Journal of Theoretical and Applied Information Technology
20th May 2013. Vol. 51 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

IMPROVED ALGORITHM FGF-LRU FOR I/O PERFORMANCE OF LUSTRE

[1] LI LINLIN, [2] SUN LIANGXU

[1] Lecturer, School of Software, University of Science and Technology Liaoning, China
[2] Lecturer, School of Software, University of Science and Technology Liaoning, China

E-mail: [1]linlin20002_0@163.com, [2]sunliangxumail@163.com

ABSTRACT

In Lustre parallel file system the I/O performance of coarse-grained data access is good, while the I/O performance of fine-grained data access is relatively worse. Therefore optimizing the I/O performance of fine-grained access is the key for the improvement of the system's overall I/O performance. This study analyzed Lustre's I/O access mode, fine-grained I/O service procedure and page reclaim algorithm, and proposed a Fine-Grained First LRU (FGF-LRU) algorithm. In this algorithm, fine-grained I/O pages are retained in cache of OST and client as long as possible, to slow down the pages' sinking, and extend the time that the pages are in main memory, and thereby, decrease the number of times of disk access and reduce the disk access cost. Through comparison and analysis of the test data, the algorithm is confirmed effective; I/O performance of fine-grained access is improved without affecting the coarse-grained I/O performance, and the system overall I/O performance is improved.

Keywords: *Parallel file system, Lustre, Fine-grained, I/O*

1. INTRODUCTION

With the growth of the demands for massive data storage and processing from large-scale applications like internet and cloud computing, the performance requirement of parallel computers' storage system increases as well. In practice, most parallel computers' storage system need to store many small files besides large files; and the system needs to read and write these small files often, or perform fine-grained read/write to the large files. The fine-grained read/write to large number of files is not caused by occasional application error, they are from different sources; this indicates that the fine-grained read/write of files is common. Therefore researches on the performance of fine-grained read/write have important practical meaning.

To make the structure of the parallel file system more reasonable and the performance better, the research on the parallel file system need to deal with mainly the following three aspects of problems:

1) Determine user I/O access mode. Different user application has different I/O access mode. In order to provide high-performance services, determine to use what kind of optimization method according to specific user access mode. It is difficult to provide a more complete access mode information and system design guideline for the analysis work of the current I/O access mode.

2) Develop access parallelism. There are two main methods to develop access parallelism: define explicitly by users and excavate by the parallel file system itself. But the above two methods have advantages and disadvantages. The users often cannot understand the distribution of the underlying data; it is difficult to provide parallel information for many access modes. For example, when the access behavior of the user is complex, or the size of the access data is small, or the distribution of the access data is fine-grained, it is difficult to develop access parallelism for the parallel file systems by itself. Therefore, the best solution is to organically combine the above two methods. But how to realize is the current research difficulty.

3) Optimize parallel services. The key of optimizing parallel services is to take full advantage of the performance of system components, eliminate bottlenecks and get maximum bandwidth and minimum delay. For different data distributions, the existing service mechanism often doesn't take into account system performance as a whole and leads to the emergence of new bottleneck component, such as the communication between nodes when the data distribution is fine-grained.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Therefore, based on considering the characteristics of system components, the research on appropriate parallel service mechanism is the key to achieving optimization of parallel services.

Lustre[1]-[2] is an object-based file system shown as Fig.1, composed by Metadata Server (MDS), Metadata Target (MDT),Object Storage Server (OSS) and Client. Client carries out file data I/O interaction with the Object Storage Target (OST) under OSS and namespace operation interaction with MDS.

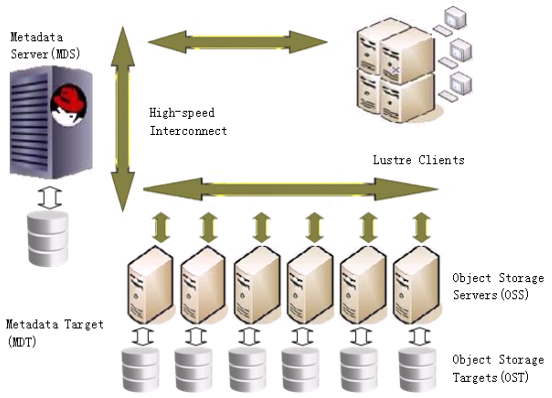

Figure 1. Lustre Components

Lustre is a high-performance open-source file system, usually used in high-performance computing environments. It is based on Linux, designed, developed and maintained by Oracle, with the participation of some individuals of open-source communities and some companies also. Lustre optimized the continuous coarse-grained read/write of files, and it is able to provide great I/O throughout rate for cluster systems.

