Journal of Computational Information Systems 8: 16 (2012) 1–8
Available at http://www.Jofcis.com

Correlation-aware Prefetching in Fault-tolerant Distributed Object-based File System ⋆

Jiancong TONG, Bin ZHANG, Xiaoguang LIU, Gang WANG∗

Nankai-Baidu Joint Lab, College of Information Technical Science, Nankai University, Tianjin, China

Abstract

Storage systems usually leverage the power of spatial locality based prefetching to improve the read performance. However, it is still a big challenge to initiate the prefetching requests effectively in distributed environment. Since replication and parity are frequently employed techniques to provide high reliability in distributed file systems, this paper presents a new prefetching approach which takes advantage of both data redundancy and the correlation among objects. Two objects distribution algorithms are proposed to guarantee both required fault tolerance and efficiently prefetching by means of maintaining an orthogonal layout. The performance results from experiments on our testbed object-based file system show that these approaches can improve the throughput significantly. The experimental results also show that the overall performance can benefit from the proposed prefetching approach even under the degraded mode.

Keywords: Object-based Storage; Prefetching; Replication; Parity; Orthogonal Layout

1 Introduction

With the rapidly burst of massive data and the continuous growth of cloud storage, the distributed file system has attracted increasing attentions. In distributed systems, frequent cache missing implies massive disk read operations and network transfer, which leads to unacceptable response time. Prefetching technologies, which fetch data from disk in advance before data is really requested, are the commonly used methods to alleviate this kind of latency. Modern Operating systems (e.g. Linux) also employ prefetching strategy to improve the performance [5, 14]. Traditional prefetching strategies typically read consecutive blocks which are physically adjacent to the current requested data [1, 6]. That is, they assume that physically adjacent blocks are also logically adjacent, and user access pattern has good spatial locality. In a distributed object-based file system (e.g. Panasas [13]), however, such assumptions do not stand any more since datum are organized in the form of objects and are distributed over many storage nodes. Therefore, the

⋆This paper is partially supported by the National Nature Science Foundation of China (No. 60903028, 61070014, 61170184, 61170301), Key Projects in the Tianjin Science & Technology Pillar Program (11ZCK-FGX01100).

∗Corresponding author.
Email address: wgzwp@163.com (Gang WANG).

1553–9105 / Copyright © 2012 Binary Information Press
August 15, 2012

objects stored on a single OSD may not be logically consecutive parts of a single file. Obviously, the traditional prefetching methods would no longer work well and novel strategies which are suitable for the distributed systems should be exploited.

On the other hand, as the size of storage systems grows larger and larger, the component failures in server cluster become increasingly frequent. In order to offer a stable service, it is necessary to employ replication and/or parity mechanism in distributed systems to provide high reliability and high availability [3]. However, few references focus on improving prefetching performance through redundant data layout.

In this paper, we extend the correlation-aware prefetching strategy introduced in [19]. The recently proposed method allows applications to pre-fetch data objects according to the logical relationship, avoiding the useless prefetching problem facing by traditional spatial locality based prefetching strategy. When consider the correlation among objects, the multiple cross-nodes prefetching requests can be transformed to a single prefetching operation which request consecutive blocks in a single server. The "aggregation" of prefetching requests draws support from a carefully designed orthogonal data layout for replication and parity based storage systems. The experimental results show that this new strategy increases throughput and decreases network traffic remarkably, even under the degraded mode [20]. In fact, the performance in the rebuild mode [20], as known as reconstruction mode, can also benefit from this orthogonal layout.

The rest of this article is organized as follows. Related work is summarized and discussed in Section 2. Section 3 provides some details about our testbed distributed object-based file system and the objects mapping algorithm used to distribute objects. Section 4 presents the details of our orthogonal layout as well as the correlation-aware prefetching approach based on it. The double-fault-tolerant feature is also explained in this section. Section 5 reports the detailed experimental results as well as the analysis. Finally, Section 6 provides some concluding remarks and discusses future work.

2 Related Work

Prefetching, also known as read-ahead, is a useful technique for alleviating the storage access latency and reducing the response time in storage system. There are numerous works devoted to it. Some of them focused on utilizing the existence of locality (both spatial and temporal) in disk access patterns [10, 11], and tended to fetch consecutive blocks from disk. Others developed different prefetching strategies by investigating history logs and introducing data mining techniques. [2, 9, 16, 18] detected some certain access patterns and used them to determine what data should be pre-fetched.

