.

IV. PERFORMANCE EVALUATION

A. Hardware specification

Before we discuss the performance of Wofs, we have to know the specification of computers used to evaluate the performance of Wofs. Because we need a lot of clients to evaluate the performance of Wofs in Section IV.E, we use 4 computers with 8 physical CPU cores as our clients to create 32 clients totally. Then, to handle many requests from these 32 clients, we also use this type of computer as our MDS. We use five computers each with one physical CPU core and one RAID-1 disk system as our OSD servers. All computers have Gigabit Ethernet cards to connect with each other via a Gigabit switch.

In all performance evaluations, basically, we use a bptree to manage all metadata of a file and use the object-range locking policy to lock objects. Besides, we always access a file randomly and only access 1MB of this file. We also limit the standard object size to be 1MB, and we only use 1 client to do the performance evaluations. However, there are exceptions in some performance evaluations as listed in Table I. In Section IV.D, we will compare bptree with array, so we also use an array to manage all metadata of a file in Section IV.D. We also need to use different values of file entropy to compare bptree with array, so the sizes of some objects in Section IV.D need to be smaller than 1MB. The definition of file entropy will be introduced in detail in that section. Besides, in Section IV.E, we will compare object-range locking with global locking, so we use the global locking policy to lock objects in Section IV.E. Moreover, in Section IV.E, we need a lot of clients to do performance evaluations, so the number of clients in Section IV.E needs to be more than 1. All the default experimental factors and exceptions are summarized in Table I.

In all tests, we limit the object size to be 1MB. That is, the standard size of an object is 1MB. This is because using a bigger object size can reduce the object number in a file, and then clients can reduce the communication with MDS. But using a bigger object size will increase the amount of the data read out and written back when we do insertion and truncation. So for compromise, we choose 1MB to be the standard size of an object.

TABLE I. THE EXPERIMENTAL FACTORS IN ALL PERFORMANCE EVALUATIONS.

	Default	Exceptions
Data structure for file metadata	Bptree	Array
Locking policy	Object-range locking	Global locking
Access model	Random access	None
The amount of a file accessed once	1MB	None
Object size	1MB	< 1MB
The number of clients	1	>1(up to 32)

B. Basic performance evaluation

In this performance evaluation, we test 5 kinds of operations. These 5 kinds of operations are shown in Fig. 7. At first, we read 1 complete object from a file and write 1 complete object into a file as shown in Fig. 7(a)(b). And then, we insert 1 complete object into the middle of some object in one file just as shown in Fig. 7(c). The so-called "Trunc 1" means we truncate 1 complete object in one file, and the so-called "Trunc 2" means we truncate 1MB across 2 objects in one file and leave two objects of 0.5MB in size just as shown in Fig. 7(d)(e). No matter what the operation is, all accessed objects are randomly picked.

Fig. 8 and Fig. 9 show the results of the basic performance evaluation. In Fig. 8, we can see these 4 operations have stable performance in Wofs no matter how big the file is. It also shows that the time needed for 1 insertion is more than the time needed for 1 read or 1 write. This is because inserting 1 object into the middle of an object will result in data migration inside the OSD server as shown in Fig. 2. The data migration means reading out data and writing it back. In addition, we can see the time needed for these 2 types of truncation is much less than the time needed for 1 read or 1 write. This is because a client in Wofs does not need to send a lot of data to OSD servers to do truncation. It just needs to send the parameters about the objects it wants to truncate to OSD servers, and the OSD servers will truncate these objects for it. Moreover, we can see the time needed for "Trunc 1" is much less than the time needed for "Trunc 2". This is because OSD servers just need to delete the object truncated completely in "Trunc 1", but OSD servers have to truncate objects partially in "Trunc 2", which needs data migration inside OSD servers and takes much more time than "Trunc 1".

Fig. 9 shows the throughput of these 4 operations in Wofs under the Gigabit Ethernet network. Because we use a Gigabit switch to connect all components in Wofs, theoretically, the speed a client sends data to OSD servers can reach 125MB/s. Although clients need to do 3 communications with MDS and OSD servers in Wofs, and MDS and OSD servers also need some time to handle the requests from clients and access objects, in Fig. 9, we still can see that the throughput of read and write in Wofs can reach 82MB/s. Besides, in Fig. 9, although the data migration inside OSD servers takes some time, we still can see the throughput of insertion in Wofs can reach 60MB/s. Furthermore, in Fig. 9, we can see the throughputs of "Trunc 1" and "Trunc 2" are much more than the throughputs of read, write and insertion. We see the throughput of "Trunc 1" can reach about 700MB/s, and the throughput of "Trunc 2" can reach about 290MB/s. Because clients just need to send the parameters for truncation to OSD servers to do truncation, the throughput of truncation can be much larger than the bandwidth of Gigabit Ethernet.