In practical applications, access to bulk data has high probability to be one-time access and takes more resource, while accesses to small pieces of data has high probability to be repeated but takes less resource. This means in file read/write process, fine-grained accesses are more likely to be repeated than coarse-grained accesses, thus the low performance of fine-grained access become the bottleneck of Lustre's overall I/O performance improvement.

2. RELATED WORK

For the problem about metadata prefetching of server driver and the flat namespace caused by the distribution storage architecture of Lustre, modify the metadata of objects and add the fields which reflect the relationships between objects in the metadata of objects, when access to MDS to do the operations of metadata each time, pre-fetch the metadata of the objects relevant to the accessed object this time, so as to decrease the access number of MDS.

Lustre caches only in the client. When different users access the same file from the same OST, need to store a copy of the file in the file system cache of the client. From a global point of view, this will waste cache resources of the system and need additional I/O bandwidth. For this problem, the collaborative cache technology [3] is to design cache from the global point of view and deal with cache of all clients as a whole; one node not only can use the cache resources of the node, but also use the cache resources of other nodes. The technology can take advantage of the cache resource of cluster system and also save I/O bandwidth of a small file.

When the I/O operation is running in the medias whose storage speed level has a big gap such as RAM and Disk, improve I/O performance often with cache technology. According to the locality principle, read the files which may be used by users in a certain time period from the low speed storage media such as Disk to the high speed storage media such as RAM and cache as a copy in low speed storage media. When the system is idle, write asynchronously the copy in cache to the disk. The majority of the file systems apply the cache technology in different way to improve the I/O performance in a certain extent.

In system running, the number of the clients may change at any time, and the cache resource of each client may be respectively different which cause that management of cache resources is very complex. In order to fully taking advantage of the speed of cache storage media, the management and maintenance of cache resources must be as simple as possible. For this problem, in order to improve the I/O performance of small file, design a cache only for the I/O of small file in OST component, let the I/O data of small file pass through the cache firstly, and then write asynchronously to disk, while the I/O operation of big file does not change[4].

3. PAGE RECLAIM MECHANISM

3.1. Page Reclaim Requirement

Operating system manages the physical memory pages, and also allocates the memory. Applications apply physical pages from the OS through memory allocation functions, and after using, release these physical pages through corresponding memory release functions. However, some users would not release the physical pages initiatively. If such pages

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

are always occupied and not been released, the physical memory will run out eventually. Therefore, for the physical pages which can't be initiatively released, the OS needs to provide appropriate functions to release them. In Linux, a mechanism of page reclaim algorithm is provided for such purpose.

3.2. Page Reclaim Algorithm

Page reclaim in Linux is based on the Least-Recently-Used (LRU) algorithm, while LRU is based on the fact that the pages been frequently used recently are more likely be accessed again in the near future; in contrast, pages that have not been accessed for a long time would not be accessed frequently in short time. Therefore, the pages have been least accessed recently will be swapped out when the physical memory is not enough.

Linux added an "Accessed" bit in the page table entry, when the page is accessed, this bit is set. The bit been set indicates that the page is young and should not be swapped out. Afterward in the system's running, this page's age will be changed by the system. Related operations in Linux mainly run with two LRU linked lists and two page status identifiers.

LRU linked lists include active linked list and inactive linked list. Active linked list stores the pages been accessed recently, and inactive linked list stores the pages have not been accessed in a period of time. This benefits the improving of the efficiency and reduces of the moving time. Pages are moved between the two two-way linked lists, and the system would determine which list to put a page in according to the active level of the page. The least recently used pages will be put to the tail of the inactive linked list one by one, and the system would reclaim the pages from the tail of the inactive linked list.

4. I/O GRAINED ANALYSIS

Using object storage technology, Lustre divides each file into a number of segments of the same size, and orderly stores them to designated OSTs; all the segments on one OST for the same file are treated as an object. For the data buffering and network design, the data dividing strategy of Lustre fits well for large-volume sequential for following two reasons:

1) At one time point (or in a time period), if there are I/O requests for multiple continuous bulk data, and the requests are for different segments of the file, then these requests will be distributed to different OSTs and served by them in parallel. In this I/O model, the bigger the granularity of the access is, the more segments are assigned, then the parallel degree of the system is higher, and the aggregate I/O bandwidth is bigger.

