A Correlation-Aware Prefetching Strategy for Object-Based File System[⋆]

Julei Sui, Jiancong Tong, Gang Wang, and Xiaoguang Liu

Nankai-Baidu Joint Lab, College of Information Technical Science, Nankai University
94 Weijin Road, Tianjin 300071, China
{nkujulei,lingfenghx}@gmail.com,wgzwp@163.com,liuxg74@yahoo.com.cn

Abstract. The prefetching strategies used in modern distributed storage systems generally are based on temporal and/or spatial locality of requests. Due to the special properties of object-based storage systems, however, the traditional tactics are almost incompetent for the job. This paper presents a new prefetching approach, which takes the correlationship among objects into account. Two orthogonal replica distribution algorithms are proposed to aggregate prefetching operations. A moving window mechanism is also developed to control prefetching. We implement these approaches in our object-based file system called NBJLOFS (abbreviated for Nankai-Baidu Joint Lab Object-based File System). The experimental results show that these approaches improves throughput by up to 80%.

Key words: object-based storage, prefetching, object duplication, orthogonal layout

1 Introduction

With the continuous growth of storage device capacity and the rapid development of applications, traditional block-based storage systems can no longer meets the demand. Object-based storage technology emerged and has attracted increasing attention. Generally, "object-based" means that the basic storage unit is object instead of block. An object is a combination of file data and a set of attributes [1]. In a distributed object-based file system, a file may be divided into many objects, which are distributed over several storage nodes.

In modern computer systems, I/O time often dominates the total running time. Cache and prefetching technologies are the standard way to alleviate the performance gap between disk and main memory/CPU. Traditional prefetching strategies typically read additional blocks physically adjacent to the current read request. That is, they assume that logically adjacent blocks are also physically adjacent. However, these strategies do not work in distributed object-based

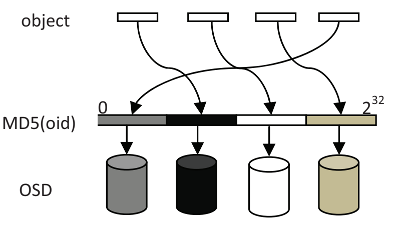
[⋆] Supported partly by the National High Technology Research and Development Program of China (2008AA01Z401), NSFC of China (60903028), SRFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCYBJC13000).

storage systems because objects are distributed over many storage nodes. Moreover, these strategies can not deal with prefetching based on other relationship among objects. This paper puts forward an innovative prefetching strategy taking objects' correlation into account. It gains "prefetching aggregating" by an orthogonal object duplication layout. The experimental results show that this new strategy increases throughput and decreases network traffic.

The rest of this paper is organized as follows. In Section 2, we briefly describe NBJLOFS and namespace splitting algorithm. In Section 3, we introduce our prefetching approach together with details of our implementation. In section 4, we present the experimental results. Section 5 discusses related work and Section 6 concludes the paper.

2 NBJLOFS

NBJLOFS is a distributed object-based file system based on IP-SAN [2]. It employs FUSE [3] as filesystem interface and Berkeley DB [4] as storage infrastructure. Every file in NBJLOFS is split into objects. Each object is uniquely represented by a quintuple (fnso, fno, offset, type, flag) called object identifier (OID). NBJLOFS has no metadata servers. The clients provide a standard file system interface to the users, and the storage nodes (called Object-based Storage Devices, OSDs for short) are in charge of object storing and accessing. When a file is stored, the system splits it into fixed-size objects and distributes them over OSDs. When a read request arrives, the client node dispatches it to the proper OSD. The OSD will acknowledge the client with the required object and maintains a copy in its local cache. Objects are distributed according to a namespace splitting algorithm. As shown in Fig. 1, a 32 bits identifier space is divided into several segments. Each segment corresponds to a unique OSD. Every object is mapped to a value in the identifier space using MD5 algorithm (Message-digest Algorithm 5). Then the object is assigned to the corresponding OSD.



```
Procedure map_object(object_id , osd_num)
    x ← MD5(object_id) & 0xffffffff
    region ← ( uint32_t )( ~0 )
    interval_span ← region / osd_num
    position ← x / interval_span
    Distribute the object to osd[position]
end procedure
```

Fig. 1. Namespace splitting algorithm

3 Correlation-Aware Prefetching

In a Distributed Object-based File System, objects are dispersed. The objects stored on a single OSD may not be logically consecutive parts of a single file according to the object distributed algorithm mentioned above. So the traditional prefetching strategy that reads additional disk blocks adjacent to the current request no longer works. Instead, we should fetch logically related objects from different OSDs. Since objects are distributed over OSDs, inevitably, lots of prefetching requests are launched to many OSDs to prefetch a batch of logically adjacency objects. Therefore, as the system size increases, the network traffic will dramatically increases. In order to solve this problem, this paper presents a “file-centralized” duplication strategy. This strategy aggregates the replicas belonging to the same file, so only one request is need to be issued for a prefetching operation.