Since a file may be scattered over multiple storage nodes in distributed object-based file systems, traditional prefetching methods mentioned above may probably result in useless extra I/Os. Sui *et al.* [19] proposed a correlation-aware prefetching strategy which can solve this problem in a certain way. They took the correlation among objects into account and pre-fetched the objects from a single node with the help of replications. Though this method can reduce the network traffic caused by cross-nodes prefetching effectively, it brought some overhead as it required larger (exactly twofold) disk space to store both the data and the replications. Our work extends this correlation-aware idea with the consideration of fault-tolerant and introduces the mechanism of parity. With replication and parity, the system is now double-fault-tolerant while the prefetching

efficacy is still the same as the original method achieved. Moreover, the extra used disk space gives full play to its effect now.

The motivation of replication and parity came from the existing storage system and fault-tolerance approaches. Both Ceph [17] and GFS [8] used replication to maintain system availability and ensure data safety. RAID 1 [15] and RAID 5 [7] are the most classic technologies to provide high reliability. Furthermore, [21] combined mirroring, parity and parity de-clustering approaches to construct more reliable erasure code. Our replication and parity mechanisms are closely related to RAID 1 and RAID 5. Any systems that already have any kinds of replication and/or parity mechanism(s), including RAID 1 and RAID 5, can adopt our prefetching model at a very low cost.

3 Testbed Architecture

We implement the orthogonal layout (with replication and parity) and the prefetching strategy on our previous work, *OFS*, a Ceph [17]-like distributed object-based file system which has been introduced in [19]. In OFS, the client nodes provide the standard file system interface to the users. And the storage nodes, called Object-based Storage Devices (OSDs) [12], are responsible for objects storing and accessing.

The basic unit for data storage in OFS is objects. Every file in OFS is divided into fixed-size objects that are assigned to several OSDs using an *Objects Mapping Algorithm* (OMA) [19], shown as Algorithm 1. Each object is addressed by a unique object ID (128-bit) called 'object identifier' (OID), represented by a quintuple (fsno, fno, offset, type, flag). All the objects of one file share the same value in both 'fsno' and 'fno' fields. The clients and OSDs access objects using the OID.

Objects are distributed over multiple OSDs according to the OMA. As shown in Algorithm 1, an object is first represented by a fingerprint related to its OID (line 3-5). And then its fingerprint is mapped into a segmented 32 bits namespace which has several intervals (line 6-8). After that, this object is assigned to the corresponding OSD which is associated with a particular interval (line 9). The experimental results show that this mapping algorithm distributes objects almost evenly over OSDs (see Table 1), which is the same as [19] claimed.

Algorithm 1: Objects mapping

Input: File f and the number of OSDs osd_num

Output: Distribute the objects of f to the OSDs on which they should resided

1 Divide f into objects set S.
2 **foreach** *object* o *in* S **do**
3   Mark o by corresponding OID *oid*.
4   Compute the MD5 fingerprint *md5sum(oid)* of o.
5     $v \leftarrow md5sum(oid)$ **AND** $0xFFFFFFFF$
6     $region \leftarrow (uint32_t)(\sim 0)$
7     $interval_span \leftarrow \lceil region \div osd_num \rceil$
8     $pos \leftarrow \lceil v \div interval_span \rceil$
9   Distribute o to the *pos*-th OSD.
10 **end**

x	No.1	No.2	No.3	No.4	No.5	No.6	No.7	No.8
2000	265	249	251	263	244	242	233	252
4000	525	493	513	522	489	490	467	500
6000	766	755	772	768	734	747	705	752
8000	1033	992	1006	1020	983	1014	941	1010
4000	607	557	597	584	546	563	545	#
4000	695	669	689	653	635	658	#	#
4000	805	820	827	762	785	#	#	#
4000	1018	1035	979	967	#	#	#	#

Table 1: Evenly distribution of objects over multiple OSDs according to Algorithm 1. The number of OSDs in the upper part of the table is fixed (which is 8), while in the lower part it varies. x denotes the total amount of objects, and the value in the array denotes the number of objects that resided on the j-th OSD (No. j).

