An Efficient Server-Side Prefetching Scheme to Optimize Performance of Distribution File Systems

Yong Li*, Shuibing He‡, Qian Zhao†, Zhan Shi✉†, Yi Qin*, Weixu Zong*, Peng Xu*, Lingfang Zeng✉*
*Research Center for High Efficiency Computing Infrastructure, Zhejiang Lab
†Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
‡The State Key Laboratory of Blockchain and Data Security, Zhejiang University

***Abstract*—Because of the increasing speed gap between speed of compute and storage, caching is critical for improving the throughput of distributed file systems. It has been shown that prefetching can hide the latency resulted by network communication or disk operations. However, conventional client-based prefetching schemes are not efficient in distributed file systems as the limited computing and memory power of client nodes. In this paper, we present an effective and load-aware server-side prefetching scheme for distributed file systems, name SSPF. As an orthogonal approach, SSPF can be coupled with any existing caching scheme. First, SSPF exploits spatial locality to improve the efficiency of the prefetching cache and minimize memory requirement. Then, for maximizing the efficiency of cache, a multi-queue based cache manager is designed to coordinate between the prefetching blocks and other caching blocks. Furthermore, a heuristic-based request distribution strategy is proposed to optimize the balance between data server nodes and improve the overall performance. Finally, we have implemented and evaluated SSPF on the real distributed file system. Experimental results show that SSPF can significantly improve the read performance with negligible memory overhead.**

***Index Terms*—Prefetch, Cache, Distribution file systems, Server-side, Load balance.**

I. Introduction

In the era of Big Data, the amount of data in the world is exploding. According to International Data Corp. (IDC) report [1], the global data volume will grow to 175ZB in 2025, ten times the 16.1ZB of data generated in 2016. Distributed file system, which is well-known for its scalability and reliability, provides storage capabilities for such large amounts of data. However, the performance of disk is far slower than processor and memory. In addition, the performance of disk increases are also slower than processor and memory increases [2]. The increasing gap between computing and storage has become a major performance drawback of distributed file systems.

Caching is a common approach to improve the disk performance in distributed file systems. One approach is to cache popular data in client-side cache and access the data directly from the local cache when the cache hits, thus avoiding accessing data from remote server nodes [3]–[5]. Another approach is to add a caching layer to the I/O hierarchy of a distributed file system, aggregating multiple small, discrete requests into larger, contiguous requests, with the result that the average latency is improved by accessing more data from high-speed memory in cache layer [6], [7]. Many HPC systems use burst buffers to absorb the bursty I/O of scientific applications and relax the I/O provisioning requirement of the distributed file systems [8]–[10].

Unlike caching, prefetching scheme for server-side cache in distributed file systems have rarely been studied. In distributed file system, prefetching data from disk to memory before it is accessed can hide visible I/O costs, thus alleviating the performance bottleneck of system. However, there are several challenges in implementing prefetching scheme on the server-side cache of distributed file systems.

1) There is a risk of wasting time by prefetching data that will not be used. The penalty of prefetching misses in server-side cache is less than that in client-side cache, because of prefetching to the client-side cache requires one additional network transmission.
2) Applications access data from the data server nodes only when the client's local cache miss. As a result, it is hard to obtain the accurate access pattern of an application on the server side. Therefore, the common prefetching scheme based on access pattern prediction is inefficient on the server side.
3) Because the server-side cache is shared by many clients, prefetched data may be evicted without being accessed due to limited cache. It is difficult to decide which data should cache and prefetch to maximize overall performance.

In this paper, we present the design and implementation of SSPF: an effective and load-aware prefetching scheme for distributed file systems with restricted memory. SSPF uses a combination of new and existing techniques that are carefully engineered to achieve this goal. 1) SSPF use the hints about access patterns provided by the client to help data server prefetches only the next data block into the cache. The idea is to use spatial locality to improve the efficiency of the prefetching cache and reduce the cache requirements of prefetching. 2) If a data request arrives while prefetching is not complete, it will be pending until completion, which increases access latency. We use two approaches to alleviate this problem : One approach tries to distribute data requests to least-loaded data servers; Another approach is to distribute data requests to different data servers in a round-robin fashion, giving each data server

enough time to complete prefetching. The idea of both approaches is to improve parallelism between data servers and reduce access latency. 3) SSPF splits the data processing into multiple stages, including first prefetching data to the server-side cache, then transferring it to the client-side cache, and finally completing the data processing. SSPF uses this multi-stage processing to comprehensively schedule client-side computing, network transfer and server-side disk I/O, increasing the overall throughput of the system.

In brief, we make the following three contributions:

1) We design and implement SSPF, a server-side prefetching scheme for distributed file systems, which only needs a small amount of cache. Then a hybrid cache manager is introduced to coordinate SSPF with a wide range of existing caching schemes.
2) We introduce two data parallel access methods to improve the system throughput. One method is to use a load-aware request distribution strategy, which distributes requests in a round-robin manner, to improve parallelism among data server nodes. The other method is to use a pipeline architecture to interleave server-side prefetching, network transfer, and client-side processing to increase parallelism among client and server nodes.
3) We have implement and evaluate SSPF on a real distributed file system. The experimental results show that SSPF can significantly improve I/O performance.

