A New Data Replication Scheme for PVFS2

Nianyuan Bao, Jie Tang[1(✉)], Xiaoyu Zhang, and Gangshan Wu

State Key Laboratory for Novel Software Technology
Department of Computer Science And Technology
Nanjing University, Nanjing, China, 210046

Abstract. PVFS is one of the most popular distributed file systems with parallelism, which is still widely used today. Now PVFS is in its version 2, called PVFS2. PVFS2 has a leading performance on I/O operations, but the reliability and stability are not as good. One of the reasons is the lack of data replication. This paper presents a new data replication scheme in PVFS2. In our approach, the backup operation is done on the servers, therefore the user experience is not affected while creating copies of files. In addition, we optimized the read operation of PVFS2. With copies, we can choose the servers to read from, so we can maintain parallelism of read operation under complex conditions such as a server is down or the load of some servers are obviously higher than others. Experimental results verify the effectiveness and efficiency of our method.

Keywords: PVFS2; Distributed File System; Parallel File System; Data Replication; Read Optimization

1 Introduction

With the rapid growth of data size in scientific computing, High Performance Computing (HPC) has received wide concern. While the computing performance is the most influential factor in a HPC cluster, the storage module is also important[1]. Typically, the storage module is a distributed file system, such as Parallel Virtual File System (PVFS).

PVFS is a widely used distributed file system that works well in Linux, now in its 2nd version[3]. The original PVFS was built by Ross R B and Thakur R in 2000, which was designed to have a high parallelism[2]. In fact, the performance of parallel I/O operations of PVFS is in the leading level[4, 5]. Now PVFS is widely used in computing clusters, distributed databases and Storage Area Networks (SANs).

However, PVFS has no data replication[6]. As we know, PVFS slices files into stripes and store them evenly on all data servers. This manner of file storage ensures the high parallelism of PVFS, but it raises the risk of data corruption without data replication. Data has been stored on all servers without a copy, so any server crash will lead to collapse of the whole system. Thus the robustness has been in a great influence. So the data replication of PVFS is essential.

To improve the robustness of PVFS, we managed to achieve a method of data replication. We store the redundant copies in different servers, so that when any server is down, there are always at least one copy stored on another server. In order to maintain the efficiency of PVFS, backup operations are executed by data servers after the users write operation. So the users would not notice the time spend on data replication.

Furthermore, we optimized the read performance of PVFS with data replication. When reading data, PVFS will first find the stripes of the data, then read them. The data layout scheme of PVFS ensures that these stripes would be stored evenly on each data server. So when data servers are in the very balanced state, PVFS can have a nice read performance. But when any server crash happened, or when the server load is badly imbalanced, the parallelism of PVFS would be partly broken, so the performance of the original read operation can no longer meet the expectation. In this condition, redundant copies can be used to rebalance the read time on every data server, so we optimized the read performance in this way.

2 Relative Works

Despite the high performance on I/O operations, the original PVFS has some vital disadvantages on its stability. It has Single Point of Failure (SPoF) along with the lack of data replication[6].

In order to make improvements to the stability and robustness of PVFS, the developers of PVFS have tried to make lots of optimization since 2004[10], and finally released Parallel Virtual File System, version 2 (PVFS2) in 2006[3]. PVFS2 inherited the high parallelism, and improved the stability[11–13]. In this approach, the server architecture of PVFS has been changed. All servers in PVFS2 play the same role. They use the same resource to do the same job, and are under unified management. Thus PVFS2 solved the SPoF problem of PVFS. But still, PVFS2 has a lack of data replication[8, 14].

A cost-effective, fault-tolerant parallel virtual file system (CEFT-PVFS, CEFT in short)[6] has been proposed by Zhu Y and Jiang H in 2006. They borrowed some ideas from Google File System (GFS), made some complement and extension for the function of PVFS, and enhanced its fault-tolerant level to RAID10[6, 15]. They used mirror disks to backup data so that they managed to increase the fault tolerance of PVFS at the cost of decreasing the write performance. However, CEFT is based on the original PVFS, which means it hasn't solve the SPoF problem. Further more, as we mentioned, PVFS2 has been released just in 2006 and has made lots of changes in PVFS to improve its performance and reliability, so CEFT has been out of time.

A fault tolerant PVFS2 based on data replication was made by Nieto E, Camacho H E and Anguita M[8] in 2010. In their approach, they made some changes to the creation state machine so the client will create the copy after the original data. In this method, the client has to write twice or more before it gives the return to the user, so the performance of create and write is severely

affected. In their experiment, a one-copy write operation in their approach costs almost twice as much time as in PVFS2[8].