4 Prefetching in Fault-tolerance Systems

4.1 Correlation-aware prefetching with replication system

A "relationship-clustering" replication idea for OFS has been presented in [19]. OFS keeps a replica (*replica object*) for each object (*original object*) and guarantee that these two selfsame copies are resided on different OSDs. A replica object is called as the *twin* of its original one, and vice versa. Both the twinned copies of an object (that is, original and replica) are assigned to their corresponding OSDs using the same OMA, that is, Algorithm 1. The only difference is that it is the file identifier $oid.fno$ instead of object identifier oid to be considered during the mapping procedure of the replica objects. Therefore, replicas with the same "fno" field in OID, i.e., belonging to the same file, are aggregated in a single OSD. The rectangular dotted line region of Figure 1 shows an example. X_i denotes the i-th original object of file X and X_i' denotes the corresponding replica object. All the replica objects of file A resided on $osd1$ (the *replica node* of A) while the original objects distributed over other OSDs (the *original nodes* of A). Two modified OMA are proposed to extend the primitive one for maintain such an *orthogonal layout.*

This layout aggregates the replicas of logically adjacent objects, so only one request needs to be initialized for a single prefetching operation, which is called correlation-aware prefetching strategy. When a read request is initiated by a client, the prefetching is performed. Note that it is the serving OSD rather than the client to initiate the prefetching requests in OFS. The serving OSD dose not initiate multiple prefetching requests to many OSDs which have the logically contiguous objects. Instead, it directly initiates a single prefetching request to the replica node of the current required file. While the serving OSD acknowledges the client with the current required object, the replica OSD continues its prefetching. These two procedures are overlapped, forming a pipeline, which reduces idle time significantly. Then the following read requests can be properly (controlled by a dynamic sliding window mechanism, proposed in [19]) redirected to the replica OSD, where the objects have already resided in the main memory by previous prefetching.

4.2 Correlation-aware prefetching with parity system

Though the replication aggregating method does reduce the number of prefetching requests and disk access latency greatly, it requires 2-fold disk space to store both the data and the replicas. Since it is common in storage systems to employ both replication and parity techniques to provide high reliability, we can extend the replication idea to a fault-tolerant system by introducing the parity.

For a certain file, the original objects are organized into *parity stripes*. Every stripe has its own *parity object* that is the *XOR* sum of the data objects in the same stripes. All stripes again form an orthogonal layout, that is, a file's parity objects are stored in a particular OSD (*parity node* of that file) while all its original and replica objects stored in other OSDs. To provide double-fault-tolerant, the parity OSD and the replica OSD of one file should not be the same node.

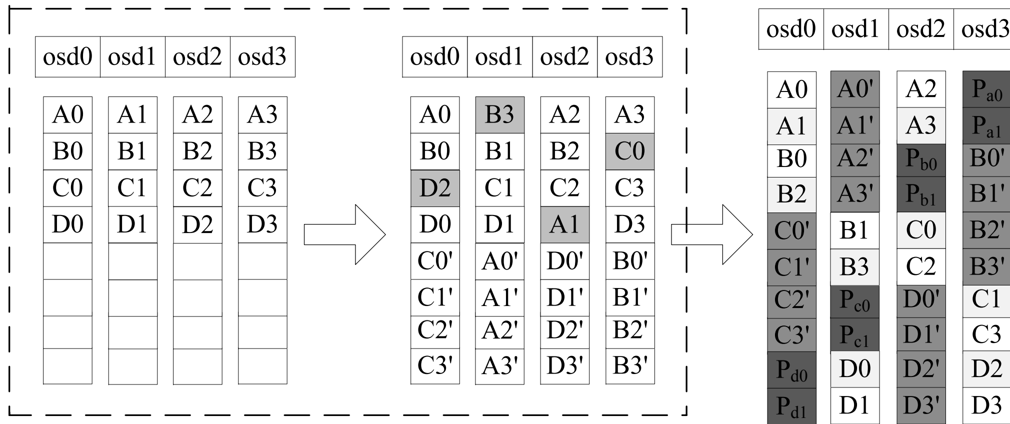

Fig. 1: Orthogonal layout with parity

The right part of Figure 1 shows an example of orthogonal distribution with replication and parity. The meaning of X_i and X_i' are the same as in Section 4.1, while P_{xi} denotes the parity object of the i-th parity stripe of file X. In this example, there are four files A, B, C, D. We can see that object A_0 together with A_2 forms a parity stripe, while P_{a0} is the parity object of this stripe. Also we can find that all parity objects of file A are assigned to *osd*3, while all replica objects are assigned to osd1. Like RAID-5, the parity OSDs of files are chosen in a round robin way that leads to even parity distribution, therefore even updating load distribution. However, compared with the layout without replication and parity (the most left part of Figure 1), some original objects must be re-mapped in order to avoid superposing with their twins in a same OSD. Thus, we extend the *migration* strategy and *repartition* strategy [19] to insure the orthogonality.

