J Supercomput
https://doi.org/10.1007/s11227-018-2333-6

APS: adaptable prefetching scheme to different running environments for concurrent read streams in distributed file systems

Sangmin Lee[1,2] · **Soon J. Hyun**[1] · **Hong-Yeon Kim**[2] · **Young-Kyun Kim**[2]

Abstract Distributed file systems (DFSs) are widely used in various areas. One of the key issues is to provide high performance of concurrent read streams (i.e., multiple series of sequential reads by concurrent processes) for their applications. Despite the many studies on local file systems (LFSs), research has seldom been done on concurrent read streams in DFSs with different running environments (i.e., different types of storage devices and various network delays). Furthermore, most of the existing DFSs have a sharply degraded performance compared with a LFS (i.e., EXT4). Therefore, to achieve high performance in concurrent read streams, this study introduces a populating effect that keeps sending subsequent reads to a storage server and then proposes an adaptable prefetching scheme (APS) to obtain the effect even in different running environments. Hence, our APS resolves all the problems that we identified as dramatically degrading the performance in existing DFSs. In three different types of storage devices and in various network delays, the evaluation results show that our prefetching scheme (1) achieves almost the same performance as a LFS from an individual server and (2) minimizes the performance degradation of random reads.

✉ Sangmin Lee
sangmin2@kaist.ac.kr; sangmin2@etri.re.kr

Soon J. Hyun
sjhyun@kaist.ac.kr

Hong-Yeon Kim
kimhy@etri.re.kr

Young-Kyun Kim
kimyoung@etri.re.kr

1 Department of School of Computing, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea

2 High Performance Computing Research Group, Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea

Published online: 23 March 2018

Keywords Distributed file system · Concurrent read streams · Data prefetching · Device type · Network delay

1 Introduction

Nowadays, distributed file systems (DFSs) have key roles in various areas. In the cloud computing area, reliable DFSs (e.g., Amazon S3 and Gluster) are used as the infrastructure for cloud storage [12,22]. In big data and social networking areas, large-scale DFSs (e.g., GFS (Google File System) and HDFS (Hadoop Distributed File System)) are also adopted to support batch processing [10,31]. Additionally, parallel file systems (e.g., Lustre and PanFS) are widely used in large-scale cluster computing for supercomputing such as bio-information computing and environment modeling [8,37].

Depending on the applications, DFSs have the following running environments. First, DFSs have a wide range of scales from a single storage server to tens of thousands of storage servers [7,8,31,36,37]. Second, according to their scales, DFSs have differently scaled network fabrics to interconnect all of the clients and the storage servers. Thus, network routes between clients and storage servers have different numbers of hops, which cause different network delays due to the per-hop delay [23,24]. Furthermore, a client could be located far from a storage server for a certain purpose (e.g., data migration and data backup), which causes a very long network delay between them [19,35]. Finally, DFSs could have different types of storage devices at a storage server to satisfy their performance requirements, such as a conventional hard disk for large capacity and a striped RAID (Redundant Array of Independent Disks) (e.g., RAID-0 and RAID-5) or a SSD (Solid-State Drive) for high performance [2,13,33].

On the other hand, a key issue of local file systems (LFSs) and DFSs is the performances of a single read stream (i.e., a series of sequential reads by a process) and concurrent read streams (i.e., multiple series of sequential reads by concurrent processes) [2,5,9,11,15,18,21,30,38]. Especially in DFSs, the performance of concurrent read streams is much more important than that of a single one because concurrent read streams are frequently issued to an individual storage server by multiple clients that are used to serve cloud data, analyze big data, and calculate scientific data [4,27–29].

To enhance the performance of concurrent read streams, LFSs are provided with a CFQ (Completely Fair Queuing) I/O scheduler, which minimizes the number of seeks to the device for the concurrent read streams. Additionally, LFSs perform prefetching, known as readahead in Linux, by invoking the current request and its subsequent one [30]. The prefetching in a LFS (i.e., EXT4) has different policies based on the device type such as a single disk and a striped RAID [9].

However, despite the use of those performance-enhancing features of LFSs, existing DFSs have seriously degraded performances for concurrent read streams and random reads in all running environments due to the following reasons. First, the existing DFSs suffer from degraded performance in concurrent read streams because the typical

processing structure of their storage servers (i.e., a single request queue shared by concurrent I/O workers) incurs a large number of seeks to a device at a server on CFQ.

Second, for each individual read stream, existing DFSs fail to obtain the same performance as a LFS (i.e., EXT4) from an individual storage server in different running environments (i.e., different types of devices and lengths of network delays) due to their fixed prefetching method. In other words, even if they conduct prefetching on the client and server sides, their prefetching policies are fixed without considering the type of storage device and the delay length of a network, which cause serious performance degradation. To meet their performance requirements, however, it is important to (1) provide the performance that they expect for a certain type of storage device and (2) maintain it in different network delays.

Finally, to achieve a high performance and maintain it, most of the existing DFSs (e.g., Lustre and HDFS) sacrifice the performance of random reads. For example, simply by setting the prefetching size to a large value at a client, Lustre might achieve good performance on different types of storage devices and in long network delays. However, this simple policy leads to the degraded performance of random reads.

Therefore, the aim of this paper was for our proposed adaptable prefetching scheme (APS) to achieve the same performance as a LFS (EXT4) from an individual storage server (1) on three types of storage devices and (2) in various network delays between a client and a storage server.

To this end, this study makes the following contributions to a DFS:

- For concurrent read streams, we define the problem that seriously degrades the performance and then resolve it by dedicating an individual stream to a specific I/O worker at a storage server.
- For each individual read stream, we describe why existing DFSs fail to obtain the expected performance in different running environments by introducing a populating effect which keeps sending subsequent reads to a device at a storage server without halting the device. Then, we use our adaptable prefetching mechanism (i.e., APS), which is aware of the type of storage device and the delay length of a network to obtain the populating effect.

Through performance evaluations on three types of storage devices and in various network delays, we show that (1) existing DFSs have a highly degraded performance, and (2) our APS achieves almost the same performance as a LFS from an individual server for all concurrent stream cases with as little performance degradation of the random reads as possible. We also show that (3) our APS achieves a much higher performance for realistic workloads (i.e., the Web-server workload of the FileBench) than those of existing DFSs.

The remainder of this paper is organized as follows. We begin with previous studies on prefetching for DFSs and LFSs in Sect. 2. In Sect. 3, we introduce a populating effect and then define the problems that degrade the performance of concurrent read streams in a DFS. We propose our adaptive prefetching scheme (APS) for different running environments in Sect. 4. Section 5 describes the performance of random reads in a DFS. In Sect. 6, we suggest the use of dedicated workers and then present our design of the APS. Section 7 shows the performance evaluation results of our APS

and existing DFSs in different running environments using two micro-benchmarks (IOzone and FileBench). We conclude the paper in Sect. 8.

2 Related work

2.1 Prefetching in existing DFSs

As an open-source storage, Gluster [12] is one of the popular DFSs for cloud storage. Instead of the prefetching facility of VFS (Virtual File System), it uses a readahead translator which conducts prefetching based on its own decision on whether to issue readaheads.

Lustre [8] like PanFS [37] is a prevalent parallel file system for use in large-scale cluster computing. To improve the performance of read streams, it enables the client to conduct very large-sized readaheads (i.e., 40 MB). Additionally, for object management, the storage server (i.e., OSS (Object Storage Server)) uses ldiskfs (Lustre Disk File System) as an underlying LFS instead of general LFSs (e.g., EXT4 and XFS) to read a large data block (i.e., 1 MB block) at one time. Different from other existing DFSs, it does not issue readaheads at an OSS to minimize useless reads [8].

As an alternative open-source solution to GFS [10], HDFS [31] is a large-scale DFS optimized for a batch processing system (i.e., MapReduce). It provides consistent performance for read streams even in a very large-scale network environment. However, it has poor performance in random reads. To overcome this limitation, many efforts have been made [31]. Additionally, some works have been done to obtain enhanced performance from fast storage devices (i.e., SSD and NVM (Nonvolatile Memory)) [13,14,33]. However, they used fixed prefetching methods without taking into consideration the different types of devices. On the other hand, unlike the other existing DFSs, HDFS has the following two characteristics. First, it provides users with its own APIs instead of a POSIX (Portable Operating System Interface) [20]. Thus, it does not issue readaheads by a VFS at a client but instead conducts prefetching aggressively at a storage server. Second, HDFS splits the whole data of a file into 64 MB chunks and spreads them across storage servers.

NFS (Network File System) is the protocol of a prevalent file system to access data that are stored on networked storages. For the high throughput of sequential reads, it has a fixed large readahead size (e.g., 7.5 MB) at the client and server sides. To enhance the throughput, previous works have discussed out-of-order readaheads for which sequential read requests for a read stream were processed in a reverse order at a storage server [7,25,26]. Ellard et al. [7] proposed a SlowDown algorithm which considered the read requests whose positions were within a series of pages at a configured boundary as the same read stream at a storage server. Recently, Rago et al. [26] tried to serialize readahead requests at a storage server with the sequence number issued by a client. On the other hand, Chen et al. [3] merged multiple I/O requests into a single vectorized one for concurrent read streams. However, all the above works were unable to cope with different types of storage devices and with various network delays due to their fixed prefetching strategies.

Consequently, all the existing DFSs conduct prefetching in a fixed manner. This fact implies that they suffer from performance degradation in different running environments.

2.2 Research on prefetching

2.2.1 Prefetching in LFS

Many efforts have been made in LFSs to improve the prefetching performance [2,5,9,11,15,18,21,30,38]. Most of the works tried to optimize the prefetching framework or improve the performance of either one read stream or concurrent ones only for a single type of storage device (i.e., hard disk). On the other hand, EXT4 (a default LFS in Linux) [9] and SASEQP [2] conducted prefetching with different policies based on the geometric information of a storage device. For instance, EXT4 set the readahead size to 128 KB for a hard disk and 2 * (the number of striped disks) * (strip size) for a striped RAID. Additionally, AMP [11] used an adaptable prefetching for multiple read streams, but its prefetching was adapted to not different types of storage devices but to different types of workloads.

2.2.2 Prefetching in DFS

Most of the studies on prefetching in DFSs mainly focused on not the performance of the prefetching itself but on the efficiency or utilization of resources (e.g., cache and disk) with a preloading set consisting of two methods: sequential detection and aggressive prefetching (i.e., selecting the prefetching size) [6,16,17,32,39–41]. With a single preloading set, STEP [17] and PFC [41] detected sequentiality in block requests and promoted aggressive prefetching at a server to preload data into the large memory cache of a client. To cope with different kinds of application workloads, Soundararajan et al. [32] proposed a context-aware prefetching with multiple preloading sets. Unlike the above approaches for general applications, Lee et al. [16] suggested a new preloading set optimized for Web cluster systems.

Unlike the above studies, Martin et al. [19] reported the performance evaluation according to the network delay. They found that their target system had great insensitivity to long network delays, which means that the high performance for a short delay is maintained even in long network delays. However, they did not discuss the reason.

Like the existing DFSs, the previous prefetching studies for DFSs have fixed policies without being aware of the type of storage device as well as the delay length of a given network. Their fixed policies could cause highly degraded performance in concurrent read streams for different running environments.

3 Performance degradation in concurrent read streams

This section describes a populating effect for a DFS for maximized throughput and then defines the problems that degrade the performance of concurrent read streams.