3.1 Object Duplication

We do not create mirror for each OSD like RAID-1 [5]. Instead, we adopt object-oriented duplication. Moreover, we aggregate replicas according to their correlationships. In this paper, we consider the “file-belonging” correlationship, that is, replicas with the same “fno” field in OID are aggregated. Note that we can easily aggregate replicas using other correlationships.

For each object, we make and maintain a replica for it and guarantee that these two copies of the object are stored on different OSDs. For clarity, we call them the *original object* and the *replica object* respectively. The former is assigned to a OSD using the namespace splitting algorithm mentioned in Section 2. The replica is assigned to the OSD determined by the MD5 digest of its file identifier (“fno”). As objects belonged to the same file share the same value in field “fno”, the replica objects of a file aggregated in a specific OSD and the original objects are distributed over other OSDs. For a certain file, we call the specific OSD where all its replica objects aggregated as its *replica OSD* or *replica node*, and other OSDs as its *original OSDs* or *original nodes*.

Fig. 2 shows an example of orthogonal distribution. We use Xi denote the i-th original object of file X and Xi' denote the replica. In this example, there are four files A, B, C and D. Note that in NBJLOFS the objects are not really indexed in this way, the indices are just used to illustrate this problem simply. We can see that all replicas of file A are assigned to OSD1, and its original objects are distributed over other OSDs. From Fig. 2, we can find that the replica objects of a certain file didn’t reside with any of its original objects in the same OSD. We call this way of distribution as *orthogonal layout.* The original objects and replicas of other files are distributed in a similar way. Nevertheless, guaranteeing orthogonal distribution is not straightforward. We can see that, compared with the layout without duplication, some objects must be redistributed to avoid conflicting with their replicas. We extend namespace splitting algorithm in two ways for this purpose.

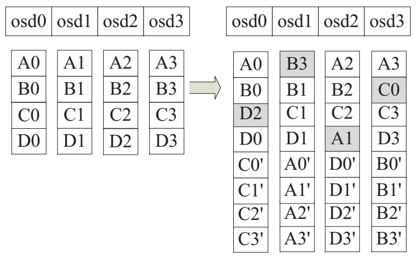

osd0 osd1 osd2 osd3
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3
osd0 osd1 osd2 osd3
A0 B0 D2 D0 C0' C1' C2' C3'
B3 B1 C1 D1 A0' A1' A2' A3'
A2 B2 C2 A1 D0' D1' D2' D3'
A3 C0 C3 D3 B0' B1' B2' B3'

Fig. 2. Orthogonal distribution

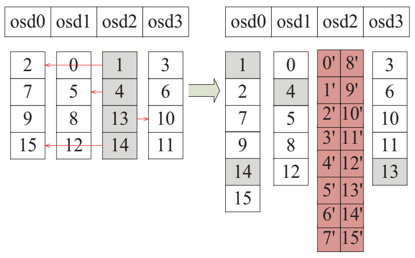


Fig. 3. Migration strategy

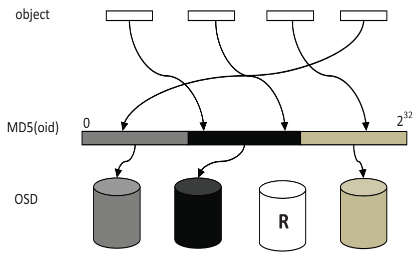


Fig. 4. Repartition namespace

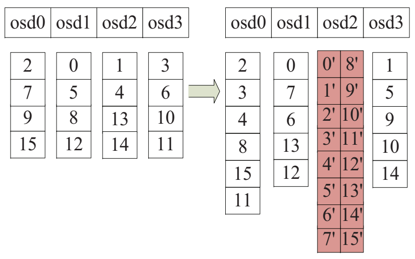


Fig. 5. Repartition strategy

Migration Strategy This strategy does not remap all original objects. Instead, it just redistributes those original objects on the replica node at the very start. Fig. 3 illustrates this strategy. The replica node is OSD2 in this example. We can see that the objects originally stored on ODS2, i.e., 1, 4, 13, 14 are migrated to other OSDs.