- **Repartition Strategy** This strategy repartitions the 32 bits namespace mentioned in Section 3 into $N-2$ segments as shown in Figure 2. Here N equals to the number of OSDs, while 2 corresponding to the preserved OSDs for replication and parity. In other words, the replica node and the parity node are excluded during the mapping of original objects. By applying the original OMA under the new partitioned namespace, each original object is mapped to an OSD except its replica node (marked as R) and the parity node (marked as P). Figure 3 illustrates an example.

- **Migration Strategy** In contract with repartition strategy, this method dose not re-map all the original objects. Instead, only the original objects that would be assigned to the R node and the P node by the original OMA will be reassigned. As shown in Figure 4, eight original objects reside on *osd*1 and *osd*3 are migrated to other original OSD evenly.

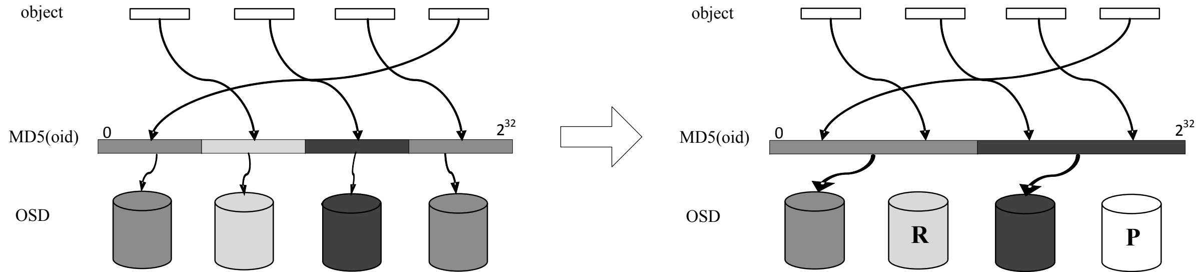

Fig. 2: Namespace original vs. Namespace with replication and parity

osd0	osd1	osd2	osd3
2	0	1	3
7	5	4	6
9	8	13	10
15	12	14	11

osd0	osd1	osd2	osd3
2	0' 8'	0	P0
1	1' 9'	3	P1
7	2' 10'	5	P2
4	3' 11'	6	P3
9	4' 12'	8	P4
13	5' 13'	10	P5
15	6' 14'	12	P6
14	7' 15'	11	P7

Fig. 3: Repartition strategy with parity

osd0	osd1	osd2	osd3
2	0	1	3
7	5	4	6
9	8	13	10
15	12	14	11

osd0	osd1	osd2	osd3
0	0' 8'	1	P0
2	1' 9'	4	P1
3	2' 10'	5	P2
7	3' 11'	6	P3
9	4' 12'	11	P4
8	5' 13'	12	P5
10	6' 14'	13	P6
15	7' 15'	14	P7

Fig. 4: Migration strategy with parity

Note that there is no need to worried about the load imbalance problem that may raised by using these two new re-mapping strategies. Recall Table 1, the objects will still be (almost) evenly distributed over $N-x$ OSDs in spite of x nodes are excluded out from the whole cluster (N nodes in total). Then with this new load-balancing orthogonal layout, the correlation-aware prefetching strategy can work the same way as described in Section 4.1. Let S denote the amount of the original data. Layout in Section 4.1 costs $2S$ disk space while the new layout costs $(2+1/(N-2))S$. That is, the reformed layout achieves the same efficacy and extra fault-tolerance than the former one with only $S/(N-2)$ extra disk space cost.