2) During bulk data I/O operations, the OST-side disk's magnetic head moves orderly, after one disk addressing, the next data segment can be accessed directly without re-addressing. This sequential data transmission could reduce the frequency of the magnetic head's moving and the cost of disk addressing.

However most of the actual loadings are not pure bulk data accesses, there's large number of I/O accesses of small piece data also. In Lustre's objective storage structure, data path and control path are separated which is not good for fine-grained I/O for the following 3 reasons:

1) Since metadata and file data are stored in MDS and OST separately, interaction with MDS is required before file access to obtain the location of the file. For fine-grained I/O, comparing with traditional local file system, this step costs additional network transmission and metadata server access. These two additional costs are quite large for small data piece I/O. This is a common problem in all object storage-based file systems.

2) The fine-grained I/O performance is not good, additional bandwidth is required. Client cache is only effective on local machine, and when different clients access the same file at OST side, every client needs to copy the data from OST to local cache first. Even if the data is only a small part of the page (which is common for small piece data I/O), the whole page needs to be read from the OST disk to cache before accessing. This process of reading data from OST to client cache requires additional bandwidth; the more frequent the fine-grained I/O is, the more additional bandwidth is required, and the lower the system's overall I/O performance gets.

3) The performance of small piece data read/write at OST-side file system is not good. When reading and writing small piece data, the latency caused by disk addressing takes a big part of the total time cost of one small piece data I/O. The fine- and coarse-grained mixed I/O model also leads to disk fragment issue thus reduces the disk's overall performance. This is determined by the disk's physical characteristics, and it is a common problem in the storage end of all file systems.

In conclusion, frequent fine-grained I/O impacts the overall I/O performance greatly; the actual runtime I/O bandwidth is less than the maximum bandwidth available.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

5. FINE-GRAINED FIRST LRU (FGF-LRU) ALGORITHM

5.1. I/O Grained Determination

In general, the response time T is the combination of OST response time T_{ost}, client delay time T_{Client} and OST network transmission time T_{net}, i.e. $T=T_{ost}+T_{Client}+T_{net}$. When the accessed data is over the size of one segment which means the data is stored in more than one OST, because of the segment parallel handling mechanism, each OST handles the corresponding segment, so $T= T_{Client}+Max(T_{ost} +T_{net})$. This formula means that the I/O response time of multiple segments is decided by the longest OST response time. If the longest OST response time is not reduced, it cannot make any difference to the access' performance even other OSTs' performances are improved. So once the data access' granularity is over one segment, it will greatly enjoy the benefits of parallel handling and pre-read mechanism, thus have less demand of the performance optimization for fine-grained I/O.

Therefore the impact of fine-grained access on performance is closely related to the segment size, which should be the determination standard of fine- and coarse-grained.

Fine-grained: the number of bytes read/written one time is smaller than the size of one segment;
Coarse-grained: the number of bytes read/written one time is bigger than the size of one segment.

5.2. LRU Algorithm Analysis

The procedure of Linux' original LRU algorithm is: add the newly allocated page descriptor to the head of the inactive linked list; when the page is been accessed, set its status to "referenced" or move it to the active linked list; when free pages are not enough and page declaim is needed, the system scans the active linked list from the tail and put the list item whose status is not "referenced" to the head of the inactive linked list, then scan the inactive linked list from the tail, reclaim the pages in proper status until enough pages are reclaimed.

When a large amount of fine-grained accesses are in the cache, a few coarse-grained accesses would make a lot of them be swapped out. Then the fine-grained requests of high access repeating probability would have to do the network data transmission and disk read/write again, which affects the system's overall I/O performance significantly. And coarse-grained accesses usually read data in bulk continuously, this means that even if the first read did not hit in the cache, the prefetching mechanism of the file system could still compensate the performance lost, however this mechanism doesn't compensate the performance loss of the fine-grained accesses with high repeating probability.

Therefore, in order to improve the overall performance, the fine-grained access pages should be retained in cache longer, while coarse-grained access pages should be swapped out first. However the original Linux LRU algorithm performs LRU ordering only according to the pages' access time, and does not consider the impact of file accesses' granularity on the access repeating possibility. The coarse-grained access pages should be put to the tail of LRU linked lists as far as possible, so that the fine-grained access pages of high access repeating probability won't be swapped out because the coarse-grained access pages of low access repeating probability occupied the cache, and therefore improve the performance of fine-grained I/O accesses.