3.1 Introduction to populating networked reads (PNR)

For high performance in each individual read stream, this subsection introduces an effect that keeps on populating read requests from a client to a storage server and then describes the condition to obtain the effect according to two cases: (1) no prefetching and (2) prefetching at a client.

3.1.1 Simple read stream

To analyze the performance of an individual read stream in a DFS, we did the following. First, our experiment was conducted with a running environment in which a client and a storage server with a single 6 TB hard disk were interconnected through a 10-Gigabit switch. Second, we used a client and a server program to simplify the DFS. The client program repeats the following steps until receiving an EOF (End Of File) response from the server: (1) sends a sequential read request, (2) waits for a reply to the data request, and (3) then receives the replied data. On the other hand, the server program executes the following steps: (1) waits for a read request from a client, (2) receives the request, (3) reads the data for the request from a disk, and (4) sends the data to the client. Third, we measured the performance according to (1) the request sizes from 4 KB to 4 MB at a client and (2) the two readahead sizes, 128 and 256 KB, at a storage server. Finally, we analyzed the measurement result as the utilization of a disk at a storage server. Disk utilization indicates how much time the server spends serving I/Os to a disk. For instance, if it has 100% disk utilization, it implies that the server dispatches and serves I/Os without any halts. The reason why we selected disk utilization instead of throughput is that performance measurements vary in throughput according to the kind of disk controllers (e.g., SATA-II and SATA-III) even for the same disk.

As shown in Fig. 1 where RA denotes a readahead, the result shows the following. In a LFS (i.e., EXT4), disk utilizations are all the same regardless of the request size because the prefetching mechanism of the VFS splits a large read request into readaheads or merges small ones into a readahead. In a DFS, however, for a 128 KB RA, the disk utilizations for request sizes from 32 to 256 KB are equal to those in EXT4, whereas those for the other request sizes are lower than those in EXT4. Similarly, for a 256 KB RA, while the disk utilizations for request sizes from 32 to 512 KB are the same with EXT4, those for the others are lower than those in EXT4. This experimental result indicates that the disk utilization of a DFS depends on the request size of a client even if a storage server runs on top of EXT4.

3.1.2 Populating read requests

Even if it takes time for a DFS to send a request and receive its reply through a network, some measurements (e.g., 32, 64, 128, and 256 KB for a 128 KB RA) in the previous result are the same as those with a LFS. To explain this, Fig. 2 shows what happens at a storage server. When the server receives a read request for data i from a client (N1.i), it reads data i (D.i) using its LFS and then sends the data i to the client (N2.i). At the same time, the LFS invokes a readahead for data i+1 (D.i+1) through the VFS in an asynchronous way. Only when the server receives a read request for data i+1 (N1.i+1)

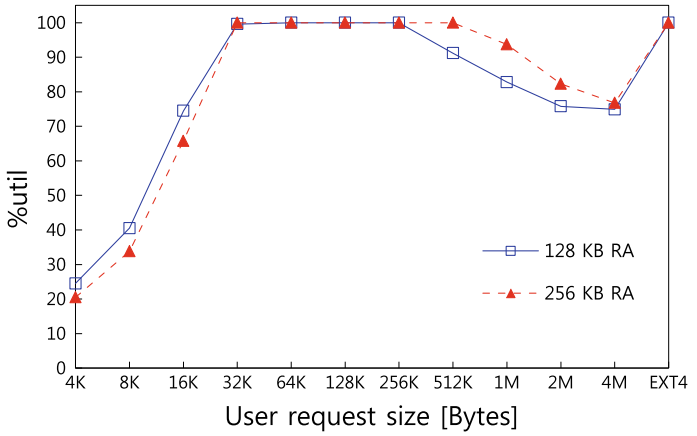

Fig. 1 Disk utilization in a simplified DFS

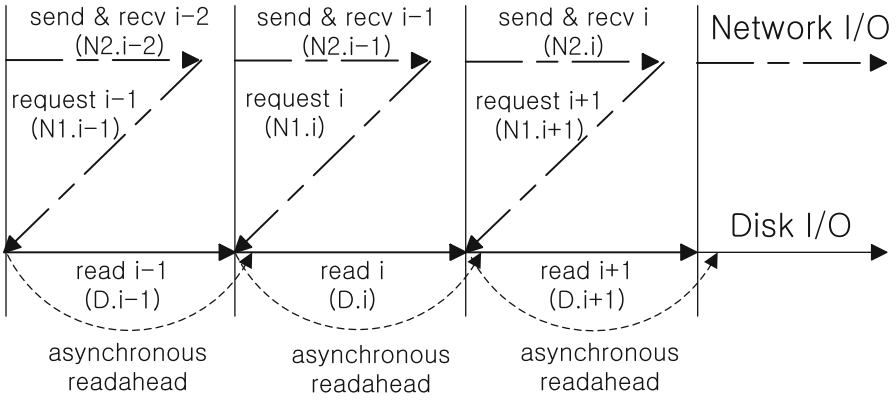

Fig. 2 Populating Networked Reads in a DFS

before the completion of an asynchronous readahead for data i+1 (D.i+1), it keeps issuing subsequent I/Os to a storage device without halting the device. Therefore, a DFS can achieve the same performance as a LFS in an individual read stream. We name this kind of populated effect in a DFS as Populating Networked Reads (PNR).

To specify the PNR, we define the variables, operations, and functions, shown in Tables 1, 2, and 3, respectively. Additionally, we define the SendAndReq (i, sz, j) operation. This operation consists of the following serial sub-operations: (1) a storage server sends the ith sz-sized data to a client via a network, and the client receives the ith data; (2) the client sends a read request for the jth sz-sized data to the server, and the server receives the request.

$$\text{SendAndReq}(i, sz, j) = \text{Send}(\text{Len}(\text{the } i\text{th data}))_{s \to c} + \text{Send}(\text{Len}(smsg_{j,sz}))_{c \to s} \tag{1}$$

where $i < j$

Table 1 Variable definitions for a read stream

Variable	Description
ra_s	The readahead size of a device at a storage server (s)
ra_c	The readahead size of a client (c)
RA_c	The large readahead size of c
$rqst_c$	The explicit request size of c
RB_{dev}	The average read performance (Bytes/second) of a storage device dev
p_{dev}	A ratio of the peak read performance of dev to RB_{dev}
P	The number of striped disks
S	The strip size of a striped RAID
i	The ith read among fixed-length sequential ones for a file
$smsg_{i,sz}$	A request message for the ith sz-sized sequential read

Table 2 Operation definitions

Operation	Description
Send(sz)	A transmission of sz bytes
ReqRecv(req)	c sends a request req to s and receives a reply for the request
Read(pos, sz)	s reads sz bytes from the starting position pos
Proc(req)	s processes a request req

Table 3 Function definitions

Function	Description
Len(msg)	The length of a message msg
T[op]	The execution time to conduct an operation op

3.1.3 PNR condition for a DFS without prefetching at a client

In a DFS without prefetching at a client (e.g., HDFS and GFS), the client only issues an explicit read request for the current request like our simplified DFS. Thus, to obtain the PNR effect, before a storage server finishes performing a readahead (i.e., Read($(i + 1) * ra_s, ra_s$)), it should receive the read request for the readahead from a client (i.e., Send(Len($smsg_{i+1,rqst_c}$))$_{c \to s}$).

If ra_s is an integer multiple of $rqst_c$ or $rqst_c$ is an integer multiple of ra_s, the condition to obtain the PNR effect is divided into the following two cases based on the size of $rqst_c$:

If $rqst_c < ra_s$, then

$$\sum_{j=0}^{n-1} \mathrm{T}[\mathrm{SendAndReq}(i+j, rqst_c, i+j+1)] \leq \mathrm{T}[\mathrm{Read}((i+1) * ra_s, ra_s)] \quad (2)$$

where $n = ra_s / rqst_c$

Otherwise,

$$\mathrm{T}[\mathrm{SendAndReq}(i, rqst_c, i+1)] \leq \mathrm{T}[\mathrm{Read}((i+1) * rqst_c, ra_s)] \quad (3)$$

According to the above-defined conditions, the PNR condition mainly depends on the request size of a client because the reading time of the right side is almost fixed. This fact explains why the disk utilization is different depending on the request size in our simplified DFS.

3.1.4 PNR condition for a DFS with prefetching at a client

To support POSIX for legacy applications, the client of a DFS should run on top of the VFS as the clients of most existing DFSs (e.g., Ceph, Lustre, and PanFS) do. Thus, the client issues not an explicit read request but a readahead because the prefetching mechanism of the VFS splits a large read request into readaheads or merges small ones into a readahead. Additionally, it issues an asynchronous readahead by its VFS as a storage server does. That is, as shown in Fig. 3, both a client and a storage server perform readaheads simultaneously with their own readahead contexts, which consist of (1) start_off (the starting position of a readahead window) and (2) async_sz (the current readahead size).

In the DFS with prefetching at a client, because (1) the request size is ra_c and (2) the number of read requests is 2, the duration of the right side of the PNR condition becomes lengthened compared with that of the previous DFS. Therefore, the PNR condition can be divided into two cases according to ra_c.

If $ra_c < ra_s$, then

$$\sum_{j=0}^{n-1} \mathrm{T}[\mathrm{SendAndReq}(i+j, ra_c, i+j+2)] \leq \mathrm{T}[\mathrm{Read}((i+1) * ra_s, ra_s)] + \mathrm{T}[\mathrm{Read}((i+2) * ra_s, ra_s)] \quad (4)$$

where $n = ra_s / ra_c$

Otherwise,

$$\mathrm{T}[\mathrm{SendAndReq}(i, ra_c, i+2)] \leq \sum_{j=0}^{n-1} \mathrm{T}[\mathrm{Read}((i+1) * ra_c + j * ra_s, ra_s)] + \mathrm{T}[\mathrm{Read}((i+1) * ra_c, ra_s)] \quad (5)$$

where $n = ra_c / ra_s$

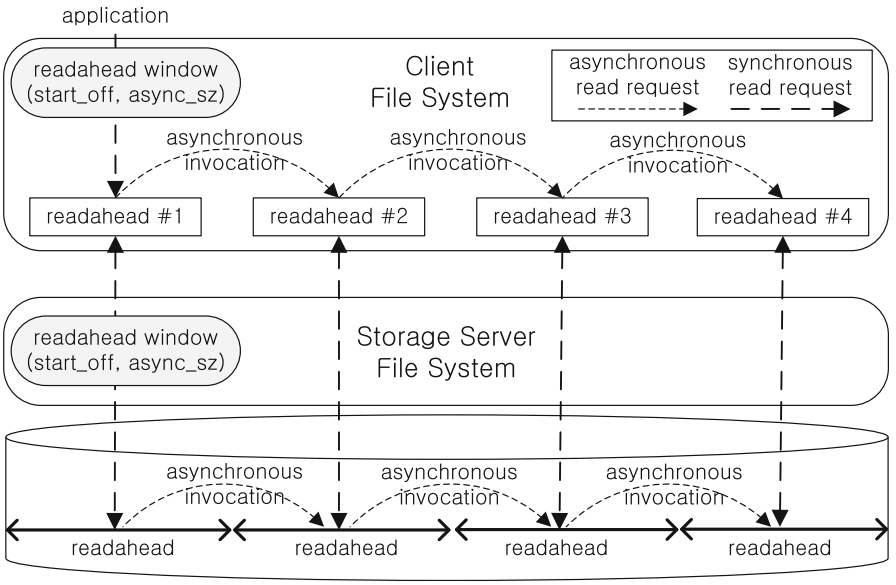

Fig. 3 Simultaneous prefetching at a client and a storage server

Therefore, when compared with a DFS without prefetching at a client, it is easier with this kind of DFS to satisfy the PNR condition because of the asynchronous readahead at the client (i.e., Read$((i+1)*ra_s, ra_s)$ for $ra_c < ra_s$ and $\sum_{j=0}^{n-1}$ Read$((i+1)*ra_c + j*ra_s, ra_s)$ for $ra_c \geq ra_s$).