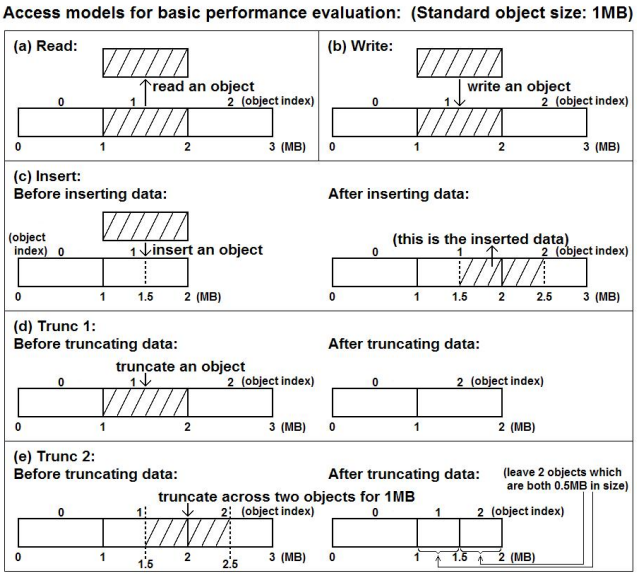

Figure 7. The access models for basic performance evaluation.

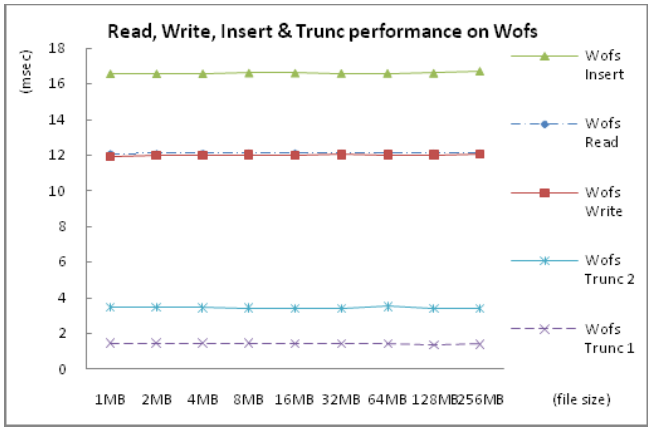

Figure 8. The time needed for doing 1 read, 1 write, 1 insertion and 1 truncation with different file sizes. (The curve of the performance of read almost overlays that of write.)

C. *Insertion and truncation*

In Section IV.B, we can see Wofs has stable and good performance in data insertion and data truncation with variable file sizes. Now we compare Wofs with ext3 to evaluate the advantage of supporting data insertion and data truncation. We choose to compare Wofs with ext3 because ext3 is the most popular file system in Linux and OSD servers use it as their base file system to store object data. Fig. 10 shows the results of this comparison. The so-called "Insert", "Trunc 1" and "Trunc 2" in Fig. 10 are just the access models shown in Fig. 7. Besides, because Wofs is a distributed network file system and ext3 is a local file system, to compare them fairly, we eliminate the time needed for delivering data or parameters in the network to test the time required for doing insertion and truncation in Wofs.

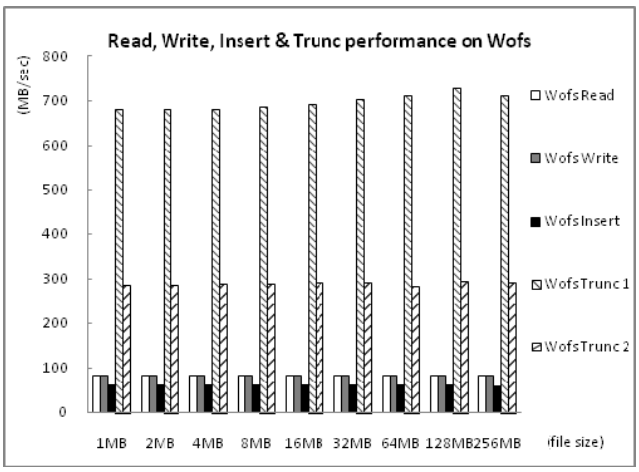

Figure 9. The throughput of read, write, insertion and truncation in Wofs with different file sizes.