5.3. FGF-LRU Algorithm Design

The main ideas of the Fine-Grained First LRU (FGF-LRU) algorithm are:

1) When Client and OST response requests, identify the requests' access granularity and handle them respectively;

2) Extend the time of fine-grained accesses in cache to increase the access hit rate, let as much fine-grained accesses run in cache as possible.

FGF-LRU algorithm is based on LRU algorithm. The procedures are:

1) For the pages in LRU linked lists, add an identifier PG_finegrain, for marking different access granularity;

2) Determine the access granularity in the files' execution path. If file pages are read into cache from disk in fine-grained mode, set this page's PG_finegrain identifier, and put the page to the head of the inactive linked list; If not, put the page to the tail of the inactive linked list.

3) When moving pages from active linked list to inactive linked list, move fine-grained pages as normal, and move coarse-grained pages to the tail of the list. This way when the system doesn't have enough free pages, these coarse-grained pages will be swapped out first.

5.4. FGF-LRU Algorithm Realization

The data transmission path of fine-grained access is Client -> OST -> Cache. FGF-LRU algorithm is divided into three modules accordingly. Combined the function of managing the page identifier in linked list, the realization of the algorithm involves four modules: Client-side reading module, OST-

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

side reading module, LRU linked list moving module and page identifier management module.

1) Client-side reading module: identify the access granularity of system requests, and handle the requests accordingly;

2) OST-side reading module: identify the access granularity of Client requests, and handle the request accordingly;

3) LRU linked list moving module: move pages of different granularity from active linked list to inactive linked list based on the idea of fine-grained first;

4) Page identifier management module: manage the checking, setting, and clearing of the page structure flags field and fine-grained identifier.

Among these four modules, the Client-side reading module, OST-side reading module and LRU linked list moving module do not have direct data exchange with each other, but all of them would call the page identifier management module directly. These four functional modules are closely integrated with relative system components for the realization of the FGF-LRU algorithm.

To accelerate the page LRU operations, a page shouldn't be put into the inactive linked list every time it is called. The reason is that the inactive linked list and the active linked list are a shared data structure protected by spin lock; frequent modification to them means intense lock competition. Per-CPU amount is a new feature of Linux2.6 kernel, where each processor has its own copy of the variables. The processors access their own copy without locking; using their own cache greatly improved the access and update efficiency. Therefore a Per-CPU amount should be defined to temporarily store the pages to be submitted to the inactive linked list, and the stored pages will be add to the actual inactive linked list when there is a certain amount of them.

6. TEST AND ANALYSIS

6.1. Test Environment

Six PCs are prepared for the test, three of which as OST and the other three as Client, MDS is on the third OST. The OST (MDS) PCs' technical details are 1.7GHz dual core CPU and 1G memory. The Client PCs' technical details are 2.4GHz dual core CPU and 2G memory. Network is Gigabit Ethernet, with gigabit exchange board. OS for all PCs is Red Hat Enterprise Linux 5, kernel is 2.6.38. Parallel file system is Lustre 1.8.

6.2. Test Case Design

Response time and bandwidth are important factors to measure a parallel file system's I/O performance. Through the comparison of the average response time and bandwidth before and after the optimization, the effect of FGF-LRU algorithm can be reflected. Define the generation probability ratio of fine-/coarse-grained accesses as m/n; and M large enough files are stored in the system. Assume all the files are stored in a small data range and would be accessed frequently.

With different fine-/coarse-grained accesses generation probability ratios, gauge the response time and calculate the average bandwidth.

The random algorithm of a file access is as following:

1) Generate a random number r in the range of 0 to 100 using uniform distribution;

2) When r<m/(m+n)*100, this access is fine-grained access, and it is coarse-grained access if reverse;

3) If the access is fine-grained, choose one of the M files randomly using uniform distribution, and perform fine-grained access to the frequently accessed data range; if reverse, randomly choose a file and perform coarse-grained access.

Use Linux shell scripts to complete the test in batch; use $RANDOM for the random number generation, and dd command to gauge the read/write time as below:

Read: $time dd if='input file' bs='segment size' count='segment amount' of=/dev/null

Write: $time dd if=/dev/zero bs='segment size' count='segment amount' of='output file'

6.3. Test Result and Analysis

To ensure the universality and randomness of the test, the test was repeated many times so that it is more actual application-like and ensure the data's reliability.