3.2 The problems of concurrent read streams

This subsection defines the problems that degrade the performance of concurrent read streams in DFSs with different running environments.

3.2.1 Mixed read streams in concurrent read streams

With the help of the CFQ I/O scheduler (i.e., the default I/O scheduler in Linux), a LFS can achieve high performance in concurrent read streams. The CFQ was originally designed to provide fair allocation of a storage device to all processes (or threads) that need to request I/Os to the device. To this end, the CFQ allocates a time slice to each process (or thread) in a round-robin way. During the time slice, only one process can dispatch I/Os to the device. For concurrent read streams in a LFS, it allows only one read stream to issue its sequential requests to the device without mixing the others, thus minimizing the number of randomized seeks to the device.

In most DFSs, on the other hand, their storage server typically is composed of (1) one request queue and (2) concurrent I/O workers shown in Fig. 4. With this typical structure, a storage server processes requests in the following way. First, the storage server receives requests from clients and then enqueues the requests in a shared request

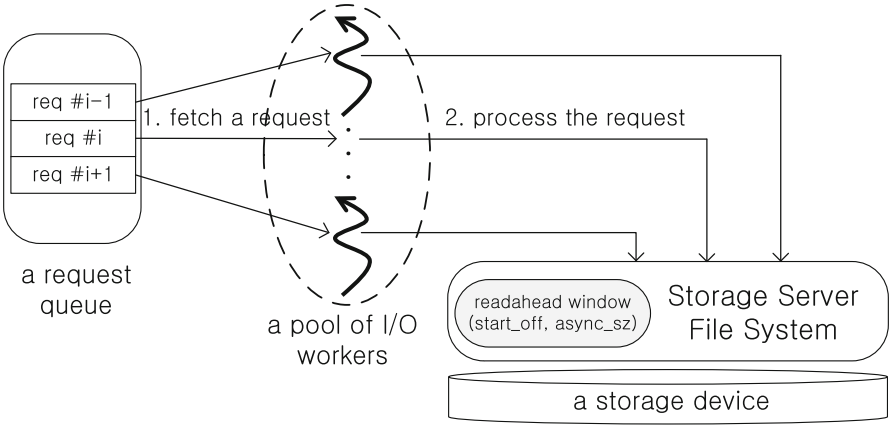

Fig. 4 The typical structure of a storage server in conventional DFSs

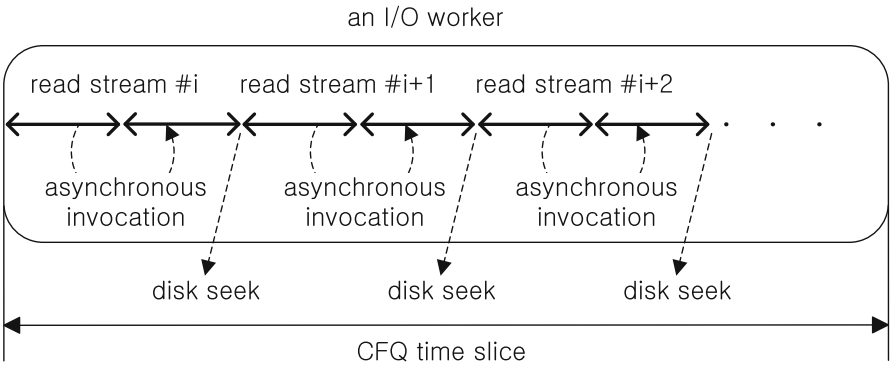

Fig. 5 Mixed read streams on a CFQ I/O scheduler at a storage server

queue to temporarily keep them. Second, if the queue is not empty, all its I/O workers fetch requests from the queue and then process them concurrently through the VFS (i.e., the file system of a storage server).

Different from a LFS, in most DFSs with such a typical processing structure, their performances for concurrent read streams are degraded dramatically because the behavior of the CFQ at a storage server is not taken into consideration. More specifically, when concurrent read streams are issued from a client (or clients), each I/O worker at a storage server processes a read request for a read stream that is chosen arbitrarily among the concurrent ones. Therefore, an I/O worker issues mixed read requests for different read streams to a storage device during its time slice shown in Fig. 5.

3.2.2 Failure to meet the PNR condition in each individual read stream

To achieve the same performance as a LFS for the concurrent read streams, a DFS should satisfy the condition for the PNR in each individual read stream. For this, in

the basic PNR conditions of the previous subsection, the duration of the right side for device reads should be equal to or greater than that of the left side for network transfers. However, the basic conditions could be satisfied only by some DFSs that run with the following simple environment. First, storage servers have conventional hard disks. Second, all the clients and storage servers in a DFS are directly interconnected by a single switch (i.e., a short network delay).

As mentioned before, however, DFSs have the following different running environments. First, for high-performance demand, a storage server can be equipped with a striped RAID or a fast device instead of a hard disk. Thus, the reading time from a high-performance device becomes shorter, which results in failure to meet the PNR condition in each individual read stream.

Second, for storing and processing large-scale data, many storage servers and clients are interconnected by a long-delayed network. Furthermore, for data migration and data backup, a client could be located far from the storage servers. Thus, the network delay from a client to a storage server becomes long, which also results in failure to satisfy the PNR condition. To resolve this problem, a DFS should be insensitive to long network delays, which means that its high performance for a short delay is not degraded despite the increasing network delay.

To be insensitive to long network delays, large-scale DFSs adopted their own simple methods by lengthening the right side of the PNR condition such as the reading time from a storage device. For instance, Lustre sets the readahead size of a client to a very large value (i.e., RA_c), and HDFS performs aggressive prefetching at a storage server without waiting for read requests to arrive at the server. However, their insensitive methods are fixed for different lengths of delays. Thus, in a certain network environment with a short delay, their fixed methods just cause performance degradation in the random reads.

Therefore, to resolve the above two problems, a DFS should perform adaptable prefetching for the type of storage device and the delay time of a given network, thus obtaining the same performance as a LFS in concurrent read streams without degradation in the random read performance.

4 APS for device type and network delay

In this section, we propose our adaptable prefetching scheme for each individual read stream and describe how to apply it to different running environments.

4.1 APS for a relaxing PNR condition

By using the PNR conditions specified in Sect. 3.1, we propose an adaptable prefetching scheme (APS) for different running environments.

In a DFS without prefetching at a client, if $rqst_c$ is equal to ra_s, the PNR condition is the following:

$$\mathrm{T}[\mathrm{SendAndReq}(i, ra_s, i+1)] \leq \mathrm{T}[\mathrm{Read}((i+1) * ra_s, ra_s)] \tag{6}$$

Similarly, in a DFS with prefetching at a client, if ra_c is set to ra_s, the PNR condition is the following:

$$\begin{aligned} \mathrm{T}[\mathrm{SendAndReq}(i, ra_s, i+2)] \leq\ & \mathrm{T}[\mathrm{Read}((i+1) * ra_s, ra_s)] \\ & + \mathrm{T}[\mathrm{Read}((i+2) * ra_s, ra_s) \end{aligned} \tag{7}$$

Because $\mathrm{T}[\mathrm{Read}((i+1) * ra_s, ra_s)] \approx \mathrm{T}[\mathrm{Read}((i+2) * ra_s, ra_s)]$, Condition (7) is converted into the following:

$$\frac{\mathrm{T}[\mathrm{SendAndReq}(i, ra_s, i+2)]}{2} \leq \mathrm{T}[\mathrm{Read}((i+1) * ra_s, ra_s)] \tag{8}$$

Thus, Condition (8) becomes relaxed by half compared with that of a DFS without prefetching at a client (Condition (6)) because of an asynchronous readahead of a client. By taking advantage of this relaxing method, we can define a dynamically relaxed PNR condition by setting the total number of asynchronous readaheads as the variable α, denoted as an APS.

Therefore, the PNR condition of the APS is as follows:

$$\frac{\mathrm{T}[\mathrm{SendAndReq}(i, ra_s, i+\alpha+1)]}{1+\alpha} \leq \mathrm{T}[\mathrm{Read}((i+1) * ra_s, ra_s)] \tag{9}$$

Therefore, our APS can relieve the PNR condition simply by incrementing the number of subsequent readaheads on the client side. Our prefetching scheme is quite different from the existing prefetching facility that simply extends the readahead size (i.e., the prefetching size). By setting α to an appropriate value, our prefetching scheme can cope with different types of storage devices (e.g., hard disk, striped RAID, and SSD) and different lengths of network delays.

4.2 Device type and network delay

4.2.1 Fast device and striped RAID

Only with two single readaheads at a client and a storage server, it is hard to meet the PNR condition for a fast device and a striped RAID. Therefore, we propose new prefetching policies using the APS for these types of devices.

On a fast device (e.g., a SSD), in the PNR condition, the reading time from the device becomes shorter than the transferring time through a network, which causes frequent failures in achieving the PNR effect. This limitation can be overcome by adopting our APS. To take advantage of the APS, an appropriate α value should be selected. To this end, first, (1) $\mathrm{T}[\mathrm{Read}((i+1) * ra_s, ra_s)]$ is estimated by $\frac{ra_s}{RB_{dev} * (1+p_{dev})}$. Thus, Condition (9) is changed to the following:

$$\frac{\mathrm{T}[\mathrm{SendAndReq}(i, ra_s, i+\alpha+1)]}{\frac{ra_s}{RB_{dev} * (1+p_{dev})}} - 1 \leq \alpha \tag{10}$$

where $\alpha \geq 1$, $p_{dev} \geq 1$

Second, (2) T[SendAndReq($i, ra_s, i + \alpha + 1$)] is also estimated by the execution time to conduct an individual operation, which will be discussed in Sect. 6.2. Finally, with the above estimated durations, an appropriate α value for the APS is extracted by selecting an integer value that is equal to or minimally larger than the left side's value of Condition (10).