Our work is based on PVFS2. We made full use of the storage structure in PVFS2 and made some small changes for data replication. In our approach, the replicate operation is not done by clients, but servers. When creating or writing data, the client just do the same job as if there's no data replication, then gives the return user immediately. After that, the server will create the copies when it is idle. In this way, we can provide data replication with little impairment in write performance. In addition, we used our data replication to optimize the reading performance on several conditions. We managed to increase the reading performance when servers have imbalance load or some of the servers are lost.

3 Overview on PVFS2

PVFS was released in 2000. It is coded in C and works in Linux. PVFS is known for its high parallelism in I/O operation which ensures its I/O performance[2]. The original PVFS had some vital problems such as SPoF or the lack of data replication, so its development team did lots of changes in the coding and architecture, and finally released PVFS2 in 2006[3, 10].

In PVFS2, all servers do the same job. There are no master server, and all servers in PVFS2 can play the role of either data server all metadata server or both. Thus PVFS2 solved the SPoF problem PVFS had. But still, PVFS2 has no data replication, so our job is to achieve data replication in PVFS2.

In this section we will introduce the features in PVFS2, so that we can explain our approach in the following sections.

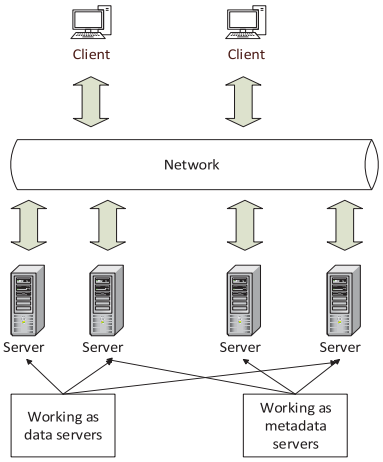

Fig. 1: The structure of PVFS2

3.1 Server Architecture

PVFS2 uses the typical Client-Server Model (C/S Model). There are three logical parts in a PVFS2 cluster, known as the clients, the metadata servers, and the data servers, as shown in figure 1.

A client is working on a user's terminal. It listens to the user's instructions and executes them. Also, a client is responsible for the access to servers during the execution of instructions.

A metadata server stores the metadata of the files stored in PVFS2. The metadata are needed in almost all the operations in PVFS2, so once a client receives a instruction, it will always have to access the metadata server to get some metadata.

A data server is in charge of data storage. When executing I/O instructions, a client will first get the metadata of the target, then access the data server to transmit data.

The metadata servers and the data servers are set up on the same devices in general. We collectively call them servers. Furthermore, in a server, the metadata server and the data server use the same process. A client uses the same interface to access the metadata and data. So we say, all servers in PVFS2 do the same job.

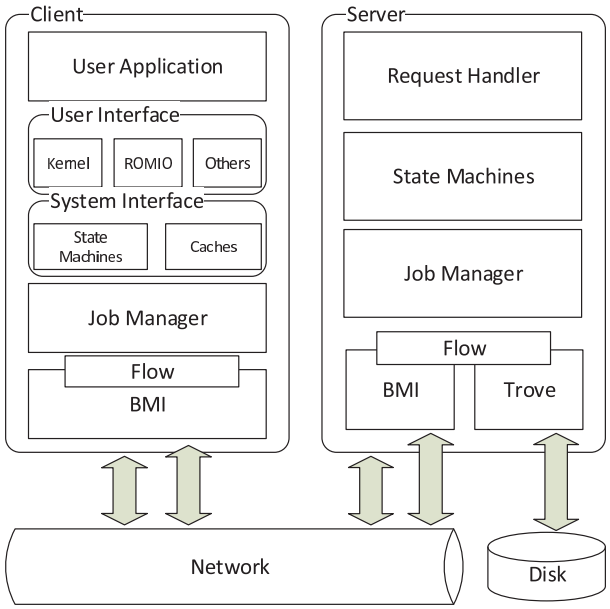

Fig. 2: The architecture inside PVFS2 clients and servers

Figure 2 shows the architecture inside a client or server.In PVFS2, the execution of instructions are in the unit of jobs, which is managed by the Job module.

The Job module exists both on the clients and the servers. They work in coordination to accomplish the whole instruction. When we need to transmit data between devices, we use the Buffered Message Interface (BMI) module. The BMI module also exists both on the clients and the servers. Not only the transmission between clients and servers, but also the transmission between servers and servers will be achieved in it. The BMI module can use different communication protocols such as TCP/IP or Infiniband. In the servers, a Trove module is used for data and metadata storage. Data and metadata are stored in the same form, and a handle is used to access everything stored in Trove.