5 Experimental Results

In this section, we present evaluation results for our correlation-aware prefetching strategy (*COR* for short) under the reformed orthogonal layout, and compare it with a baseline approach which considers the spatial locality only (denoted as *SPA*). To maximize the effectiveness of prefetching, the dynamic sliding window mechanism mentioned in Section 4.1 is employed in the following experiments. All experiments are conducted on a Gigabit Ethernet connected cluster of nine nodes. One of them acts as the client node, while other 8 nodes perform as OSDs. Each machine has a single-core 3.06GHz Intel Celeron processor and a 512MB memory, running Red Hat Enterprise Linux AS 4.

For description convenience, we define these terms: "*cor*" and "*spa*" stand for OFS with our correlation-aware prefetching method and with the spatial locality based prefetching method

respectively, both of which employ replication mechanism only. "*parity*" stands for the *normal read* (read operation in a non-fault system) in OFS which employs not only replication but also parity mechanisms. In contrast, "*degrade*" stands for the *degraded-mode read*, which means read performance under disk failure(s). Without special description, all the read operations in the experiments are sequential read.

We use a modified Bonnie [4] as benchmark to generate sequential and random workload. In each test, five 100MB files are written first and a single client reads all five files in a round robin way to test the sequential or random read performance. Each file is composed of 800 objects, that is, the object size is 128KB. Though the request generated by Bonnie is of size 16KB, each read/write operation in OFS deals with exactly an object, namely is of size 128KB (this parameter can be tuned when system initializes). Each result is the average of 5 runs.

We start by investigating how the cache size would affect the performance. We use a fixed window size (i.e. 10) and perform the experiment with five different cache sizes. Figure 5 shows that COR method outperforms SPA method regardless the cache size. And the proposed prefetching strategy can improve the throughput by up to 96% compared to the system without prefetching. Figure 6 illustrates the same trend too. When there is no disk failure occurring, the parity is transparent to read requests processing. So adopting parity mechanism does not bring any side-effect on the overall throughput. This conclusion can be easily perceived by comparing the corresponding data of normal read in the following two figures.

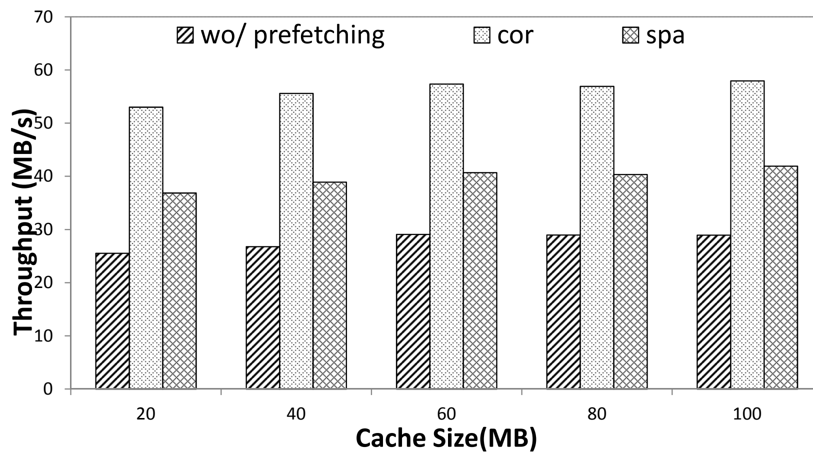

Fig. 5: The impact of cache size (without parity)

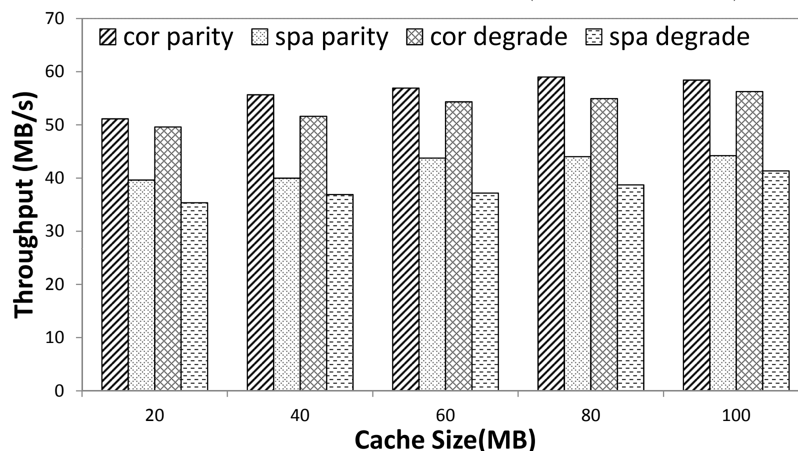

Fig. 6: The impact of cache size (with parity)