Repartition Strategy This strategy repartitions the 32 bits identifier namespace into several segments. As shown in Fig. 4, the number of segments is just one less than the number of OSDs. For each file, each segment corresponds to one of its original nodes. By applying the namespace splitting algorithm, each original object is mapped to a unique segment. Then the original object is assigned to the corresponding original OSD. And all the replica objects are stored in its exclusive replica node(mark as R in Fig. 4). The replica node of the file is not involved in the process of namespace splitting. In other words, the replica OSD is simply omitted when we distribute the original objects. Therefore original-replica conflicts never occur. Fig. 5 illustrates this strategy.

At first glance, it seems that there is a serious load-imbalance among OSDs because of replica objects' aggregation and original objects' dispersion of a single file. However, from the perspective of the whole system, since there are huge number of files, the load is essentially balanced. For each OSD, it plays not only the role of the replica node of some files but also the role of a original node of many other files.

Our experimental results show that both repartition and migration strategy distributes original and replica objects over OSDs evenly. However, if we create a duplication for an existing single-copy system, the repartition strategy must redistribute all objects, while the migration strategy only redistributes the objects on their replica nodes. So, we select the latter as our distribution algorithm.

3.2 Moving Windows

Prefetching excessively or incorrectly certainly could be helpless, even harmful to system efficiency. In order to maximize the virtue of prefetching, we introduce the dynamic window mechanism. We treat the batch of to-be-prefetched objects as a window, and denote the number of objects in the window by *window_size*. This idea comes from the moving window concept in TCP/IP protocol. The window extent will be dynamically changed according to the on the fly object rather than keep stationary all the time. Here are also two alternative strategies: *forward window*, which only prefetches the objects that following the current request, and *wing window*, which prefetches both the previous and the following objects. These two strategies have the same window_size. Fig. 6(a) and Fig. 6(b) illustrate the two strategies respectively, where the dark areas are prefetching windows. Since spatial locality involves both forwards and backwards, we select the wing window strategy as the moving windows mechanism of NBJLOFS. Both "wings" have the same length, that called the *wing_length*. This implies that the wing_length is equal to half of the value of window_size.

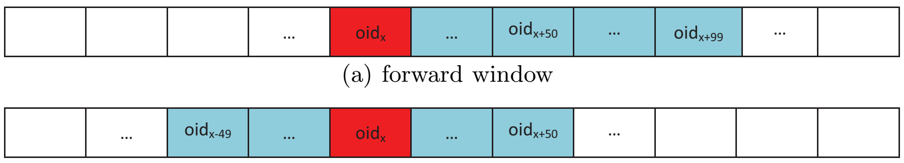


Fig. 6. Two moving window strategies($window\_size$ = 100)

In NBJLOFS, the replica node of a file maintains a unique wing window for each client that accesses this file, i.e., we have not implemented data sharing window. Each client maintains a wing window for every file it accesses. The windows in replica nodes and clients are always consistent. Any time a window in the OSD changed, it notifies the corresponding client to synchronize. Objects in the same file can be identified by their in-file offsets. We call the difference of two objects' offsets *distance*. For each window, we call the central object *pivot*. For example, the oid_x in Fig. 6(b) is the pivot.

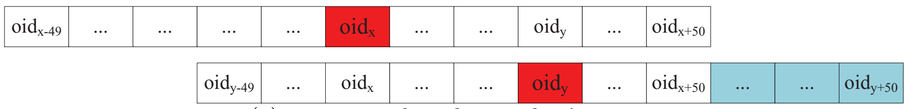
On the client side, whenever an object is accessed, NBJLOFS will determine whether it is within the wing window extent or not by comparing the wing_length with the distance between the pivot and the demanded object. If the latter is numerically smaller, which means that the object has been already prefetched from disk in the replica node, the client node will then send a request to the replica node, and correspondingly moves the window when reply is received. On the contrary, the client node will send a request to the original node.

Whenever an original OSD receive an access request, it implies that the required object has not been prefetched. The original OSD will read the required object from disk, send it back to the client and issue a prefetching request to the replica node.