3.2 Storage Structure

As we all know, PVFS split files into stripes, which ensures the parallelism of I/O operations so that the system would have a nice I/O performance. As a matter of fact, the stripes is only a logical unit. When a file is split into stripes and stored in servers, all the stripes of the file in a single server will be stored in a single file, called datafile, as shown in figure 3.

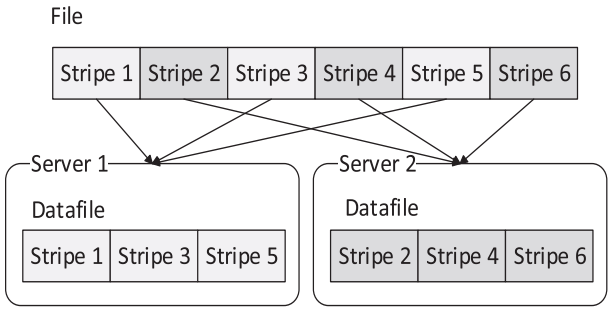

Fig. 3: The storage structure of PVFS2

In PVFS2, files are stored in the form of datafiles. When we need to access a specific stripe of a file, we should get the location of the datafile the stripe stored in as well as the offset of the stripe. The information is stored in metadata. Both datafiles and metadata are stored in Trove. Everything stored in Trove must have a handle, so that we can access it. We can treat a handle as a unique ID. So when accessing datafiles or metadata, we need to get the handle of them first.

The handles of datafiles are stored in metadata, while the handle of metadata are obtained from a hash algorithm. Within the metadata, there stored not only file information and handles of datafiles, but also the position of the stripes, including which datafile a stripe is in and the offset of a stripe inside the datafile. When accessing data in a file, the client first obtains the handle of its metadata with the hash algorithm, then get the metadata from the servers. After that, the client will get the handles of datafiles and positions of the target stripes from

the metadata, then finally get the target stripes from the servers. The process is shown in figure 4.

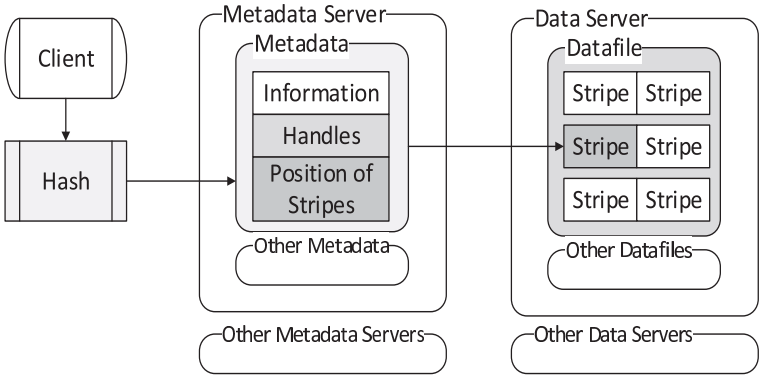

Fig. 4: Process of the I/O operation in PVFS2

4 Data Replication

Generally speaking, data replication is the technology to replicate one or more copies on other devices besides the original data[7]. Specific to a distributed file system, it brings two sort of advancements.

Firstly, the most important effect of data replication is the enhancement of the reliability of a distributed file system[6, 8, 9]. In a distributed file system, files are stored in the disks on the servers. If there are no data replication, each file will have no copies. When a server is down or a disk on the server crashes, the data on the disk will be unreachable, thus the data integrity will be damaged.

Also, data replication can be used to accelerate I/O operations[7, 15–17]. When we have data replication, the same data will be stored on different devices. So we can choose to access a faster and closer device to get the data, so that the I/O cost can be reduced. Furthermore, when more than one client is reading the same data, they can read different copies at the same time without waiting for each other. So the parallelism of the system can be further improved.

There are two major methods to achieve data replication. One of them is using mirror servers[6, 18, 19], the other is storing the copies in the existing servers[8, 20, 21].

Mirror servers are servers that only store copies. Generally, there is a one-to-one correspondence between the mirror servers and original servers. A mirror server simply copies the same data in the corresponding server. When data in a server change, the corresponding mirror server should be synchronized. The method using mirror servers can be easily achieved, and the process is simple. But the cost of this method is too much. Even if we want to store only one copy, the number of servers in the cluster should be doubled.