Next, we try to figure out the impact of the window size (a key parameter in dynamic sliding window mechanism mentioned above) on the overall system performance. The cache size is set to 40MB in this test. The window size is set constant in every single run and varies from 5 to 30 in different runs. We test the performance of both sequential read and completely random read, and the results are presented in Figure 7 and Figure 8 respectively. When the window size is not larger than 20, the throughput of COR method keeps increasing as the window size grows in both two figures. However, when the window size exceeds 20, the performance of sequential read stays unchanged while that of random read still getting improved (more slighter, though). The performance of SPA method behaviors in a different but more interesting way. The throughput begins to drop as the window size exceeds 20 in sequential read test, while the performance of random read test, in contrast, still increasing slightly.

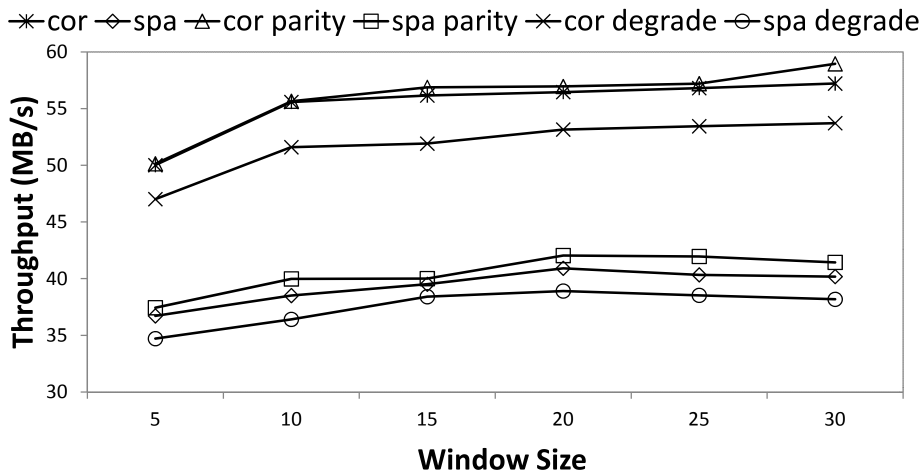

Fig. 7: The impact of window size (sequential read)

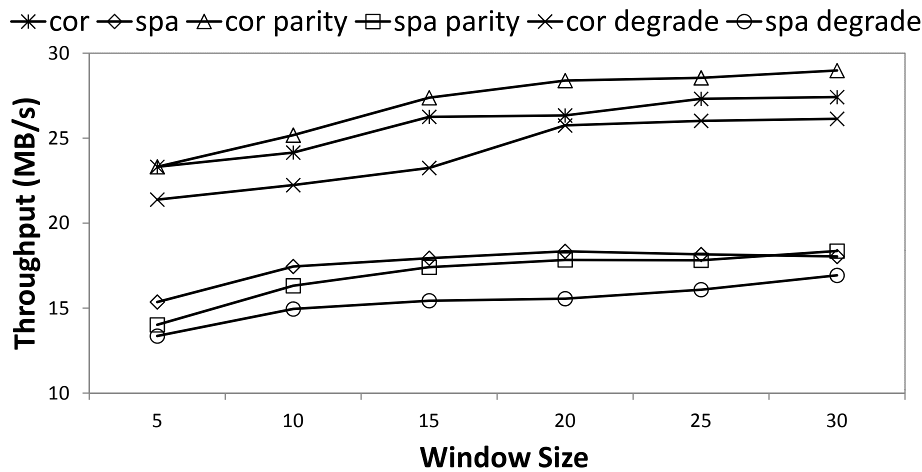

Fig. 8: The impact of window size (random read)

It is self-evident that the random read benefits more from larger prefetching window than sequential read. Since the contiguous requests of random read are not aiming at the logically adjacent objects, the more objects the system pre-fetched, the better performance the system would achieve. The reason that sequential read performs worse with larger window is also understandable. Because a mass of useless prefetching initiated by SPA could cause the memory cache occupied by irrelevant data, which leads to the bad performance.