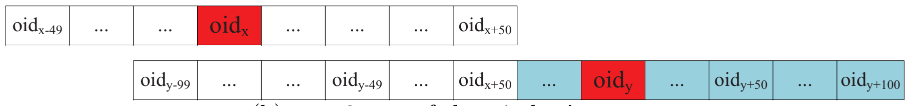
On the replica node, if a request arrives, there are two cases.

i) The request is received from a client node. This implies that the required object is within the window extent, which means that the previous prefetching operation works. So the window_size will remain unchanged. The required object will be sent to the client node. This object is chosen as the new pivot, and the difference between the new and the old windows, i.e., the objects within the new window but out of the old window, will be prefetched from disk. Fig. 7(a) shows the change of the window, where the dark area denotes the newly prefetched objects.

ii) Otherwise, the request is received from an original node. This implies that the distance is larger than the wing_length, showing that the current window is not wide enough. So the replica node doubles the window_size. In our implementation, the initial value of window_size is 1, and its maximum is limited to 40 objects (amount to 5MB data). Similar with case i, the required object is chosen as the new pivot and the difference of the new and the old window is prefetched from disk. Fig. 7(b) shows this case. In addition, the replica node will inform the related client to synchronize the window.


oid_{x-49} ... ... oid_x ... ... oid_y ... oid_{x+50}
oid_{y-49} ... oid_x ... ... oid_y ... oid_{x+50} ... ... oid_{y+50}

(a) case 1: within the window's range


oid_{x-49} ... ... oid_x ... ... oid_{x+50}
oid_{y-99} ... ... oid_{y-49} ... oid_{x+50} ... oid_y ... oid_{y+50} ... oid_{y+100}

(b) case 2: out of the window's range

Fig. 7. Two window changing cases

3.3 Analysis

In our prototype, a traditional spatial locality based prefetching approach was also implemented for comparison. It is very similar to the read-ahead strategy in Linux kernel. We call it SPA. Now we analyze this algorithm and our correlation-aware prefetching algorithm (COR for short).

Assume that the window size is W ($W \geq 1$) and the number of OSDs is N ($N > 1$). For simplicity, assume that $N > W$. When an OSD receives an access request from client, it should prefetch W objects. For SPA, since objects are not duplicated, the OSD has to issue W prefetching requests to other W OSDs assuming that the to-be-prefetched objects are evenly distributed. In contrast, COR issues only one prefetching request to the replica node. In a single-user system, COR may have no advantage over SPA if only a single file is accessed simultaneously. After all, SPA also offers timely prefetching. However, if the two algorithms are deployed in a multi-user system and many files are accessed simultaneously, COR will show its superiority. Since it induces lighter network traffic than SPA, there would be less occurrence of network saturation. Moreover, since COR stores all replicas of a file in a single OSD and just these replicas are prefetched, disk operations induced by prefetching requests are more likely to be large sequential read operations. In contrast, disk operations in SPA system are all small discontinuous (random) read operations. Our experimental results show that these two advantages of COR lead to significant performance advantage over SPA.

4 Experiment

4.1 Experimental environment

All experiments were performed on a cluster of seven single-core 3.06GHz Intel Celeron nodes. Each machine has 512MB of memory and a 80GB hard disk. Each node runs Red Hat Enterprise Linux AS 4. All the nodes are connected by a Gigabit Ethernet. One of them acts as the client node, while other six nodes act as OSDs.

4.2 Performance Evaluation

We used a modified Bonnie to generate sequential and random workload. In each test, we first wrote five 100MB files, then tested sequential or random read performance. Each data point is the average of 5 runs. Each file was composed of 800 objects, that is, the object size was 128KB. The request generated by Bonnie is of size 16KB. However, each read/write operation in NBJLOFS deals with a complete object, namely is of size 128KB.

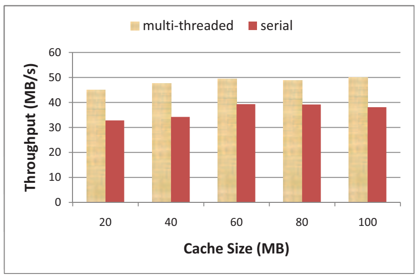


Fig. 8. Serial vs. Multi-threaded

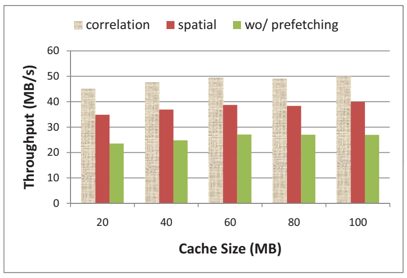


Fig. 9. The Impact of Cache Size

In pervious version of NBJLOFS, common requests and prefetching requests are processed in serial. This strategy apparently does not make full use of CPU. So we implemented a double-threaded version. One thread processes only common requests and another processes prefetching request. Fig. 8 shows the remarkable improvement of read performance. In this test, a single client read all five files in a round robin fashion.