The other method is to store the copies in the existing servers. In this method, either the client or the server should be in charge of the backup operation, and the information of copies should be stored in somewhere, usually in metadata. This method is much more flexible. We can set different redundant strategies for each file, and don't need to add new servers until disks are full or nearly full. But in this method, the system should do some extra work such as deciding the position of the copies or maintaining the replication information. The operations will be more complicated.

In our approach, we use the second method. Different with others, in our approach the backup operation is not executed on clients, but servers. When copying backups on clients, the backup operation would spend more time on network transmission or disk reading before the operation on clients is complete. In our approach, we move the time cost of the backup operation onto servers. Clients only take care of the writing operation, and the backup operation will be done on servers while clients doing other operations. When the users create a new file or write something in a existing file, our system will first complete the operations on the original file. Once these operations complete, the users will be given the result so they can continue to do other works. After that, the servers will execute the backup operation later. Thus we achieved data replication with little reduction on user experience.

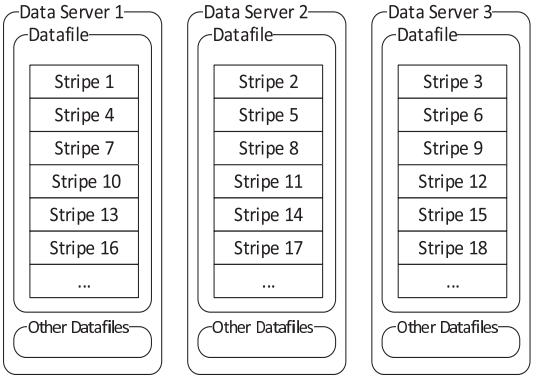

Fig. 5: The PVFS2 Round-Robin Layout

4.1 Backup Distribution

In PVFS2, a Round-Robin Layout is used to decide the distribution of data, which means all stripes are stored in the servers according to the sequence in turn, as figure 5 shows.For example, if a file is stored in 3 servers and the 1st stripe is stored in Server 1, so the 2nd stripe will be stored in Server 2, and then the 3rd in Server 3, the 4th in Server 1, the 5th in Server 2, etc. This feature shows us two things. The stripes are stored evenly in the datafiles, and

the datafiles of a file are almost the same in size. Thus, when creating copies, we can backup data in the unit of datafiles. Here we have three principles:

1. All datafiles and their copies should be stored evenly on all servers. As we all know, PVFS2 doesn't have dynamic load balance, so when we create or write data, we should make the best we can to balance the load of servers.
2. A datafile or its copies should not be stored in the same server. For our major target, the data replication is used to improve the reliability of the system. Specifically, we hope the data can be maintained after servers are down or disks crash. So the datafile or its copies must be on different servers, otherwise a single crash of disk would make two or more copies unreachable, which we don't want to happen.
3. All datafiles of a single backup should not be stored in the same server. This principle is for our backup operation. When creating copies, by default, we primarily copy all datafiles of the first backup, and then the second backup, the third backup, etc. If there are two or more datafiles of a single backup are stored in the same server, the parallelism of the backup operation will be reduced. So we make this principle to avoid this situation.

When a new file is created, we can't predict how large it will grow, so we just consider the number of datafiles stored in a server. After our backup operation, the datafiles and their copies should be evenly allocated to the servers in number. So in our approach, we use another Round-Robin algorithm for the backup distribution.

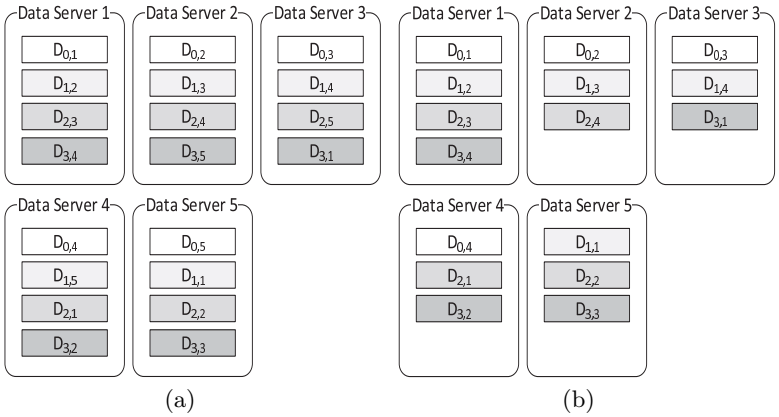

Fig. 6: Examples of backup distribution

Firstly we should get the number of datafiles D, the number of backups B (including the original data), and the number of servers N. On one hand, when D=N, which means the number of datafiles is equal to the number of servers. So the datafiles of each single backup will be stored in all of the servers. On