On a striped RAID with a wide bandwidth to combine striped disks, a LFS parallelizes read requests by a large-sized readahead ($2 * P * S$), which makes all individual disks busy in dispatching and serving I/Os. If ra_c is equal to ra_s, the PNR condition for a striped RAID is the following:

$$\frac{\text{T[SendAndReq}(i, 2 * P * S, i + 2)]}{2} \leq \text{T[Read}((i + 1) * 2 * P * S, 2 * P * S)] \tag{11}$$

To analyze the above PNR condition for a whole striped RAID, an additional PNR condition for the jth individual disk needs to be defined as follows:

$$\frac{\text{T[SendAndReq}(i, 2 * P * S, i + 2)]}{2} \leq \text{T[Read}((i + 1) * 2 * S, 2 * S)]_j \tag{12}$$

where $0 \leq j < P$

In the above condition, the duration of the left side becomes longer with respect to the increasing number of striped disks. Thus, it is hard for the existing prefetching method (i.e., expanding the prefetching size of a client to $2 * P * S$) to meet Condition (12). To overcome this limitation, we consider a striped RAID as a single fast device with the readahead size of S if S is equal to or greater than the normal ra_s (i.e., 128 KB). Then, as on a fast device, we make use of an APS whose α value is extracted with the following condition:

$$\frac{\text{T[SendAndReq}(i, S, i + \alpha + 1)]}{\frac{S}{RB_{raid} * (1 + p_{raid})}} - 1 \leq \alpha \tag{13}$$

where $S \geq ra_s$

4.2.2 Different network delays

Different network environments of DFSs cause different lengths of delays from a client to a storage server. In Condition (6) and (8), a long network delay increases the duration of the left side, whereas a short one decreases it. Thus, just like for different device types, an APS is also used by extracting an appropriate α value with the below condition.

$$\frac{\text{T[SendAndReq}(i, ra_s, i + \alpha + 1)]}{\text{T[Read}((i + 1) * ra_s, ra_s)]} - 1 \leq \alpha \tag{14}$$

Therefore, the APS can obtain the same performance as a LFS for each individual read stream in different network delays through delay-aware prefetching.

Table 4 Variable definitions for a random read

Variable	Description
pos	The starting position of a random read
sz	The request size of a random read
m	sz/ra_c
$rmsg_{p,s}$	A request message for the s-sized read from the starting position p

5 The performance of random reads in a DFS

As mentioned in Sect. 1, many existing DFSs (e.g., Lustre and HDFS) sacrifice the performance of random reads to obtain a high performance for a read stream from different running environments. Thus, this section describes why they suffer from degraded performance in the random reads.

In a DFS, the performance of random reads depends on how few useless reads are issued to a storage device at a server. Most DFSs consider a random read as part of the sequential reads. Thus, they process a random read through their prefetching facilities. Due to these prefetching facilities, they might issue useless read requests to a storage server for the random read. Therefore, as the number of useless reads increases, the random performance becomes more degraded.

In addition to the number of useless reads, the performance of random reads is also dependent on how fast it reaches that of an individual read stream. In a random read whose request size is larger than the readahead size, a DFS performs a series of sequential reads from a starting position to an ending one. Therefore, for the high performance of random reads, a DFS also requires the high performance of each individual read stream, such as the achievement of the PNR effect.

To specify the execution time to conduct a random read in a DFS, we define the variables, as given in Table 4. To simply formulate it, we assume the following things. First, ra_c is equal to ra_s. Second, a random read satisfies the PNR condition if its request size is greater than ra_c. Finally, the request size of a random read is an integer multiple of ra_c if it is greater than ra_c.

The execution time of a random read consisting of pos and sz is divided into two cases according to sz as follows:

If $sz > ra_c$, then

$$\begin{aligned} \mathrm{T}[\mathrm{Send}(\mathrm{Len}(rmsg_{pos,ra_c}))_{c\to s}] + \sum_{j=0}^{m-1} \mathrm{T}[\mathrm{Read}(pos + j * ra_c, ra_c)] \\ + \sum_{j=0}^{1} \mathrm{T}[\mathrm{Read}(pos + (m+j) * ra_c, ra_c)] \end{aligned} \quad (15)$$

where $2 * \mathrm{T}[\mathrm{Read}(pos, ra_s)] \geq \mathrm{T}[\mathrm{SendAndReq}(i, ra_c, i+2)]$

Therefore, the cost of useless reads in this case is given as follow:

$$\sum_{j=0}^{1} \mathrm{T}[\mathrm{Read}(pos + (m + j) * ra_c, ra_c)] \tag{16}$$

Otherwise,

$$\mathrm{T}[\mathrm{Send}(\mathrm{Len}(rmsg_{pos,sz}))_{c \to s} + \mathrm{Read}(pos, sz) + \mathrm{Send}(sz)_{s \to c}] \tag{17}$$

Different from the case of $sz > ra_c$, this case ($sz \leq ra_c$) has no useless reads because there are no prefetched data on the client and server sides. For instance, if a client sets the readahead size to RA_c and a storage server does not conduct prefetching, a DFS can prevent useless reads for all the random reads whose request sizes are less than RA_c. However, the DFS incurs an additional transfer cost such as $\mathrm{Send}(sz)_{s \to c}$ instead of useless reads because it cannot obtain the PNR effect due to a large RA_c. Because the useless read cost of ra_c (i.e., $2 * \mathrm{Read}(pos, ra_c)$) is fixed, the transfer cost of RA_c is higher than the useless read cost of ra_c for all the request sizes that meet the following condition:

$$\mathrm{T}[\mathrm{Send}(sz)_{s \to c}] > 2 * \mathrm{T}[\mathrm{Read}(pos, ra_c)] \tag{18}$$

where $ra_c < sz \leq RA_c$

Therefore, the RA_c policy (e.g., Lustre) fails to achieve the high performance of random reads at request sizes smaller than RA_c because there is no PNR effect.

On the other hand, the useless read cost of the APS can be calculated simply by extending useless read cost (16) as follows:

$$\sum_{j=0}^{\alpha} \mathrm{T}[\mathrm{Read}(pos + (m + j) * ra_c, ra_c)] \tag{19}$$

where $sz > ra_c$

According to the above useless read costs, the performance of the random reads depends on the readahead size or the number of asynchronous readaheads. In other words, as the readahead size becomes larger or the number of readaheads increases, the useless read cost of a random read becomes higher, thus having a bad effect on the performance of the random reads. Therefore, for the high throughput of random reads, it is the best policy to select the smallest readahead size or the smallest number of readaheads if this selection can meet the PNR condition.

Theorem 1 *In a DFS, for the performance of each individual read stream to become insensitive to long network delays, it must have an inverse ratio to the performance of random reads.*

Proof To be insensitive to long network delays, a DFS should increase the reading time of the PNR condition. There could be two ways to increase it. The first way is

to set the readahead size to RA_c (e.g., Lustre and NFS). In the first way, the useless read cost for $sz > RA_c$ is $\sum_{j=0}^{1} \text{Read}(pos + (m + j) * RA_c, RA_c)$, and the transfer cost of a random read for $sz \leq RA_c$ increases in proportion to the size of RA_c. The second way is to increase the number of $\text{Read}(pos + (m + j) * ra_c, ra_c)$ at a storage server (e.g., HDFS and APS). In the second way, because the cost of useless reads is $\sum_{j=0}^{1+\beta} \text{Read}(pos + (m + j) * ra_c, ra_c)$ $(\beta = \alpha - 1 \geq 1)$, the cost of useless reads increases in proportion to the value of β. Therefore, less sensitivity to long network delays has an inverse relationship with better performance for random reads.

6 Design and implementation of the APS

To resolve the two problems defined in Sect. 3.2, this section proposes dedicated I/O workers for the first problem (i.e., mixed read streams) and then presents our final APS design using the dedicated workers for the second problem (i.e., failure to meet the PNR condition).

6.1 Dedicated I/O worker for each individual stream

As shown in Fig. 6 where rs_id denotes a stream identifier, dedicated I/O workers whose total number is fixed are devised to dedicate an individual read stream to a specific worker at a storage server to resolve the problem of mixed read streams. To this end, unlike a shared request queue in existing DFSs, each dedicated worker has its own queue, in which all requests for a certain read stream are enqueued by a dispatcher, and processes all the requests of its queue without being chosen arbitrarily for each request. Thus, dedicated workers prevent one read stream from being mixed with the others and also resolve the out-of-order readaheads mentioned in Sect. 2.1.

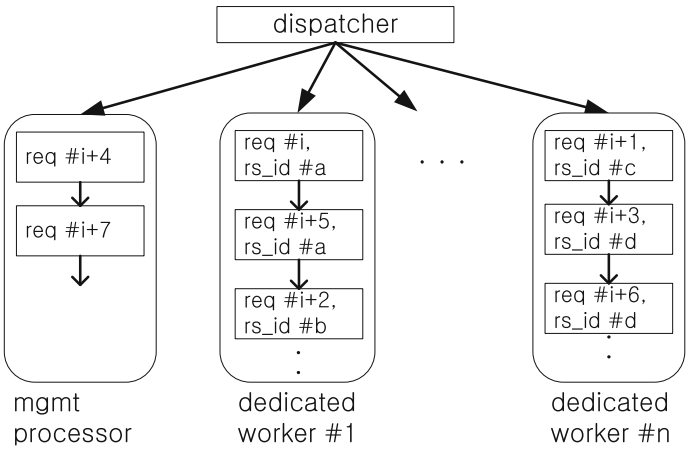

Fig. 6 An individual request queue in dedicated I/O workers

Separate from the dedicated I/O workers for I/O requests, an independent worker such as a mgmt processor is required to process the requests of the resource management (e.g., creating and deleting a file or stream) quickly, such as not waiting for earlier I/O requests to finish. To this end, it blocks all I/O workers using a read–write lock which permits the shared accesses of multiple readers and the exclusive access of a single writer. Hence, each dedicated I/O worker tries to lock the read–write as a reader, whereas a mgmt processor tries to lock it as a writer.

In the concurrent read streams whose number is more than the fixed number of dedicated workers, however, there are still mixed read streams. To overcome this limitation, a dedicated worker manages and processes requests in the following way. First, it groups requests in a queue by their stream identifier (rs_id) and sorts them in the order of arrival. Thus, a dedicated worker keeps on serving one read stream without interleaving the others. Second, it gets the first request of the queue and processes it without removing it from the queue. After finishing processing the request, a dedicated worker eventually deletes it from the queue. Thus, newly arrived requests for a currently processing stream could be processed earlier than ones for the others.

6.2 APS based on dedicated I/O workers

First, the APS should make a storage server conduct prefetching (i.e., ra_s) by the VFS for an individual read stream. To this end, the APS needs to maintain the server's readahead context corresponding to the read stream. With the stream information pointing to the readahead context, as shown in Fig. 7, the APS manages a read stream with the following four steps. First, when a read stream opens a file, a client sends a request to create a stream to a storage server, and then, a mgmt processor of the server generates the new stream information with the following fields:

- rs_id: a stream identifier.
- worker_id: a dedicated worker identifier designated for the rs_id stream.
- fd: a file descriptor number pointing to a file object. It is acquired by opening a file which physically stores data at a storage server, thus creating a readahead context via the VFS.

Second, to determine which dedicated worker takes charge of the new stream, a mgmt processor selects one I/O worker among a pool of dedicated workers using the following function:

$$\text{First}(\text{Min\_st}(\text{Min\_req}(worker_{1,2,\ldots,n})))$$

where

(i) n denotes the number of dedicated workers at a storage server.
(ii) $\text{Min\_req}(worker_{1,2,\ldots,j})$ denotes a set of dedicated workers having the minimum number of requests among $worker_{1,2,\ldots,j}$.
(iii) $\text{Min\_st}(worker_{1,2,\ldots,j})$ denotes a set of dedicated workers having the minimum number of streams among $worker_{1,2,\ldots,j}$.
(iv) $\text{First}(worker_{1,2,\ldots,j})$ denotes the first worker of $worker_{1,2,\ldots,j}$ (i.e., $worker_1$).