In order to examine the scalability of OFS, we rerun the experiments with different number of storage nodes (from 2-8). Note that the 2-nodes case corresponds to OFS with only replication but not parity, since the orthogonal layout with replication and parity requires at least 3 nodes. The cache size is also set to 40MB and the window size is set to 20 at the beginning of the test (which will automatically adjusted during the whole running [19]). As illustrated in Figure 9, the correlation-aware prefetching always exhibits the better performance, regardless of the system size. Besides, it is worth noting that the gap between the performance of "cor"-class and "spa"-class tests keeps increasing as the system size grows larger. This observation confirms the idea that the more distributive the system is, the more useless the SPA method performs.

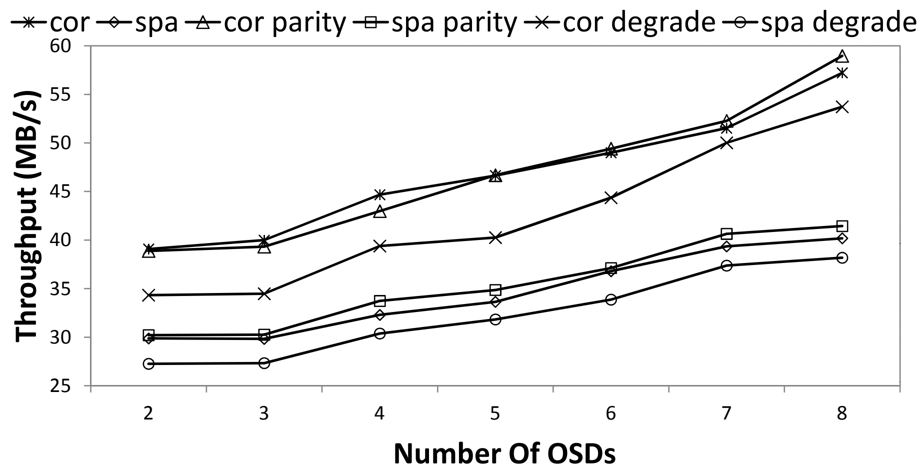

Fig. 9: Scalability

Figure 6, 7, 8, 9 together demonstrate that even under the degraded mode, COR method still performs better than SPA. That is, even under degraded mode can the overall performance benefited from the novel prefetching approach. In OFS, every OSD acts as the replica OSD for some certain files and the parity OSD for some other files, while it plays the role as one of the original OSDs of the remain files. As a result, when one or two disk fails, only a few files lost their replica OSDs. The other files can still be involved in the correlation-aware prefetching. Therefore, the system can accomplish the degraded mode read as long as there are no more than two disks fail, and still benefited from the prefetching approach.

6 Conclusions and Future Work

This paper extends the recently proposed correlation-aware prefetching strategy to fault-tolerant distributed object-based file systems. In order to maintain the prefetching effectiveness as well as the double-fault-tolerance, we carefully reform the orthogonal layout and proposed two objects distribution algorithms. Furthermore, this work extends the single-fault-tolerance to double-fault-tolerance with little overhead. The experimental results show that the novel prefetching strategy together with the proposed data layout exhibits much better performance than the traditional spatial locality based approach. Besides, the evaluation also demonstrates that the performance of degraded mode can benefit from such prefetching method too.

This research can be extended in several directions. For example, in order to alleviate the writing workload of replica and parity node, the replication and/or parity can be divided to

several OSDs instead of just one single OSD. As a result, the orthogonal layout needs to be carefully redesigned. Moreover, by employing data mining techniques on disk accessing log, other correlations among objects can be investigated. From the point of fault-tolerance, more work can be done to achieve higher reliability and better load balance, like introducing more complicated encoding algorithm and parity de-clustering approach, etc.

References

[1] Albers, S. and Buttner, M. (2003) "Integrated prefetching and caching in single and parallel disk systems", *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, 7 – 9th June, San Diego, California, USA, pp. 109 – 117.

[2] Amer, A., Long, D. D. E., and Burns, R. C. (2002) "Group-Based Management of Distributed File Caches", *Proceedings of the 22nd International Conference on Distributed Computing Systems*, 2 – 5th July, Vienna, Austria, pp. 525 – 534.

[3] Bianca, S., and Garth, A. G. (2006) "A large-Scale study of failures in high-performance computing systems", *Proceedings of the International Conference on Dependable Systems and Networks*, 25 – 28th Jun, Philadelphia, PA, pp. 249 – 258.