Next, we tested the impact of the cache size on the read performance. We used a fixed window size 10 and files are still read in a round robin fashion. Fig. 9 shows that no matter how big the cache, correlation-aware prefetching outperform simple spatial locality based prefetching and NBJLOFS without prefetching. For a certain system, the minor performance difference is induced by cache replacement.

We tried to figure out how the window size will impact the overall system efficiency. Cache size was set to 40MB in this test (we just choose it randomly). The window size was set changeless in every single run and varied from 5 to 25 in different runs. Beside sequential read test mentioned above, we also tested completely random read. The 5 files were also read in a round robin fashion. As shown in Fig. 10, the throughput keeps increasing as the window size grows until reaches 15. Fig. 10 also shows that the random read benefits from big prefetching window too.

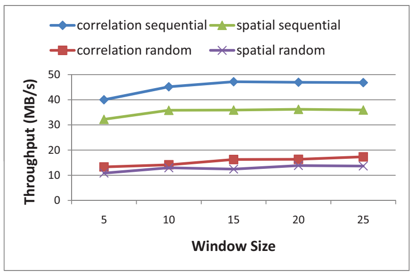


Fig. 10. The Impact of Window Size

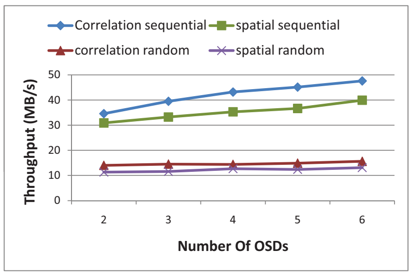
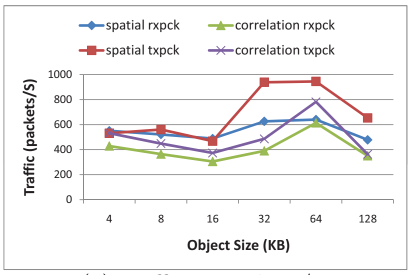
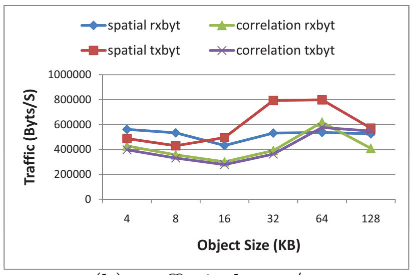


Fig. 11. Scalability



(a) traffic in packets/s



(b) traffic in bytes/s

Fig. 12. Network Traffic

Scalability is important metrics for distributed systems. We tested different system sizes. As illustrated in Fig. 11, the correlation-aware prefetching always exhibits the best performance, regardless of the system size.

To prove that correlation-aware approach indeed decreases network traffic compared with traditional one, we traced real network traffic of a single OSD using *Sysstat* utilities. Fig. 12 shows the result. Since objects are distributed over OSDs evenly and prefetching and common requests are evenly distributed too in the sense of probability, we can conclude that correlation-aware approach decreases both the number of packets sent and the number of bytes sent effectively.

5 Related Work

Many literals about caching and prefetching have been published. Some of them focused on utilizing the existence of locality in disk access patterns, both spatial locality and temporal locality [6][7]. The modern operating system design concept attempts to prefetching consecutive blocks from disk to reduce the cost of on-demand I/Os [8][9]. However, when a file is not accessed sequentially or the whole data of a file are not stored consecutive, prefetching can probably result in extra I/Os. Thus numerous works have been done to find other prefetching strategies. By implementing history log and data mining technique, [10][11][12] detect access patterns which can be put to use in the future access. J.R.Cheng et al [13] has considered semantic links among objects to prefetching data in object-oriented DBMSs, while [1] tracking multiple per-object read-ahead contexts to performance prefetching.