the other hand, when B=N, which means the number of backups is equal to the number of servers. So every datafile and its copies will be stored in all of the servers. In these two conditions, we can simply store the copies on a next server. So we always backup the datafiles in the Xth server to the ((X+1)mod N)th, ((X+2)mod N)th, ..., ((X+B)mod N)th server. For example, a datafile is stored in the third of five servers, so the first copy will be stored in the forth server, the second copy will be stored in the fifth server, the third copy will be stored in the first server, etc. Another example of the storage of a file with five datafiles and three copies in a cluster with five servers is shown in figure 6a.In the figure, $D_{i,j}$ means the jth datafile of the ith copy, while $D_{0,j}$ means the jth datafile of the original file.

When B and D are both less than N, we first get S=(B*D)mod N, which means there are S servers that need to store more datafiles than others.We choose S servers that have the lightest load, so now we know the number of datafiles or their copies to be stored in each server. Then we use a Greedy algorithm to allocate the datafiles and their copies into the servers, following the three principles. A example of the storage of a file with four datafiles and three copies in a cluster with five servers is shown in figure 6b.

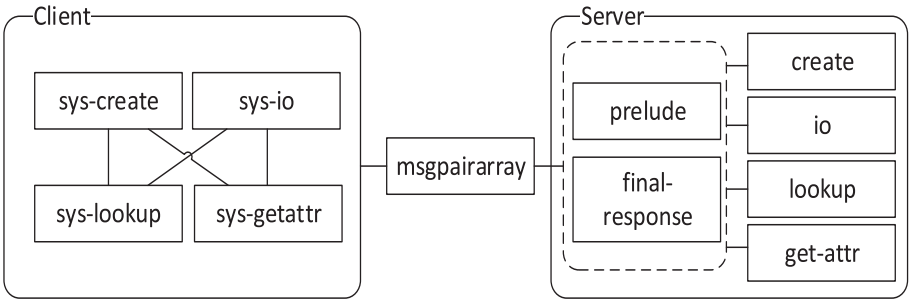

Fig. 7: Some major state machines about I/O operation

4.2 State Machines

State machines are an important part of PVFS2. All the instructions and operations in PVFS2 are executed through state machines. Figure 7 shows some of the major state machines about I/O operation in PVFS2 and some of their relationship.

Due to the modification in the metadata we made, almost all the state machines are updated. In this chapter we only focus on the major changes we made in state machines. We mainly made some changes in the create state machine on the server, the sys-io state machine on the client and the io state machine on the server. Furthermore, we added a new state machine on the server called backup.

The create state machine on the server has the following main steps:

1. Create metadata for the new file.
2. Allocate servers for the datafiles.
3. Create local datafile.
4. Create handles.
5. Send the request to other servers for creating datafiles on them.
6. Save the handles of datafiles into the metadata.
7. Finish the state machine.

In our approach, we allocate servers for the copies in the second step in the create state machine. In the following steps, we create datafiles and handles for the copy along with the same work done for the original file. The creation doesn't take much time, so it can be done on the client.

The sys-io state machine on the client has the following main steps:

1. Obtain the information of the file.
2. Find the target stripes in the datafiles.
3. Send the request to the servers the datafiles are on for data transmission.
4. Transmit data.
5. Finish the state machine.

In our approach, we differ read and write operations in the second step. For a write operation, we will transmit backup information along with the request in the third step.

The io state machine on the server has the following main steps:

1. Receive requests from the clients and do the prelude operation.
2. Send an acknowledgement character to the client.
3. Do the transmission.
4. Send the result to the client.
5. Finish the state machine.

In our approach, if the request contains backup information, the io state machine will jump to the backup state machine when finished.

The backup state machine on the server is a new state machine added in our approach, which takes charge of the backup operation. It has the following main steps:

1. Analyze the backup information.
2. Obtain the metadata.
3. Change the backup status in the metadata.
4. Wait until the server is in a rest.
5. Send a request to the target servers for backup.
6. Transmit copies.
7. Refresh the backup status in the metadata.
8. Finish the state machine.

5 Reading Optimization

With data replication, most data will have one or more copies, so we can optimize the reading performance of PVFS2 now.

As shown in the previous chapters, data replication can be used to accelerate I/O operations[7, 15–17]. Specific to PVFS2, the parallelism will be improved with data replication. For instance, when a file with three datafiles is stored in a PVFS2 cluster of five servers, we can read data from no more than three servers at the same time. After data replication, the datafiles and their copies will be stored in all five servers, so we can read data from up to five servers at the same time, which means the reading speed can be increased by up to 66%.