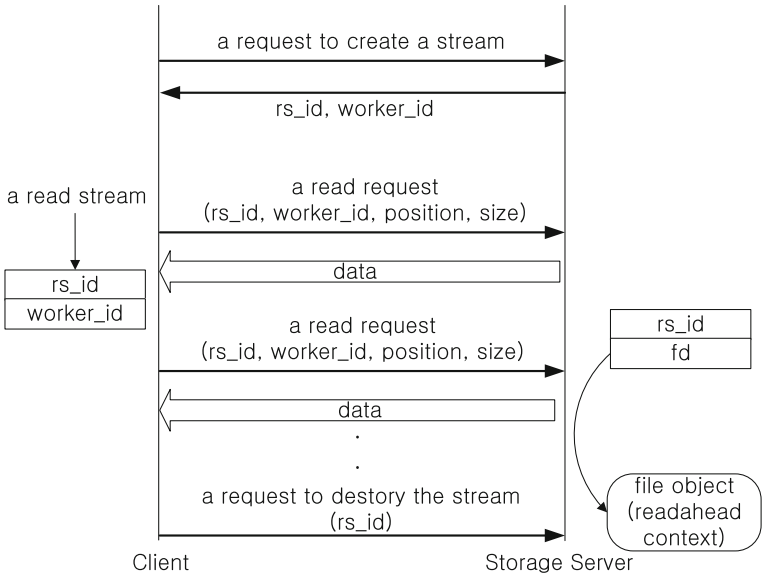

Fig. 7 Management procedure of the stream information

Third, whenever the read stream issues a sequential read request, the client converts the request into a networked request with a rs_id and a worker_id and then sends it to the server. At the server, a dedicated worker with the worker_id processes the request through a file descriptor of the corresponding stream information to the rs_id, thus automatically issuing a readahead at the server.

Finally, when the read stream closes the file, the client sends a request to destroy the rs_id stream to the server, and then, a mgmt processor closes the file descriptor (fd) of the stream information about the rs_id to destroy the readahead context at the server.

In addition to the stream information, the storage server of the APS requires adaptive prefetching information about a storage device. The adaptive prefetching information consists of (1) max_ra_sz (max readahead size), (2) RB (average read bandwidth), and (3) p (the ratio of the peak read performance to RB) for the device.

As shown in Fig. 8, with the adaptive prefetching information of a storage server, a client extracts the value of α for a read stream in the following way. When a storage server receives a request to create a stream (denoted by cs), a mgmt processor creates stream information and then sends to the client the created information as well as two additional fields, (1) the time taken to process the cs request (T[Proc$(cs)_s$]) and (2) the dummy max_ra_sz-sized data. With the time taken to receive a reply after sending the cs request (T[ReqRecv$(cs)_c$]), the client can calculate the duration of T[SendAndReq$(i, ra_s, i+\alpha+1)$] ($\because$ T[SendAndReq$(i, ra_s, i+\alpha+1)$] $\approx$ T[ReqRecv$(cs)_c$] $-$ T[Proc$(cs)_s$]). Finally, it selects an integer value (i.e., α) that is equal to or minimally larger than the left side's value of the below final condition:

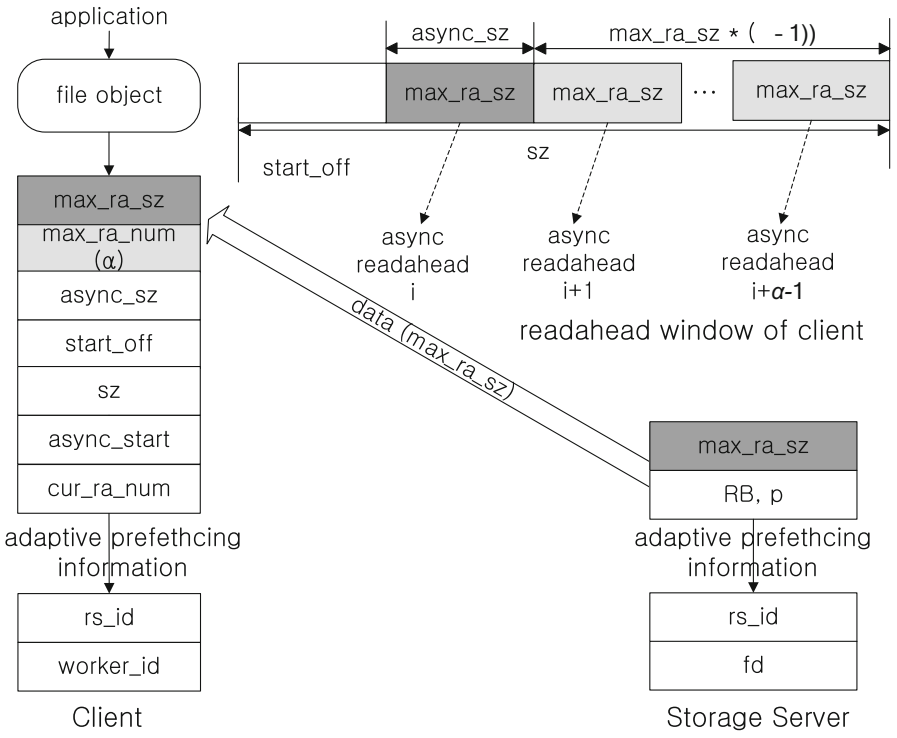

Fig. 8 The structure of the APS

$$\frac{\mathrm{T}[\mathrm{ReqRecv}(cs)_c] - \mathrm{T}[\mathrm{Proc}(cs)_s]}{\frac{max\_ra\_sz}{RB_{dev} * (1 + p_{dev})}} - 1 \leq \alpha \tag{20}$$

where $p_{dev} \geq 1$

According to the above condition, the α value gets extracted by taking into consideration both the type of storage device and the length of a network delay as follows. First, for the device type, $\frac{max\_ra\_sz}{RB_{dev} * (1 + p_{dev})}$ is calculated with adaptive prefetching information. Second, for a network delay between a client and a storage server, $\mathrm{T}[\mathrm{ReqRecv}(cs)_c] - \mathrm{T}[\mathrm{Proc}(cs)_s]$ is calculated with the $\mathrm{ReqRecv}(cs)_c$ and $\mathrm{Proc}(cs)_s$ operations, each of which has a network delay. Therefore, our adaptable α value can cope with different types of storage devices and various network delays.

To perform adaptive prefetching based on the extracted value of α, the client of the APS needs the following adaptive prefetching information:

- max_ra_sz: the max readahead size. It is set dynamically according to the corresponding device at a storage server.
- max_ra_num: the max number of asynchronous readaheads (i.e., α). This value is obtained by Condition (20) depending on the type of storage device and the delay time of a given network.
- async_size: the current readahead size. It can reach max_ra_size.

- start_off: the starting position of a readahead window to determine whether a read request is sequential.
- sz: the total size of a readahead window.
- async_start: the position of the next asynchronous readahead.
- cur_ra_num: the current number of asynchronous readaheads. Its value increases from 1 to α.

As presented in Algorithm 1, the client of the APS performs readaheads according to max_ra_sz and max_ra_num. While the readahead size is less than max_ra_sz, Algorithm 1 conducts prefetching in the same way as the existing readahead facility that multiplies the readahead size by either 2 or 4 and issues a single readahead. However, Algorithm 1 invokes readaheads in a different manner when the readahead size becomes equal to or greater than max_ra_sz. From that moment on, it extends the readahead size by max_ra_sz by increasing cur_ra_num until the cur_ra_num reaches max_ra_num. In addition, it keeps on issuing subsequent readaheads with the size of max_ra_sz until the readahead size reaches its maximum size (i.e., cur_ra_num $*$ max_ra_sz).

Based on max_ra_num (α) and max_ra_sz, our APS can dynamically conduct prefetching for each individual read stream according to the type of storage device and the delay time of a network.

Algorithm 1 Adaptable Prefetching Algorithm

Input: ra: readahead context, $size$: request size
Require: ra.sz $\geq size$
1: ra.sz = ra.sz $- size$
2: ra.start_off = ra.start_off $+ size$
3: **if** ra.async_sz $<$ ra.max_ra_sz **then**
4: **if** ra.ra_sz $<$ ra.async_sz **then**
5: ra.async_start = ra.async_start $+$ ra.async_sz
6: ra.async_sz = get_next_ra_size(ra,ra.max_ra_sz)
7: ra.sz = ra.sz $+$ ra.async_sz
8: send_async_read_req(ra.async_start,ra.async_sz)
9: **end if**
10: **else**
11: **if** ra.sz $\leq$ (ra.async_sz $*$ ra.cur_ra_num) **then**
12: **if** ra.cur_ra_num $<$ ra.max_ra_num **then**
13: ra.cur_ra_num++
14: **end if**
15: **while** ra.sz $\leq$ (ra.async_sz $*$ ra.cur_ra_num) **do**
16: ra.async_start = ra.async_start $+$ ra.async_sz
17: ra.sz = ra.sz $+$ ra.async_sz
18: send_async_read_req(ra.async_start,ra.async_sz)
19: **end while**
20: **end if**
21: **end if**

7 Performance evaluation

7.1 Experimental setup

To evaluate our proposed APS, we carried out experiments with the following running environment. We used six Linux 3.10.0-327 machines, each with two 3 GHz quad-core CPUs and 4 GB of memory. These machines were interconnected with a 10-Gigabit switch. Two of them acted as storage servers, and the others acted as clients. Each storage server had a RAID (SATA-III) controller which was connected to either four 6 TB hard disks (Seagate ST6000NM0024) or one 1 TB SSD (Samsung SSD 850 EVO). Additionally, we set up a striped RAID using four hard disks with a Linux S/W RAID configurator, Mdadmin. Its stripe size was set to 512 KB as the default.

To compare our APS with existing DFSs, we used Gluster version 3.7.8, Hadoop version 2.7.5 for HDFS, Lustre version 2.8.50, and NFS version 4 at the Linux Kernel. Unlike Gluster, HDFS and Lustre need an additional component such as a NameNode for HDFS and a metadata server for Lustre. For each component, we used a separate disk from the storage devices for the performance evaluation.

For consistent evaluations of all the DFSs, the I/O schedulers of all the devices at the storage servers were set to CFQ. Additionally, the low_latency option of the CFQ was disabled to enhance the performance of concurrent read streams in the LFS.

Finally, the following two DFSs had additional configurations. In the HDFS, the I/O transfer size (io.file.buffer.size) and the number of chunk replicas (dfs.replication) were set to 128 KB to support the 128 KB I/O size and to 0 to prevent data replication, respectively. In our APS, the number of dedicated I/O workers was set to 32 to show how read streams are handled when the number is more than the number of I/O workers, and all the values for $p_{\text{hard disk}}$, p_{ssd}, and p_{raid} were set to 1.2.

7.2 IOzone: a microbenchmark

We evaluated the read performance with an industry benchmark tool, IOzone [34]. We used a file size of 10 GB and an I/O unit of 128 KB for the concurrent read streams and a file size of 40 GB for the random reads. These file sizes in our experimental environment prevented data from being cached on the client and server sides.

In the HDFS without POSIX, using our benchmark program, we evaluated the performance in the same manner as IOzone. Unlike the concurrent read streams, the performance evaluation of the random reads required positional information about the random requests. To this end, we extracted the information from IOzone and then executed our benchmark program with it.

NFS was excluded in the case of two storage servers because it supported only a single storage server.

7.2.1 Single hard disk

According to EXT4 and all the DFSs, the performances of the concurrent read streams on a single hard disk are shown in Fig. 9a for a client and a storage server and Fig. 9b for four clients and two storage servers, respectively.