Figure 10. The time needed for doing 1 insertion and 1 truncation on Wofs with different file sizes ignoring the time needed for network communication, and the time needed for doing 1 insertion and 1 truncation on ext3 with different file sizes.

In Fig. 10, we can see Wofs has stable performance in data insertion and data truncation as usual, no matter how big the file we access is. In comparison, in Fig. 10, we can see the bigger file we insert and truncate, the more time needed for doing insertion and truncation in ext3. It shows the advantage of supporting data insertion and data truncation. In Fig. 10, we can see the time needed for doing insertion and truncation in Wofs is a little more than the time needed in ext3 when the file size is 1MB. This is because the object size in Wofs is equal to 1MB, and the file which is 1MB in size only has 1 object. So inserting or truncating a file in Wofs is totally the same as inserting or truncating a file in ext3 except that Wofs needs to do communication inside itself to do 1 file access but ext3 doesn't need that. Except that special case, Wofs has better performance than ext3 in data insertion and data truncation ignoring the time needed for network communication.

D. Bptree and array

Since the performance of the data structures used to manage the file metadata is very important to Wofs, we also need to test and compare the performance of these data structures. We use 2 different data structures to manage the file metadata for comparison: bptree and array. The bptree is the data structure shown in Fig. 6, and the array is the data structure shown in Fig. 5. We have already described the disadvantage of the array and the advantage of the bptree in Section III.D.

Now, to discuss how much the number of nonstandard objects in a file affects the performance of bptree and array, we need to define a value called "file entropy". File entropy means the ratio of the number of nonstandard objects in a file to the total number of all objects in a file. From Section III.D, we know the array has worse performance with bigger file entropy. But the value of file entropy does not affect the performance of bptree at all.

To evaluate the actual effect of the file size and the file entropy on the performance of bptree and array, we do this test by reading one object from a file and measure the time MDS needs to handle a read request. Fig. 11 and Fig. 12 show the read performance of bptree and array based on different file sizes and different values of file entropy. In Fig. 11, we can see the performance of the bptree is almost not affected by different values of file entropy, but the performance of the bptree is slightly affected by different file sizes. This is because a large file has a higher bptree, and MDS needs a little more time to traverse the entire bptree to access the object metadata. So MDS needs a little more time to handle the request for a large file. In comparison, the performance of the array is not so stable and much affected by different file sizes and different values of file entropy. By comparing Fig. 11 with Fig. 12, we can see MDS needs much more time to handle a read request when the file size becomes larger and the value of file entropy becomes bigger, which is going to dominate the total time needed for doing 1 complete operation. Besides, we can see the bptree has much more stable performance than the array, and the bptree always maintains a much better performance

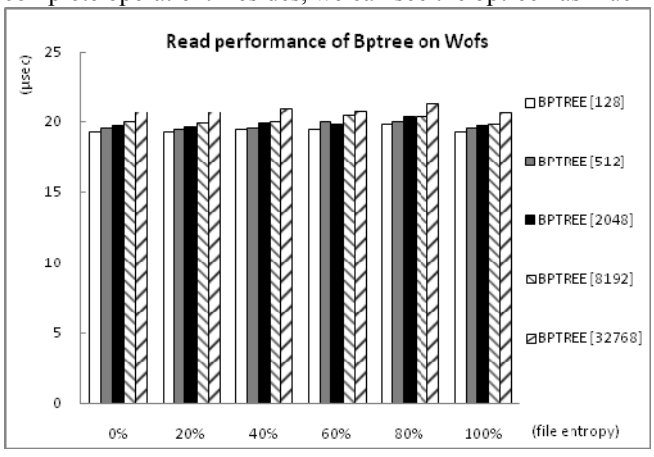

Figure 11. The time MDS needs to handle one read request with different file sizes and different values of file entropy when using the bptree to manage object metadata. (The number "X" in BPTREE[X] in the right side of the figure means the object number of a file.)