Moreover, with data replication, the target to read can be chosen wisely to accelerate I/O operations. Without data replication, every single stripe of a file is only saved on one server, so we have no other choice when we want to access this stripe. But the situation has been changed because of data replication. Now datafiles have their copies stored in other servers, so there are more than one choice to access a stripe. We can choose to access a stripe on a server with a lightest load. When accessing more than one stripes at once, we can adjust the data scale of the targets on each server, so that we can access less data from a server with a lighter load and more data from a server with a heavier load.

In the extreme case, when some servers are down, we have to read data from other servers. In our approach we can still read data evenly from the rest of the servers, so the parallelism is maintained.

5.1 Hotspot Detection

The load of servers will not be always the same. In actual operations, there will always be some servers with a higher load, while some others with a lower one. A server is called a hotspot when the load of it becomes so high that its operating or I/O efficiency is badly influenced.

There are some ways to detect hotspot. Most of them use a percentage to show how high the load of the server is. In our approach, we'd rather use a expected transmission speed (ETS) to show this, so that when reading, we can use the ETS to work out a best allocation of the targets.

We consider of the read speed of the disk first. There are always a standard read speed of a disk, and we call it S. We define X as the current read speed of the disk, N as the network speed, and E as the ETS. When the load of the server is extremely low, we consider E=S. If some accident happens that the network speed becomes so slow, we consider E=N. In general case, X will have some effect on E, but in our practice we notice that the effect is not linear. When the value of X is small, it has little effect on E, while when the value of X is big, it will have a greater effect than we expect. The RAM and CPU occupancy rate can also affect the ETS, but the case only happens when the RAM or CPU occupancy rate grows so high that the server process has no response, in this case we assume E=0. We temporarily use the following formula to calculate the

value of E:

$$E = min(S - aX^b/S^c, N)$$

While the value of a, b and c still need to be adjusted. In our practice, we found a=3, b=2, c=1 is a simple answer that has a fair performance.

5.2 Target Selection

Because of the data replication, we can choose the datafiles or their copies to read, so we can read more data from servers with lower load and read less data from servers with higher load. To achieve this theory, we have to decide which server do we read each stripe from.

First we can obtain the ETS of the servers. Considering the load of the server is a real-time attribute, the client should send an additional request to get the ETS. Once we got the ETS of the servers, we can obtain a allocation of servers to read stripes from so that the transmission time in every server can be the same. Thus we can get the optimistic plan that has the least read time. For example, if we have to read 1000 stripes from two servers A and B. After the first step we get the ETS of A and B to be 200MB/s and 300MB/s, so we can read 400 stripes from A and 600 stripes from B, so that the expected transmission time of A and B can be the same, and the read time can be the shortest. Besides, a server should be selected for each stipe we read. In the selection, we prefer to read the stripes continuously stored in a datafile from one server, so we use a depth-first search. Figure 8 shows a simple example of target selection.

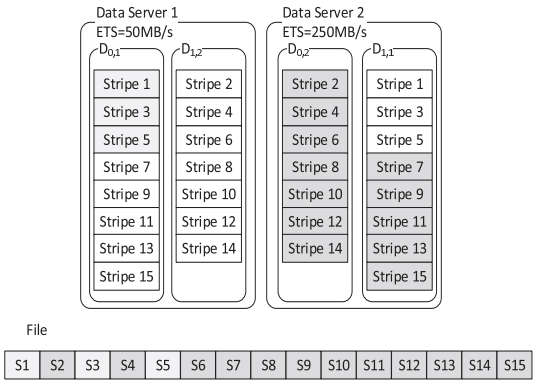

Fig. 8: A simple example of target selection

6 Experimental Results

First we verified the effectiveness of our data replication. We used a number of servers as data servers, and another server as the metadata server to do the

experiment. Thus, the influence of the lost of metadata has been prevented. As table 1 shows, we used different files that have different sizes, different number of datafiles, different number of copies stored in different number of servers.Then we shut one of the servers down, and check if the data stored in the rest of the servers are complete. The table shows under the correct settings, a file with copies stored in other servers can maintain complete when a server is down. As a result, the fault tolerance of PVFS2 has been improved.