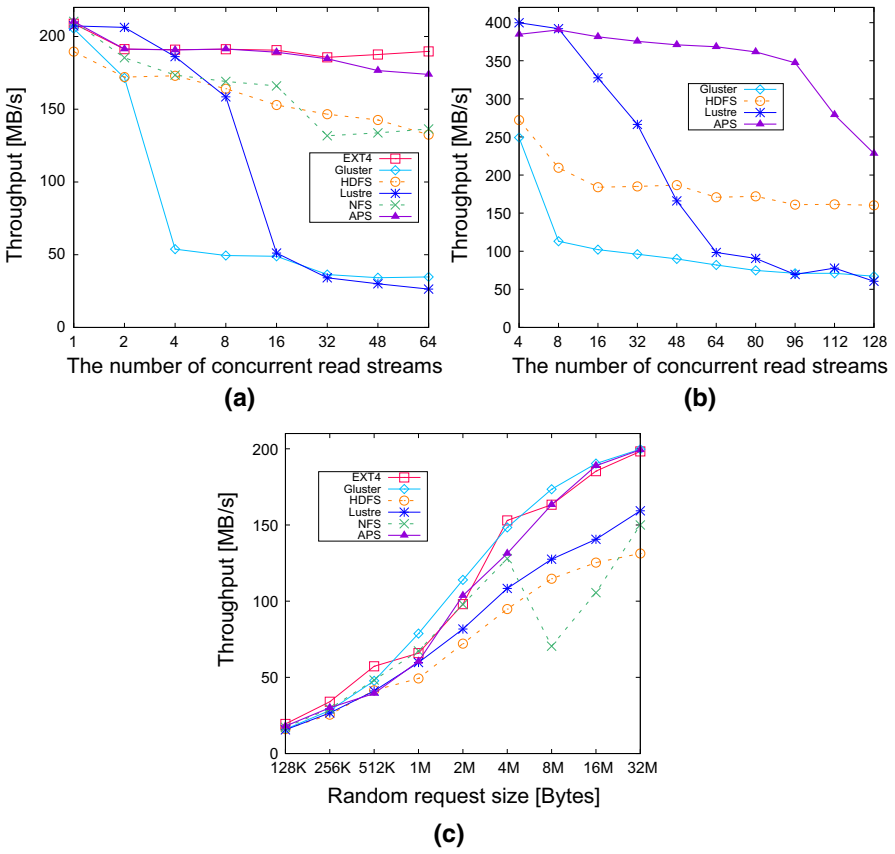

Fig. 9 Performance on a hard disk measured by IOzone. **a** Sequential reads in a client and a server. **b** Sequential reads in 4 clients and 2 servers. **c** Random reads in a client and a server

First, in the case of a client and a storage server, EXT4 has the best throughput among all the DFSs for all read stream cases except for two streams as expected. For all the existing DFSs, however, the throughputs for four or more streams are much lower than those in EXT4 due to the problem of mixed read streams. Especially in Lustre and Gluster, the throughputs become degraded dramatically from 4 and 16 streams, respectively. However, their throughputs for a single stream are slightly lower than that in EXT4 because they are not affected by the problem of mixed read streams. Interestingly, Lustre has the highest throughput among EXT4 and all the DFSs for two read streams with the help of its own underlying LFS (ldiskfs). On the other hand, HDFS and NFS could barely maintain a little high throughput for large numbers of read streams unlike the other existing DFSs because HDFS conducts aggressive prefetching at a storage server and NFS has a large readahead size (7.5 MB) at a server. However, their prefetching methods could incur many useless reads for random reads.

Unlike the existing DFSs, the APS has the same throughput as EXT4 at equal to or fewer than 32 streams. However, its throughput is degraded slightly at the 48 and 64 stream cases because of the insufficient number of dedicated I/O workers (i.e., 32).

Second, in the case of two storage servers, despite four clients, all the DFSs except for HDFS have almost doubled their throughputs compared with the case of a single storage server. The reason why HDFS fails to achieve a scaled performance is as follows. First, it fails to distribute 64 MB chunks evenly across the storage servers. Second, to read a file, HDFS should open and close many split chunks at different servers. On the other hand, Lustre achieves a little better individual throughput from two servers at 32 streams than from one server at 16 streams, and the APS obtains a more degraded individual throughput from two servers at 128 streams than from one server at 64 streams.

The performance result of multiple clients and storage servers leads to the following facts. First, the aggregated performance of scaled DFSs mainly depends on (1) the prefetching performance between a client and a storage server and (2) the resource distribution policy among multiple storage servers. Second, a single client can accommodate a large number of read streams without multiple clients only if the network bandwidth between it and its storage servers is available.

As shown in Fig. 9c, the performance of random reads tends to degrade in proportion to the number of useless reads.

In Lustre, NFS, and HDFS, the random throughputs become worse as the request size increases. Due to the largest readahead size at a client (i.e., 40 MB) among all the DFSs, Lustre invokes no useless reads but fails to achieve the PNR effect at the smaller request sizes than its readahead size, i.e., the high transfer cost of RA_c due to the lack of the PNR effect. Unlike Lustre, NFS incurs many useless reads at the larger request sizes than its large readahead size (i.e., 7.5 MB), i.e., the high cost of useless reads due to RA_c. These are expected results by RA_c as mentioned in Sect. 5. Like NFS, on the other hand, HDFS incurs many useless reads due to aggressive prefetching at a storage server.

However, the APS achieves a similar performance to EXT4 for all request sizes because it has the least useless reads among all the DFSs except for Gluster.

In Gluster, on the other hand, the random throughput is slightly better than that even in EXT4 at most request sizes. We assume that its own prefetching algorithm does not conduct prefetching on a hard disk at a client and a storage server.

Consequently, among the existing DFSs, HDFS and NFS have the highest performance for concurrent read streams but the lowest performance for random reads. This result indicates that it is designed to achieve high performance for concurrent read streams by sacrificing the performance of random reads. However, our APS achieves almost the same performance as a LFS (EXT4) without degrading the random performance.

7.2.2 Striped RAID

The performance results for concurrent read streams on a striped RAID are shown in Fig. 10a for a client and a storage server and Fig. 10b for four clients and two storage servers, respectively.

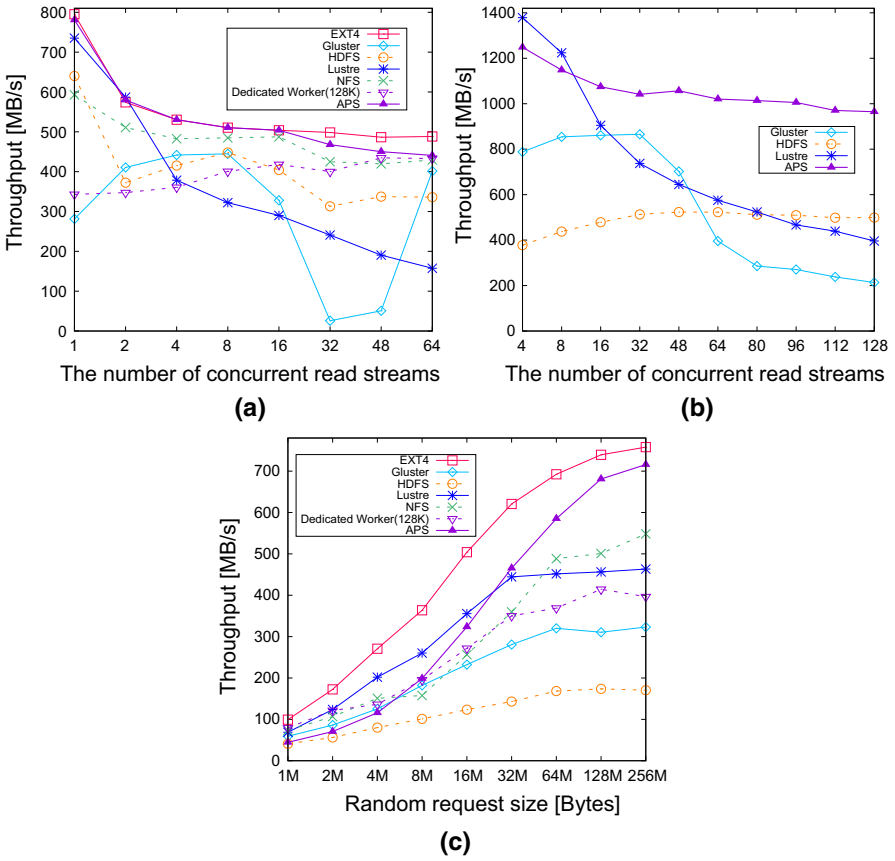

Fig. 10 Performance on a striped RAID (four disks, 512 KB strip size) measured by IOzone. **a** Sequential reads in a client and a server. **b** Sequential reads in 4 clients and 2 servers. **c** Random reads in a client and a server

First, in the case of a client and a storage server, all the throughputs of the existing DFSs are lower than that of EXT4 due to the problems that we defined in Sect. 3.2. Especially in Gluster and Lustre, the throughputs get degraded dramatically with respect to the increasing number of read streams.

Compared with Gluster and Lustre, HDFS has a much more degraded throughput for two streams but much higher throughputs for four or more streams. The reason is that the prefetching mechanism of HDFS (i.e., aggressive server-side prefetching) on a striped RAID frequently fails to meet the PNR condition for a small number of streams but only occasionally fails to satisfy it for a large number of streams.

Among all the existing DFSs, NFS has the highest throughput for two or more read streams because it has a large readahead size (7.5 MB) at a server. In comparison with EXT4 and APS, however, the NFS still has a lower throughput for a small number of streams because it frequently fails to meet the PNR condition.

Unlike the existing DFSs, the APS obtains the same throughput as EXT4 for 32 or fewer streams. However, it has a slightly degraded throughput for the 48 and 64 read

stream cases due to the same reason as on a hard disk. On the other hand, the dedicated I/O worker (part of the APS) has much lower throughputs for a small number of streams than the whole APS because it hardly satisfies the PNR condition on a striped RAID.

Second, in the case of four clients and two storage servers, all the DFSs except for HDFS have almost doubled their throughputs compared with the case of a single storage server, just like on a hard disk. For the same reason as on a hard disk, HDFS does not achieve a scaled performance on a striped RAID. Interestingly, different from a hard disk, the APS maintains the doubled throughput even at the 112 and 128 stream cases with the help of its large readahead size (512 KB).

Figure 10c shows the performance of the random reads for a striped RAID according to the request sizes from 1 to 256 MB.

In Gluster, different from a hard disk, the random throughput becomes seriously degraded according to the increasing request size because it can hardly satisfy the PNR condition on a striped RAID.

Due to the same reason as on a hard disk, HDFS achieves the lowest performance of the random reads among all the DFSs, and NFS has a highly degraded performance at request sizes that are larger than 7.5 MB.

In Lustre, the random throughput is the highest among all the DFSs at request sizes that are 16 MB or smaller because there are no useless reads. However, it cannot enhance the throughput at request sizes that are larger than 16 MB just like on a hard disk.

Unlike the existing DFSs, as the request size becomes larger, the APS achieves the highest throughput of random reads. However, it has a lower throughput than that of Lustre for small request sizes due to useless reads. Compared with the APS, on the other hand, only the dedicated I/O worker has lower throughputs for large request sizes due to the failure to satisfy the PNR condition but higher throughputs for small ones because of fewer useless reads.