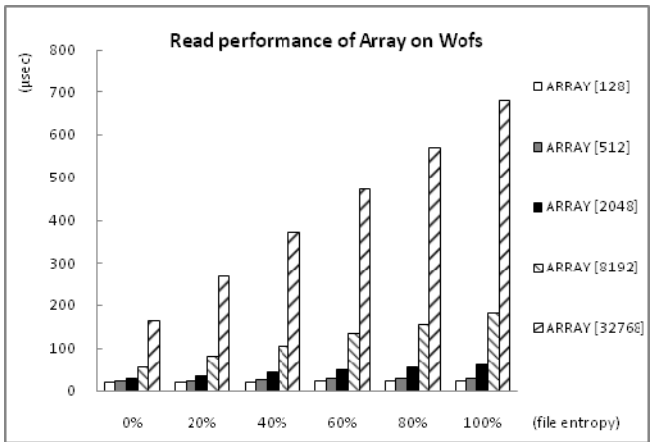

Figure 12. The time MDS needs to handle one read request with different file sizes and different values of file entropy when using the array to manage object metadata. (The number "X" in ARRAY[X] in the right side of the figure means the object number of a file.)

E. .Object-range locking and global locking

Traditionally, different users can't modify the same file simultaneously even when the ranges of the file they modify are different. In Wofs, MDS locks the metadata of accessed objects when a client is accessing the data of those objects, and MDS uses the object-range locking policy to lock the object metadata. With the object-range locking policy, different clients can modify the data of different objects at the same time even when these different objects belong to the same single file. So we can infer that the performance of object-range locking should be better than the performance of global locking. To verify this inference, we do a test to evaluate the performance of object-range locking and global locking.

Fig. 13 shows the relationship between the number of clients and the number of times of write error per successful write in Wofs. A write error means a client tries to write but it fails. In object-range locking, a client can fail on a write and get a write error only if another client is writing the same object with the metadata of that object locked. In global locking, a client will fail to write and get a write error if another client is writing to any place of the file since the second client must have the file locked already.

The object number of the file in Fig. 13 is 1024, and each client tries to write only one object randomly into the same file as one write. So it is difficult for 2 clients to write the same object at the same time theoretically. In Fig. 13, we can see clients easily fail to write the same file simultaneously if MDS uses the global locking policy. And inversely, clients can easily succeed to write the same file simultaneously if MDS uses the object-range policy. Here we verify the performance of object-range locking is really better than that of global locking. With the object-range

locking policy, many clients may easily modify the same file simultaneously without waiting for others.

Figure 13. The number of times of the write error per write with different locking policies and different client numbers.

V. CONCLUSION

Wofs provides a more efficient method for quick file modification by supporting fast arbitrary data insertion and truncation. With Wofs, we can modify a file more efficiently and neatly. We don't need to modify a file by rewriting a large part of the whole file anymore. Besides, it is much better to use the bptree to record and manage the metadata of objects. It provides stable and good performance, and also helps MDS access and modify object metadata more efficiently. Moreover, the object-range locking policy can reduce the frequency of access error significantly and improve the performance of Wofs.

ACKNOWLEDGMENT

The authors would like to thank the support from National Science Council under grant 96-2221-E-007-131-MY3.

REFERENCES

[1] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes", *Acta Informatica*, vol. 1, pp. 173–189, 1972.

[2] M. J. Carey, D. J. DeWitt, J. E. Richardson, E. J. Shekita, "Object and file management in the EXODUS extensible database system", in *Proceedings of the 12th International Conference on Very Large Data Bases*, pp. 91–100, Aug. 1986.

[3] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters", in *Proceedings of the 4th Annual Linux Showcase and Conference*, pp. 317–327, 2000.

[4] D. Comer, "The Ubiquitous B-Tree", *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, Jun. 1979.

[5] S. Ghemawat, H. Gobioff, and ST. Leung, "The Google File System", *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, Dec. 2003.

[6] E. V. Hensbergen and R. Minnich, "Grave robbers from outer space: Using 9p2000 under linux", in *Proceedings of Freenix Annual Conference*, pp. 83–94, 2005.

[7] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-Based Storage", *IEEE Communications Magazine*, vol. 41, no. 8, pp. 84–90, Aug. 2003.

[8] Myricom, *"Myrinet"*, http://www.myri.com/.

[9] O. Rodeh, and A. Teperman, "zFS - A Scalable Distributed File System Using Object Disks", in *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 207–218, Apr. 2003.

[10] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters", in *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pp. 231–244, Jan. 2002.

[11] *"spfs"*, http://sourceforge.net/projects/npfs.

[12] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System", in *Proceedings of the USENIX 1996 Technical Conference*, pp. 1–14, 1996.

[13] *"v9fs"*, http://swik.net/v9fs.

[14] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System", in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 307–320, 2006.