Table 1 The efficiency of data replication

File size/MB	1	16	1024	1024	1024	1024
Datafile number	1	3	3	1	1	1
Backup number	1	2	2	0	2	0
Server number	3	3	3	1	3	3
If passed validation	**Yes**	**Yes**	**Yes**	**No**	**Yes**	**No**

Then we did some experiments to test the write performance of our approach. Figure 9a shows the write performance of PVFS2 with and without data replication.In the experiment, we separately wrote different files into a PVFS2 cluster with four servers with and without data replication. We recorded the write time on the client side, which shows how long the users have to wait until they can do their next works. In the figure, we can find the PVFS2 with data replication is a little slower than the original PVFS2. This is because we have to spend more time on creating metadata and allocating servers for copies. The time we do these jobs are almost regular, so when the data size grows large, the time gap between the two systems will be shorten. As shown in the figure, when the size of the file grows larger than 64MB, the two systems have almost the same performance.

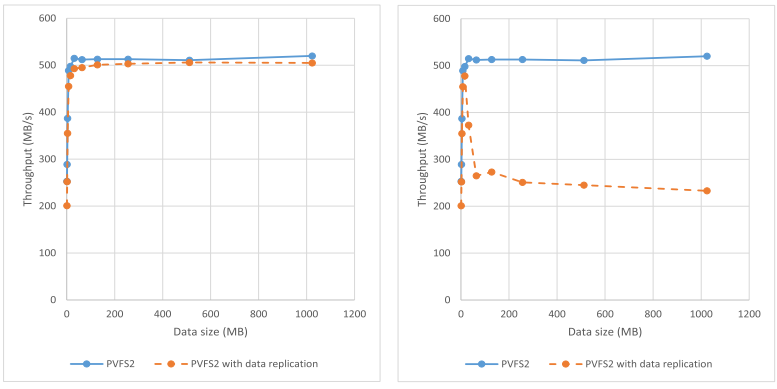

Fig. 9: Write performance of PVFS2 with data replication

Figure 9b shows the write performance a short period after the previous write operation.We can find the write performance of our approach is badly affected in this experiment. In our approach, the backup operation is done by the servers when they are free, which brings us a shortcoming. When the user wants to do some write operation just when the servers are doing backup operation, the write performance can be badly endangered. This is because of the limit of write speed of the disk. When copies are being written in the servers, the disks are so busy that they can't show their best performance in writing new data.

We also did some experiments on the reading optimization. The reading optimization is designed to deal with read operation in complex environment such as when a server is down or when the load of some servers are obvious high. So when the cluster is health and stable with even loads of all servers, our approach will be a little slower than the original one PVFS2 provides. Figure 10 shows the contrast reading files from a cluster with four servers.Though the difference is very small, our approach is slower indeed. So we allow users to use the original read operation PVFS2 provides. Figure 11 shows the read performance when a server crashed.In the experiment, we store the file and three copies in a cluster with four servers, then we shut one server down. As we can see, the read performance of the rest three servers is almost the same as which in a healthy cluster with three servers. So in our approach, we can ensure the parallelism even in the extreme case that a server is down.

At last we tested the read performance of a cluster with three servers when one of the servers has a very high load. As we can see in figure 12, the read performance is obviously better than the one before reading optimization. This is because in our method, the size of data read from the server with the high load is less.

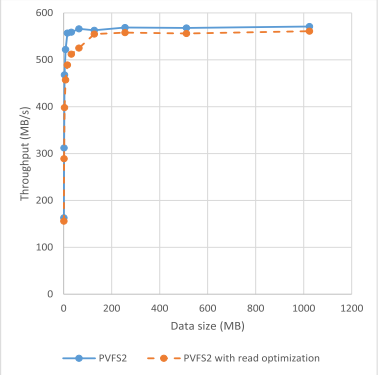

Fig. 10: Read performance

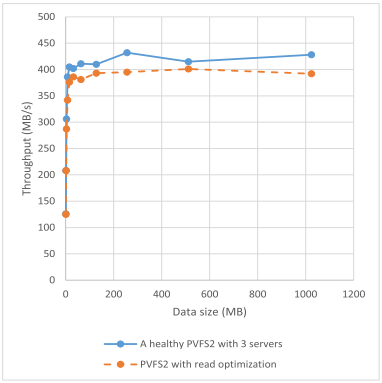

Fig. 11: Read performance when a server crashed

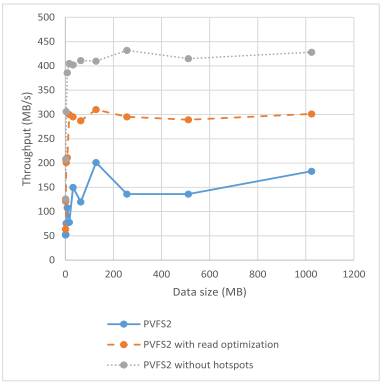

Fig. 12: Read performance when the load of servers is imbalance

7 Conclusion

In our approach, we achieved data replication in PVFS2. We placed the backup operation on the server, so that the user experience will not be affected while the stability of the system is improved. The result of our experiment shows the reliability of our approach, and the speed is nearly the same with the original PVFS2. In additional, we optimized the read performance using copies. As a result, our system managed to maintain parallelism when the cluster is not in a perfect situation.