Define the occurrence ratio of fine-grained and coarse-grained read/write as m/n, and probability of fine-grained access is m/(m+n). A single test contains following steps:

1) Store 10 files of 1.5G in advance;

2) Conduct 100 times of random read/write test to fine-grained probability 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2 and 0.1 separately and get 9 total response time; calculate the 9 average total bandwidth values (fine-grained bandwidth + coarse-grained bandwidth);

3) Transform the total response time get in step 2 to fine-grained accesses total response time, and calculate the average bandwidth of fine-grained accesses.

Journal of Theoretical and Applied Information Technology
20th May 2013. Vol. 51 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Test data of overall average bandwidth and optimization percentage of the read/write test are shown in Fig.2 and Fig.3.

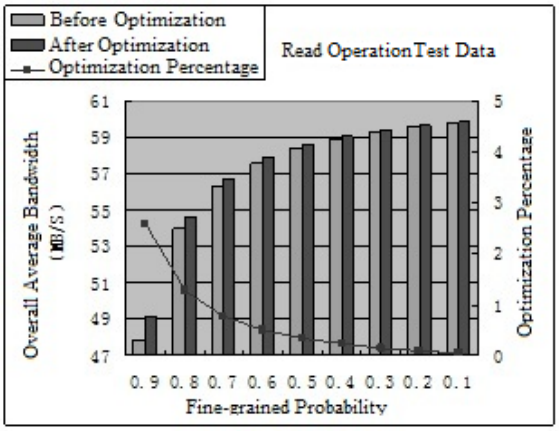

Figure 2. Read Operation Test Data

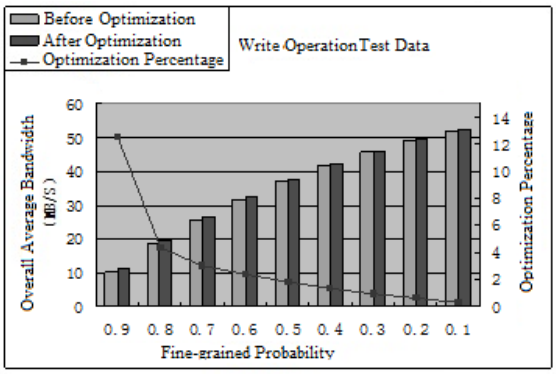

Figure 3. Write Operation Test Data

Test data of average fine-grained access response time and optimization percentage of the read/write test are shown in Fig.4 and Fig.5.

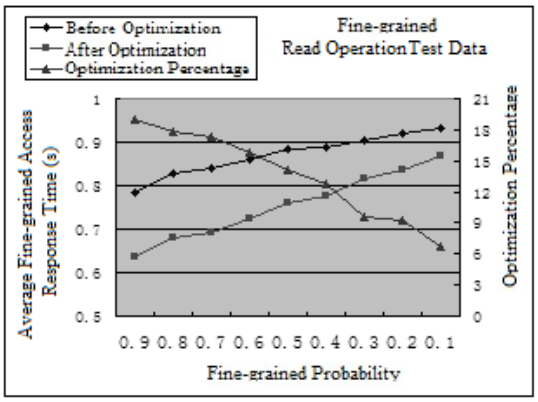

Figure 4. Fine-Grained Read Operation Test Data

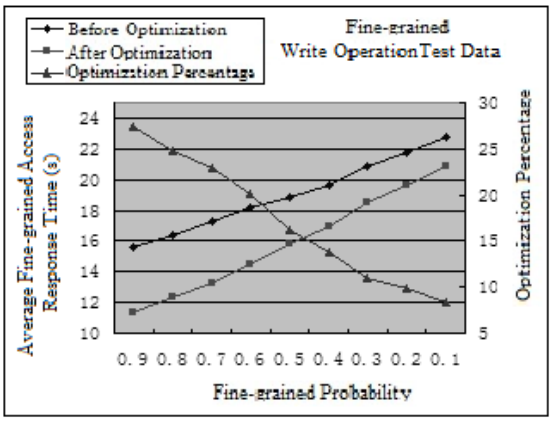

Figure 5. Fine-Grained Write Operation Test Data

Test for the overall average bandwidth of read/write operation shows that, for a certain fine-grained probability, the optimized overall average bandwidth is bigger than the average bandwidth before; along with the decrease of the fine-grained probability, the overall average bandwidth of both before and after optimization gradually increase, while the pre-post bandwidth difference and optimization percentage gradually decrease, approaching 0. The optimization effect when fine-grained probability is 0.9 is the best. Re-test after change the Client and reduce number of OSTs, the data still show the same trends, indicating that for different Client and different OST amount, the FGF-LRU algorithm could optimize the system's overall I/O performance.