As a result of this performance evaluation, a DFS should conduct prefetching in such a way that it is aware of the type of striped RAID. Otherwise, the performances in both concurrent read streams and random reads become seriously degraded like in existing DFSs.

7.2.3 SSD

The performances for concurrent read streams on a SSD are given in Fig. 11a for a client and a storage server and Fig. 11b for four clients and two storage servers.

First, in the case of a client and a storage server, because of a short seek time, the concurrent read streams on a SSD are much less affected by the problems of a DFS than those on a hard disk and a striped RAID. However, HDFS, Gluster, and only the dedicated I/O worker still have lower throughputs because they fail to meet the PNR condition. Interestingly, Gluster obtains much lower throughputs at a small number of streams (i.e., one and two) than those of the other DFSs; however, it achieves the same performance as EXT4 at four or more streams. The reason is that Gluster fails to meet an individual PNR condition for each stream; however, it satisfies the whole PNR condition for all concurrent streams on a SSD. More specifically, among the concurrent streams, even if a failure to meet the PNR condition happens in each read stream, a

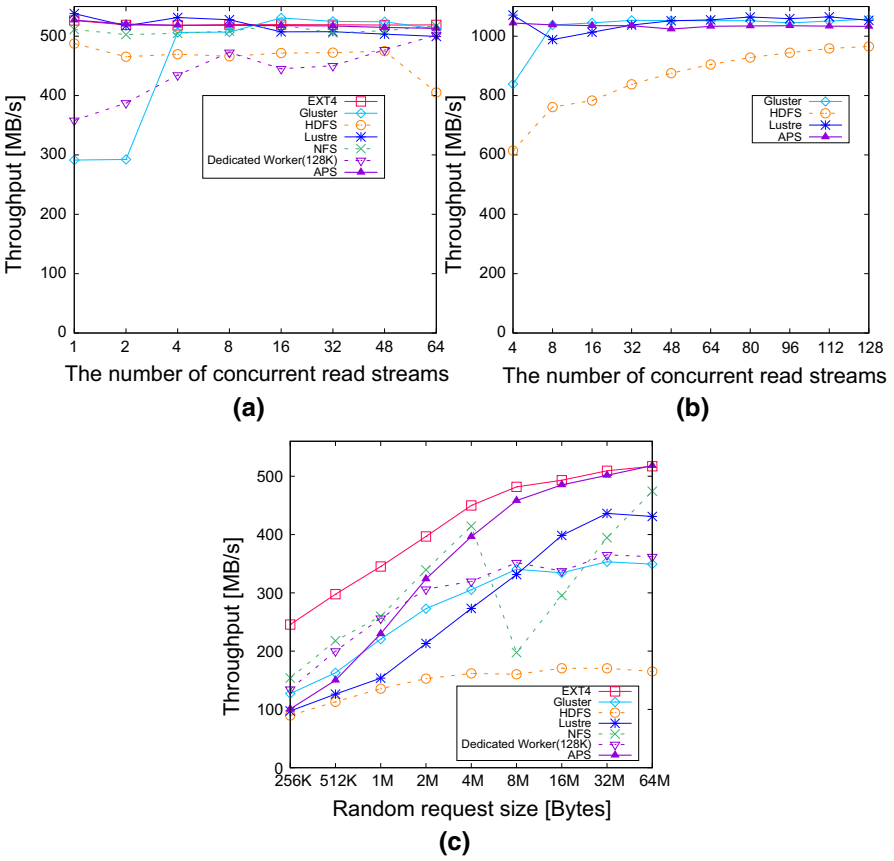

Fig. 11 Performance on a SSD measured by IOzone. **a** Sequential reads in a client and a server. **b** Sequential reads in 4 clients and 2 servers. **c** Random reads in a client and a server

storage server can choose another stream with the very low cost of a randomized seek to a SSD and then process its sequential requests without halting the SSD, which leads to satisfying the whole PNR condition for all concurrent streams.

Different from HDFS, Gluster, and only the dedicated worker, APS achieves almost the same performance as EXT4 for all read stream cases including the 48 and 64 stream cases.

Second, in the case of four clients and two storage servers, all the DFSs except for HDFS have almost doubled their throughputs for the case of a single storage server, as on a hard disk and a striped RAID. Due to the same reason as on a hard disk and a striped RAID, HDFS does not achieve a scaled performance on a SSD. As on a striped RAID, on the other hand, APS maintains the doubled throughputs even for the 112 and 128 stream cases because of the additional readaheads at a client (i.e., α).

Unlike the performance of the concurrent read streams, the random performance difference among the DFSs is very large shown in Fig. 11c. This large difference is mainly due to the different numbers of useless reads among the DFSs. Especially

in HDFS and NFS, the throughputs become degraded dramatically according to the increasing request size due to the same reason as on a hard disk and a striped RAID.

In Gluster, different from a hard disk, the random throughput becomes degraded according to the increasing request size. The reason is that Gluster can hardly satisfy the PNR condition on such a fast device.

Unlike the existing DFSs, APS achieves the highest throughput of random reads from a 1 MB request size and onwards. For the same reason as on a striped RAID, on the other hand, only the dedicated I/O worker has low throughputs for large request sizes but high throughputs for small ones.

Consequently, some of the existing DFSs can achieve the concurrent read performance that they expect for a fast SSD without resolving the problems that seriously degrade the performance on a hard disk and a striped RAID. However, they obtain such a performance at a high cost, such as a seriously degraded performance in the random reads. In contrast, our APS obtains the same performance as a LFS on a SSD by resolving the problems, which leads to only a small degradation in the performance of the random reads.

7.2.4 Network delay

To make a delayed network in our experimental environment, we inserted a pair of delays on the client and server sides. In our delayed network, we evaluated the performances of the DFSs for an individual read stream with respect to the delay time shown in Fig. 12.

In Fig. 12a, first, the throughput of Gluster becomes degraded dramatically from a 100 us delay. Similarly, only the dedicated I/O worker becomes seriously degraded from a 400-us delay. This result indicates that Gluster and only the dedicated I/O worker are very sensitive to network delays.

Unlike Gluster and the dedicated worker, Lustre, HDFS, NFS, and APS maintain high throughputs up to a 4-ms delay. However, from an 8-ms delay, their throughputs become degraded in a similar manner. This result indicates that they are insensitive up to a 4-ms delay.

On the other hand, the APS increases the α value in proportion to the length of the delay. However, its throughput becomes degraded like Lustre, HDFS, and NFS. This throughput degradation is caused by the insufficient window size of the TCP (i.e., 6 MB as the default in our Linux machines). More specifically, because the 6 MB window size is not enough for a client to hold a series of 50 readaheads with a 128 KB size, a storage server cannot send more than 50 readaheads through a network. Therefore, this limitation causes the network sensitivity of all the DFSs for delays that are longer than 8 ms.

To overcome the insufficient TCP window, another performance evaluation was conducted with a 30 MB window size shown in Fig. 12a. This evaluation result is quite different from the previous one. Especially, the APS is no longer sensitive from an 8-ms delay and onwards; however, Lustre and NFS are still sensitive from a 12- and 16-ms delay, respectively. Interestingly, unlike Lustre and NFS, HDFS is slightly degraded from a 12-ms delay and onwards because of heavy aggressive prefetching at the server.

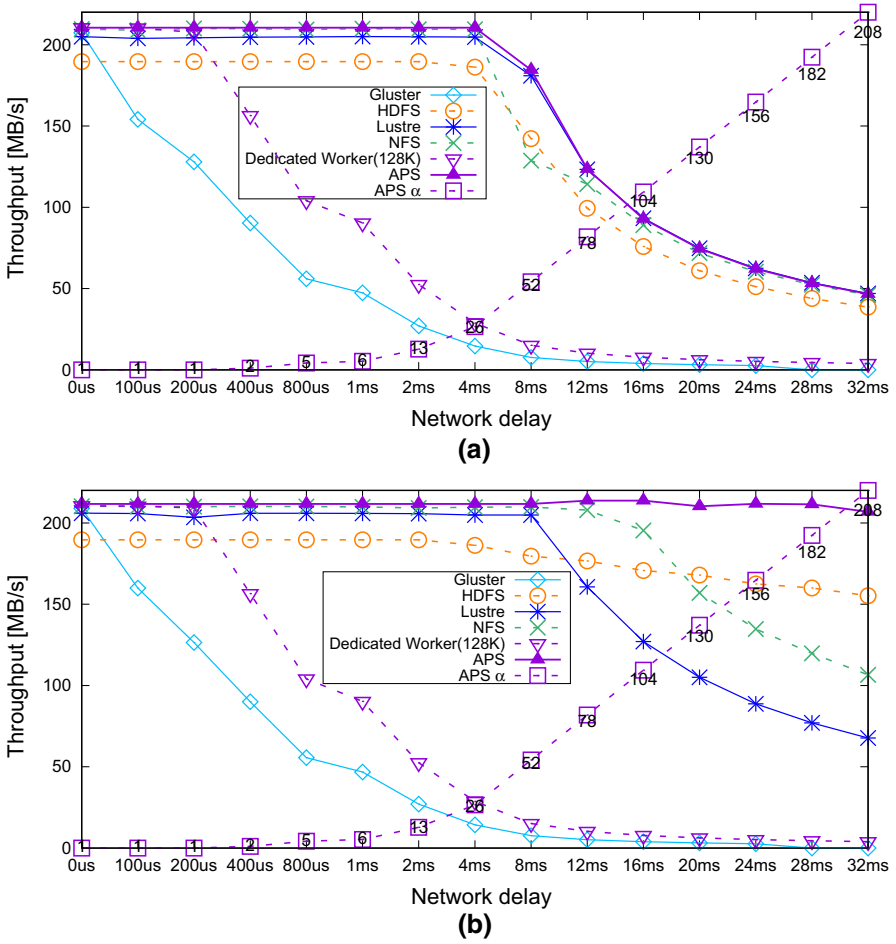

Fig. 12 Performance on a hard disk according to the network delay measured by IOzone. **a** 6 MB TCP window size (default). **b** 30 MB TCP window size

Based on the results of these performance evaluations, as the performance of the random reads becomes lower, the existing DFSs achieve a high performance in long delays. As a typical example, NFS and HDFS had a bad performance for the random reads on three types of storage devices, while they were insensitive to long delays. Therefore, this result shows that Theorem 1 is correct. Different from the existing DFSs with a fixed prefetching policy, our APS conducted delay-aware prefetching to minimize the random performance on the three types of storage devices for short delays.