Still there are some points for us to improve. We are going to find some way to avoid the conflict between backup operation and the following write operation. Also we can try to use the buffer to accelerate the backup operation. Besides,

we can improve PVFS2 in other ways. For example, we can achieve a dynamic load balance for PVFS2.

Acknowledgments

We would like to thank the anonymous reviewers for helping us refine this paper. Their constructive comments and suggestions are very helpful. This paper is partly funded by National Science and Technology Major Project of the Ministry of Science and Technology of China under grant 2011ZX05035-004-004HZ. The corresponding author of this paper is Jie Tang.

References

1. Zhao D, Raicu I.: Distributed file systems for exascale computing[J]. Doctoral Showcase, SC, 2012, 12.
2. Ross R B, Thakur R.: PVFS: A parallel file system for Linux clusters[C]. Proceedings of the 4th annual Linux showcase and conference. 2000: 391-430.
3. Parallel Virtual File System, Version 2, http://www.pvfs.org/
4. Wu J, Wyckoff P, Panda D.: PVFS over InfiniBand: Design and performance evaluation[C]. Parallel Processing, 2003. Proceedings. 2003 International Conference on. IEEE, 2003: 125-132.
5. Wu J, Wyckoff P, Panda D.: Supporting efficient noncontiguous access in PVFS over InfiniBand[C]. Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on. IEEE, 2003: 344-351.
6. Zhu Y, Jiang H.: Ceft: A cost-effective, fault-tolerant parallel virtual file system[J]. Journal of Parallel and Distributed Computing, 2006, 66(2): 291-306.
7. Bell W H, Cameron D G, Millar A P, et al.: Optorsim: A grid simulator for studying dynamic data replication strategies[J]. International Journal of High Performance Computing Applications, 2003, 17(4): 403-416.
8. Nieto E, Camacho H E, Anguita M, et al.: Fault tolerant PVFS2 based on data replication[C]. Parallel Distributed and Grid Computing (PDGC), 2010 1st International Conference on. IEEE, 2010: 107-112.
9. Satyanarayanan M.: A survey of distributed file systems[J]. Annual Review of Computer Science, 1990, 4(1): 73-104.
10. Latham R, Miller N, Ross R, et al.: A next-generation parallel file system for Linux cluster[J]. LinuxWorld Mag., 2004, 2(ANL/MCS/JA-48544).
11. Zhang X, Jiang S, Davis K.: Making resonance a common case: A high-performance implementation of collective I/O on parallel file systems[C]. Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, 2009: 1-12.
12. Kunkel J M, Ludwig T.: Performance evaluation of the PVFS2 architecture[C]. Parallel, Distributed and Network-Based Processing, 2007. PDP'07. 15th EUROMICRO International Conference on. IEEE, 2007: 509-516.
13. Chai L, Ouyang X, Noronha R, et al.: pNFS/PVFS2 over InfiniBand: early experiences[C]. Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07. ACM, 2007: 5-11.

14. Choi Y H, Cho W H, Eom H, et al.: A study of the fault-tolerant PVFS2[C]. Computer Sciences and Convergence Information Technology (ICCIT), 2011 6th International Conference on. IEEE, 2011: 482-485.
15. Zhu Y, Jiang H, Qin X, et al.: Improved read performance in a cost-effective, fault-tolerant parallel virtual file system (ceft-pvfs)[C]. Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on. IEEE, 2003: 730-735.
16. Wolfson O, Jajodia S, Huang Y.: An adaptive data replication algorithm[J]. ACM Transactions on Database Systems (TODS), 1997, 22(2): 255-314.
17. Saadat N, Rahmani A M.: PDDRA: A new pre-fetching based dynamic data replication algorithm in data grids[J]. Future Generation Computer Systems, 2012, 28(4): 666-681.
18. Cachin C, Junker B, Sorniotti A.: On limitations of using cloud storage for data replication[C]. Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on. IEEE, 2012: 1-6.
19. Lustre, http://lustre.org/
20. Ghemawat S, Gobioff H, Leung S T.: The Google file system[C]. ACM SIGOPS operating systems review. ACM, 2003, 37(5): 29-43.
21. Shvachko K, Kuang H, Radia S, et al.: The hadoop distributed file system[C]. Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010: 1-10.