Test for the average fine-grained access response time of read/write operations shows that, for a certain fine-grained probability, the average response time after optimization is less than the average response time before. When the fine-grained probability decreases, the average response time of both before and after optimization increase gradually, and the pre-post response time difference and optimization percentage decreases gradually. The trends of read/write response time's increasing along with the fine-grained probability's decreasing after optimization are greater than the trends before optimization, because that along with the fine-grained probability's decrease, the probability of network access requirement and disk read/write requirement increase for every fine-grained access, the response time tend to get close to the value of before optimization. When the fine-grained probability is 0.9, the optimization effect is the best. The FGF-LRU algorithm significantly improves the performance under big fine-grained access ratio; the bigger the fine-grained access ratio is, the better the optimization effect is. And when the fine-grained access ratio gets smaller, the optimized

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

performance gets closer to the original performance. This is reasonable since this algorithm is designed for the performance optimization of the situation that fine-grained access repeats; if fine-grained access takes smaller ratio, the optimization effect of this algorithm would be weaker. Therefore, when fine-grained access ratio is small, the system is more able to bring the performance advantage of bulk sequential data access handling of parallel system. And when fine-grained access ratio is big, the FGF-LRU algorithm can compensate the performance shortage of parallel system on small piece data handling.

7. CONCLUSIONS

This paper designed and realized a fine-grained first LRU (FGF-LRU) page reclaim algorithm to improve the fine-grained I/O performance of Lustre parallel file system. The aim of the FGF-LRU algorithm is to retain the fine-grained access pages in the Cache of OST and Client, so that the sinking of fine-grained access pages would be slowed down, and the time of fine-grained access pages stay in the main memory will be extended, in order to reduce the number of disk accessing, thus cut down the cost of the redundant accesses to disk. Test of the algorithm confirms that the FGF-LRU algorithm achieved the design purpose.

REFRENCES:

[1] Lustre: a scalable high-performance file system [EB/OL]. Cluster File System, Inc. http://www.lustre.org/docs/whitepaper.pdf. 2002.

[2] Peter J. Braarn. The Lustre Storage Architecture, Cluster File Systems Inc..ftp://crimson.ihg.uni-duisburg.de/Linux/filesys/Lustre/lustre.pdf. 2004.08

[3] Phoenix, Arizonna. Lustre: A scalable high performance distributed file system. Supercomputing 2003, ACM/IEEE Conference. Piscataway, NJ: IEEE, 2003, pp. 15-40.

[4] Li, Zhu., Zhou, Enqiang, Liao Xiangke. Filter Cache: A Method for Improving I/O Performance of Lustre File System [J]. Journal of Computer Research and Development 2009,46:71-77.

[5] Huo Yanmei, Yang Kexin, Hu Liang,Ju Jiubin. Overview of Parallel File System. Journal of Chinese Computer Systems, 2008.29(9), pp. 1631–1636.

[6] Feiyi Wang, Sarp Oral, Galen Shipman. Understanding Lustre File system Internals[R]. National Center for Computational Sciences, 2009.04

[7] Lin Yuzhang. Research on Cache Technique of Parallel File System [D]. Huazhong University of Science and Technology, 2004.

[8] Lin Songtao. Research on Parallel I/O Techniques Based on Lustre File System [D]. National University of Defense Technology, 2004.

[9] James Hendricks, Raja R Sambasivan, Shafeeq Sinnamohideen. Ganger. Improving small file performance in object-based storage, CMU-PDL-06-104. Pennsylvania: Carnegie Mellon University, 2006, pp. 28-41.

[10] Qian Yingjin. Research on Key Issues in Large Scale Clustered File System Lustre [D]. National University of Defense Technology, 2011.

[11] Li Zhu. The Design and Implement of an Improvement for Small File Performance in Distributed File System [D]. National University of Defense Technology, 2008.

[12] Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel [M]. China Electronic Power Press, 2007.