[4] Bonnie (1990). Obtained through the Internet: http://www.textuality.com/bonnie/.

[5] Butt, A. R., Gniady, C., and Hu, Y. C. (2007) "The performance impact of kernel prefetching on buffer cache replacement algorithms", *IEEE Transactions on Computers*, Vol. 56, No. 7, pp. 889 – 908.

[6] Cao, P., Felten, E. W., Karlin, A. R., and Li, K. (1995) "A study of integrated prefetching and caching strategies", *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 15 – 19th May, Ottawa, Ontario, Canada, pp. 188 – 197.

[7] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson D. A., (1994) "RAID: high-performance, reliable secondary storage", *ACM Computing Surveys (CSUR)*, Vol. 26, No. 2, pp. 145 – 185.

[8] Ghemawat, S., Gobioff, H., and Leung, S. T. (2003) "The Google file system", *ACM SIGOPS Operating Systems Review*, Vol. 37, No. 5, pp. 29 – 43.

[9] He, S., Feng, D., Li, C., and Yuan, Y. (2009) "A Rule-Based Prefetching Approach for Object-Based Storage Device", *International Journal of Distributed Sensor Networks*, Vol. 5, No. 1, pp. 55 – 55.

[10] Jiang, S., Ding, X., Chen, F., Tan, E., and Zhang, X. (2005) "DULO: an Effective Buffer Cache Management Scheme to Exploit both Temporal and Spatial Locality", *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, 14 – 16th Dec, San Francisco, CA, USA, pp. 101 – 114.

[11] Liu, H., and Hu, W. (2001) "A Comparison of Two Strategies of Dynamic Data Prefetching in Software DSM", *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, 23 – 27th Apr, San Francisco, CA, USA, pp. 62 – 67.

[12] Mesnier, M., Ganger, G. R., and Riedel, E. (2003) "Object-based storage", *Communications Magazine*, Vol. 41, No. 8, pp. 84 – 90.

[13] Nagle, D., Serenyi, D., and Matthews, A. (2004) "The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage", *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, 6 – 12th Nov, Pittsbutgh, PA, USA, pp. 53 – 62.

[14] Pai, R., Pulavarty, B., and Cao, M. (2004) "Linux 2.6 Performance Improvement through Readahead Optimization", *Proceedings of the Linux Symposium*, 21 – 24th July, Ottawa, Ontario, Canada, pp. 391 – 401.

[15] Patterson, D. A., Gibson, G., and Katz, R. H. (1988) "A case for redundant arrays of inexpensive disks (RAID)", *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 1 – 3th June, Chicago, Illinois, United States, pp. 109 – 116.

[16] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. (1995) "Informed Prefetching and Caching", *Proceedings of the fifteenth ACM symposium on Operating systems principles*, 3 – 6th Dec, Copper Mountain, Colorado, United States, pp. 79 – 95.

[17] Sage, A. W., Scott, A. B., Ethan, L. M., Darrell, D. E. L., and Carlos, M. (2006) "Ceph: a scalable, high-performance distributed file system", *Proceedings of the 7th symposium on Operating systems design and implementation*, 6 – 8th Nov, Seattle, Washington, pp. 307 – 320.

[18] Soundararajan, G., Mihailescu, M., and Amaza, C. (2008) "Context-aware prefetching at the storage server", *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 22 – 27th June, Boston, Massachusetts, pp. 377 – 390.

[19] Sui, J., Tong, J., Wang, G., and Liu, X. (2010) "A Correlation-Aware Prefetching Strategy for Object-Based File System", *Proceedings of the 10th international conference on Algorithms and Architectures for Parallel Processing*, 21 – 23th May, Busan, Korea, pp. 236 – 245.

[20] Thomasian, A., and Menon, J. (1994) "Performance analysis of RAIDS disk arrays with a vacationing server model for rebuild mode operation", *Proceeding of the 1994 10th International Conference on Data Engineering*, 14 – 18th Fer, Houston, Texas, USA, pp. 111 – 119.

[21] Wang, G., Liu, X., Xie, G., and Liu, J., (2007) "Constructing double and triple-erasure-correcting codes with high availability using mirroring and parity approaches", *Parallel and Distributed Systems*, 5 – 7th Dec, Hsinchu, Taiwan, pp. 1 – 8.