Besides, existing studies have aimed at changing the prefetching extent dynamically and adaptively without manual intervention. A window covers all the objects has semantic link was proposed in [13]. Also, Linux kernel adopts read-ahead window to manage its associated prefetching [8].

Our approach differs from these works. In NBJLOFS, since a file may be scattered over many storage nodes, and the correlation among objects can be determined easily by the object attributes, so the correlation-aware prefetching with orthogonal layout is a natural solution. Our moving window mechanism is largely derived from these previous works.

6 Conclusions and Future Work

This paper proposed an innovative prefetching strategy for distributed object-based file system. The correlationship among objects was used to determine which objects should be prefetched. We designed an orthogonal replica layout, it reduces network traffic effectively compared with the scattered layout. We presented two distribution algorithms to guarantee this kind of layouts. We also designed two moving window strategies to adjust the size and the content of the prefetching window automatically. Compared with the traditional spatial based prefetching approach, our new approach decreases network traffic and may produce large sequential instead of small random disk operations. We admit, the introducing of the prefetching mechanism bring some negative impacts along with the dramatically benefits. The way object clustering only guarantees global load balance, while load imbalance will occur locally. However, the experiment results show that the benefits are far beyond the side effects. In the distributed object-based file system, our new prefetching approach exhibited much higher performance than the traditional spatial locality based approach.

This research can be extended in several directions. For example, in order to alleviate the workload of replica node, we are thinking about dividing duplication to several OSDs other than one single OSD. As a result, the migration strategy also needs to be redesigned. Moreover, by studying the access pattern, more works can be done to investigate other correlations among objects.

References

1. Tang, H., Gulbeden, A., Zhou, J., Strathearn, W., Yang, T., Chu, L.: The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage. In: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Pittsburgh, PA, USA (Nov 2004) 53–62
2. Wang, P., Gilligan, R.E., Green, H., Raubitschek, J.: IP SAN - From iSCSI to IP-Addressable Ethernet Disks. In: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies, San Diego, CA, USA (Apr 2003) 189–193
3. Lonczewski, F., Schreiber, S.: The FUSE-System:an Integrated User Interface Design Environment. In: Proceedings of Computer Aided Design of User Interfaces, Namur, Belgium (Jun 1996) 37–56
4. Olson, M.A., Bostic, K., Seltzer, M.: Berkeley DB. In: Proceedings of the annual conference on USENIX Annual Technical Conference, Monterey, California, USA (Jun 1999) 183–192
5. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (RAID). In: Proceedings of the 1988 ACM SIGMOD international conference on Management of data, Chicago, Illinois, United States (Jun 1988) 109–116
6. Liu, H., Hu, W.: A Comparison of Two Strategies of Dynamic Data Prefetching in Software DSM. In: Proceedings of the 15th International Parallel and Distributed Processing Symposium, IEEE Proceedings 15th International, San Francisco, CA, USA (Apr 2001) 62–67
7. Jiang, S., Ding, X., Chen, F., Tan, E., Zhang, X.: DULO: an Effective Buffer Cache Management Scheme to Exploit both Temporal and Spatial Locality. In: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, San Francisco, CA, USA (Dec 2005) 101–114
8. Butt, A.R., Gniady, C., Hu, Y.C.: The performance impact of kernel prefetching on buffer cache replacement algorithms. In: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Banff, Alberta, Canada (Jun 2005) 157–168
9. Pai, R., Pulavarty, B., Cao, M.: Linux 2.6 Performance Improvement through Readahead Optimization. In: Proceedings of the Linux Symposium, July 2004, Ottawa, Ontario, Canada (Jul 2004) 391–401

10. Soundararajan, G., Mihailescu, M., Amza, C.: Context-aware prefetching at the storage server. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference, Boston, Massachusetts (Jun 2008) 377–390
11. Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D., Zelenka, J.: Informed Prefetching and Caching. In: Proceedings of the fifteenth ACM symposium on Operating systems principles, Copper Mountain, Colorado, United States (Dec 1995) 79–95
12. Amer, A., Long, D.D.E., Burns, R.C.: Group-Based Management of Distributed File Caches. In: Proceedings of the 22 nd International Conference on Distributed Computing Systems, Vienna, Austria (Jul 2002) 525–534
13. Cheng, J.R., Hurson, A.R.: On The Performance Issues of Object-Based Buffering. In: Proceedings of the First International Conference on Parallel and Distributed Information Systems, Miami Beach, FL, USA (Dec 1991) 30–37