7.3 FileBench: a microbenchmark

For real workloads, we used the Web-server workload of the latest FileBench (version 1.5-alph3) [1]. The Web-server workload was set with the following default

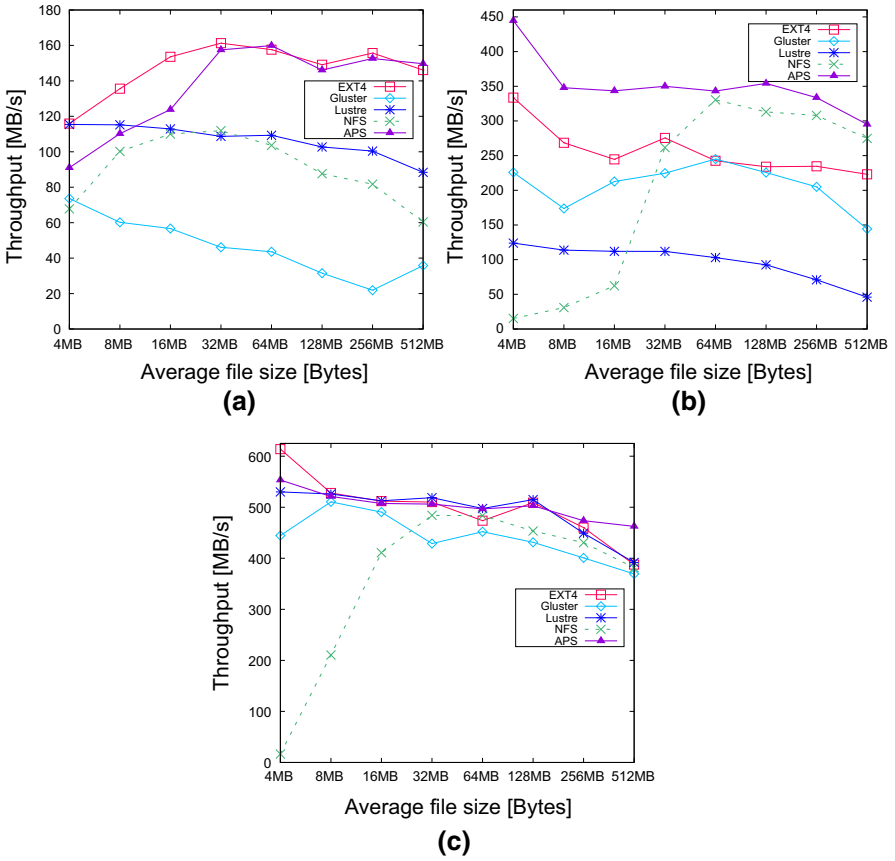

Fig. 13 Performance for the Web-server workload in a client and a server measured by FileBench. **a** A hard disk. **b** A striped RAID (four disks, 512 KB strip size). **c** An SSD

values: (1) the number of files ($nfiles) was 1000; (2) the number of threads ($nthreads) was 100; and (3) the I/O size ($iosize) was 1 MB. We set the average file size ($filesize) to be 4 MB or larger to prevent data from being cached on the client and server sides.

HDFS without POSIX was excluded in this evaluation because it cannot execute FileBench. On a SSD, the slice_idle value of the CFQ was set to 0 ms to improve the performance of the concurrent read streams for small files in a LFS (EXT4).

For the three device types, the throughputs for the Web-server workload are shown in Fig. 13 according to the average file size.

On a hard disk, the throughputs of all the existing DFSs become degraded dramatically with respect to the increasing average file size mainly due to the mixed read streams. However, the throughput of the APS increases according to the increasing file size. Especially, Lustre has the highest throughput among all the DFSs for 4 and 8 MB file sizes with the help of its own underlying LFS (ldiskfs).

On a striped RAID, in all existing DFSs, the performances of all the existing DFSs are much lower than that of EXT4 due to the problems that we defined in Sect. 3.2.

Unusually, NFS has the lowest throughput among all the DFSs for small file sizes due to the high cost of useless reads on the client and server sides. On the other hand, the throughput of the APS is the highest among all the DFSs for all file sizes. Interestingly, APS obtains a higher performance than even that of EXT4 for the following two reasons. First, the Web-server workload of FileBench is generated by 100 threads, and each of the many threads issues sequential reads in 1 MB units. Second, the performance of a striped RAID decreases in proportion to the number of concurrent read streams. Finally, unlike EXT4, the APS processes read requests at a storage server with only 32 I/O workers regardless of the number of workload-generating threads at a client.

On a SSD, the throughputs of EXT4 and Lustre look similar to the APS because of a short seek time. However, due to the same reason stated in the case of a striped RAID, the throughputs of Lustre and EXT4 decrease from a 256 MB file size and onwards. On the other hand, NFS has the lowest throughput among all the DFSs for small file sizes, just like on a striped RAID. In comparison with Lustre and APS, Gluster has lower throughputs for all file sizes due to the frequent failures in achieving the PNR effect.

8 Conclusion

This paper revealed that most of the existing DFSs had a highly degraded performance for concurrent read streams in different running environments and suffered from performance degradation in random reads. In addition, we identified the reason behind the degraded performance by defining two problems in DFSs. The main reason for the performance degradation is that DFSs conduct prefetching in a fixed way without regard to the type of storage device and the delay time of a network.

To resolve such a fixed method, we proposed an adaptable prefetching scheme (APS) for different running environments. Additionally, we discovered an inverse relationship between better performance for random reads and less sensitivity to long network delays in DFSs and then verified it by comparing two performance evaluations for random reads and long network delays.

With an adaptable scheme, our APS conducts prefetching in an appropriate manner according to a specific running environment. Thus, with as little performance degradation of the random reads as possible, it obtains the performance that we expect for a certain type of storage device and maintains the high performance in various network delays.

However, there are still improvements to be made in our research work:

- To extract the α value more accurately, another network delay estimating method is needed to deal with the dynamically changing network traffic.
- Our work does not consider all I/O schedulers (e.g., CFQ, Deadline, and Anticipatory) that Linux provides. Thus, further research needs to be done to achieve the high performance of the concurrent read streams even for different types of I/O schedulers.
- Further optimization of our work is required with some metaheuristic algorithms (e.g., monarch butterfly optimization (MBO) and earthworm optimization algorithm (EWA)).

Acknowledgements This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0126-15-1082, Management of Developing ICBMS (IoT, Cloud, Bigdata, Mobile, Security) Core Technologies and Development of Exascale Cloud Storage Technology).

References

1. A file system and storage benchmark. https://github.com/filebench/filebench/wiki. Accessed Mar 2018
2. Baek SH, Park KH (2009) Striping-aware sequential prefetching for independency and parallelism in disk arrays with concurrent accesses. IEEE Trans Comput 58(8):1146–1152
3. Chen M et al (2017) vNFS: maximizing NFS performance with compounds and vectorized I/O. ACM Trans Storage (TOS) 13(3):21
4. Cooper BF et al (2010) Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing. ACM
5. Ding X et al (2007) DiskSeen: exploiting disk layout and access history to enhance I/O prefetch. In; USENIX Annual Technical Conference, vol 7
6. Dong B et al (2010) Correlation based file prefetching approach for hadoop. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom). IEEE
7. Ellard D, Seltzer MI (2003) NFS tricks and benchmarking traps. In: USENIX Annual Technical Conference, FREENIX Track
8. Feiyi W et al (2009) Understanding lustre filesystem internals. Oak Ridge National Laboratory, National Center for Computational Sciences, Technical Report
9. Fengguang WU, Hongsheng XI, Chenfeng XU (2008) On the design of a new linux readahead framework. ACM SIGOPS Oper Syst Rev 42(5):75–84
10. Ghemawat S, Gobioff H, Leung S-T (2003) The Google file system. In: ACM SIGOPS Operating Systems Review, vol 37, no. 5. ACM
11. Gill BS, Bathen LAD (2007) Optimal multistream sequential prefetching in a shared cache. ACM Trans Storage (TOS) 3(3):10
12. Gluster File System. http://www.gluster.org. Accessed Mar 2018
13. Hong J et al (2016) Optimizing Hadoop framework for solid state drives. In: IEEE International Congress on Big Data (BigData Congress), 2016. IEEE
14. Islam NS et al (2016) High performance design for HDFS with byte-addressability of NVM and RDMA. In: Proceedings of the 2016 International Conference on Supercomputing. ACM
15. Jiang S et al (2013) A prefetching scheme exploiting both data layout and access history on disk. ACM Trans Storage (TOS) 9(3):10
16. Lee HK, An BS, Kim EJ (2009) Adaptive prefetching scheme using web log mining in Cluster-based web systems. In: IEEE International Conference on Web Services, 2009. ICWS 2009. IEEE
17. Liang S, Jiang S, Zhang X (2007) STEP: sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In: 27th International Conference on Distributed Computing Systems (ICDCS'07). IEEE
18. Li C, Shen K, Papathanasiou AE (2007) Competitive prefetching for concurrent sequential I/O. In: ACM SIGOPS Operating Systems Review, vol 41(3). ACM
19. Martin RP, Culler DE (1999) NFS sensitivity to high performance networks. ACM SIGMETRICS Perform Eval Rev 27(1):71–82
20. Mikami S, Ohta K, Tatebe O (2011) Using the Gfarm File System as a POSIX compatible storage platform for Hadoop MapReduce applications. In: Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing. IEEE Computer Society
21. Pai R, Pulavarty B, Cao M (2004) Linux 2.6 performance improvement through readahead optimization. In: Proceedings of the Linux Symposium, vol 2
22. Palankar MR et al (2008) Amazon S3 for science grids: a viable solution? In: Proceedings of the 2008 International Workshop on Data-Aware Distributed Computing. ACM
23. Papagiannaki K et al (2002) Analysis of measured single-hop delay from an operational backbone network. In: Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. INFOCOM 2002. IEEE, vol 2. IEEE
24. Papagiannaki K et al (2003) Measurement and analysis of single-hop delay on an IP backbone network. IEEE J Sel Areas Commun 21(6):908–921

25. Pillai TS et al (2017) Application crash consistency and performance with CCFS. FAST, vol 15
26. Rago S, Bohra A, Ungureanu C (2013) Using eager strategies to improve NFS I/O performance. Int J Parallel Emerg Distrib Syst 28(2):134–158
27. Roselli DS, Lorch JR, Anderson TE (2000) A comparison of file system workloads. In: USENIX Annual Technical Conference, General Track
28. Saini S et al (2012) I/O performance characterization of Lustre and NASA applications on Pleiades. In: 2012 19th International Conference on High Performance Computing (HiPC). IEEE
29. Shafer J, Rixner S, Cox AL (2010) The hadoop distributed filesystem: balancing portability and performance. In: 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). IEEE
30. Shriver EAM, Small C, Smith KA (1999) Why does file system prefetching work? USENIX Annual Technical Conference, General Track
31. Shvachko K et al (2010) The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). IEEE
32. Soundararajan G, Mihailescu M, Amza C (2008) Context-aware prefetching at the storage server. In: USENIX Annual Technical Conference
33. Sur S et al (2010) Can high-performance interconnects benefit hadoop distributed file system. In: Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds (MASVDC). Held in Conjunction with MICRO
34. The IOzone Benchmark. http://www.iozone.org. Accessed Mar 2018
35. Walker E (2006) A distributed file system for a wide-area high performance computing infrastructure. WORLDS. Vol. 6
36. Weil SA et al (2006) Ceph: A scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. USENIX Association
37. Welch B et al (2008) Scalable performance of the panasas parallel file system. FAST, vol 8
38. Wu F et al (2007) Linux readahead: less tricks for more. In: Proceedings of the Linux Symposium, vol 2
39. Yadgar G et al (2008) Mc2: multiple clients on a multilevel cache. In: The 28th International Conference on Distributed Computing Systems, 2008. ICDCS'08. IEEE
40. Yadgar G et al (2011) Management of multilevel, multiclient cache hierarchies with application hints. ACM Trans Comput Syst (TOCS) 29(2):5
41. Zhang Z et al (2008) Pfc: transparent optimization of existing prefetching strategies for multi-level storage systems. In: The 28th International Conference on Distributed Computing Systems, 2008. ICDCS'08. IEEE