II. Design

A. Architecture Overview

Figure 1 shows the high-level architecture of SSPF. Distributed file system typically decouples data and metadata management to provide better performance and scalability. Metadata server stores, manages and delivers metadata of entire file system while coordinating security, consistency and load-balancing. It also determines the mapping of blocks to data servers. Data servers are responsible for storing data and serving read and write requests. When a client wants to read data, it first contacts the metadata server to query where the data should be read from. After that, the client has the location of the data servers and sends read requests to them. SSPF implements a prefetching scheme on the server-side cache, which prefetch data into the cache of the data server before the read request arrives, reducing disk I/O waiting time. The prefetching scheme can effectively reduce the overall read latency by increasing parallelism between the network and the disk.

The SSPF consists of four major components, whose responsibilities are described as below:

1) Prefetcher in client: The responsibility of this component is to transform file layout into the access sequence of block for each data server, and send it to data server as prefetching hint before the file is accessed.
2) Prefetcher in data server: The responsibility of this components is to prefetch data blocks from disk into cache and manage the lifetime of these prefetched blocks.
3) Cache manager in data server: The manager coordinate prefetching blocks with caching blocks managed by other caching schemes, such as LRU, to improve the efficiency of the entire cache system.
4) Load balancer in metadata server: Balancer evenly schedules requests to underloaded data servers to improve parallelism between data servers, giving each data server enough time to complete prefetching and reducing disk latency.

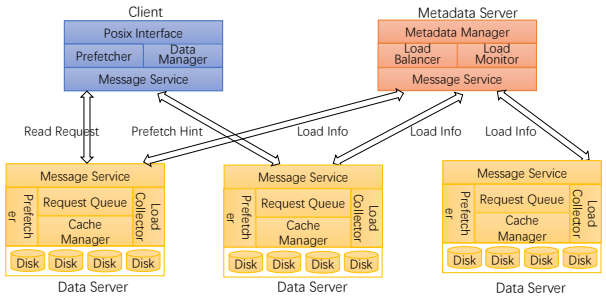

Fig. 1. The high-level architecture of SSPF.

B. Prefetching

The challenge of server-side prefetching is that it is difficult to get the exact access pattern of the application because of the two-level cache in the distributed file system, where the popular blocks are always cached in the client's local cache. The idea of SSPF is to use the file layout to predict next accessed block. However, the file metadata obtained from the metaserver is organized on a per-file basis, as shown in Figure 2. Each file is stored as a sequence of fixed-length blocks. Data distribution algorithms replicate these blocks to different data servers for fault tolerance, such as Swift's consistent hashing and Ceph's CRUSH. It is not possible to obtain the access pattern of data servers directly from these metadata. SSPF reorganizes the layout of file blocks on the basis of data server, as show in Figure 3. Note that it only needs to access one replica for each block. The replica selection is described in Section 2.4. Then, the reorganized sequence of blocks is send to the data server as prefetching hints before the block is accessed. Figure 4 shows the internals of the prefetching hint message. It places the $OpType$ used to indicate the type of message on the first 1-bytes. The $ClientId$ and $FileId$, whose size are both 8-bytes, are unique identifiers of the client and the file. The $BlockNum$ describes the total number of blocks in this message, followed by multiple 16-byte records. Each record describes the information of the block, including 8-bytes of $BlockId$, 4-bytes of offset and 4-bytes of length.

SSPF uses the client_session structure to store the information about the currently accessed data, as shown in table I. SSPF uses the file_block_map structures to store the prefetching hint, which contains a unique $FileId$ and one

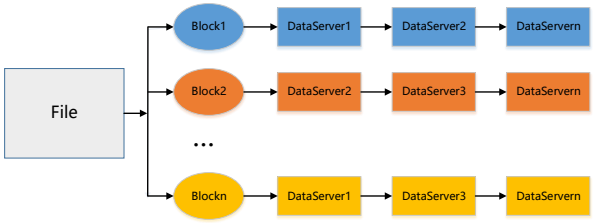

Fig. 2. The layout of file blocks in metadata (take a file with replication factor 3 as an example).

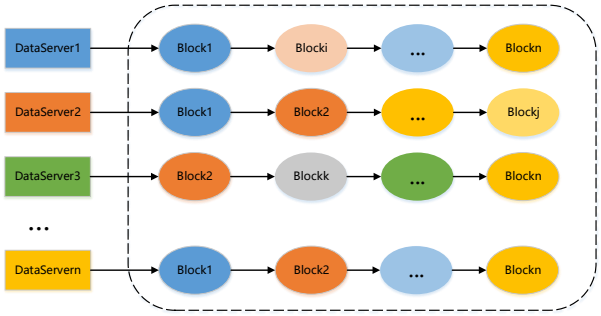

Fig. 3. Reorganized layout of file blocks (take a file with replication factor 3 as an example).

or more $BlockInfo$. Algorithm 1 presents the prefetching process. The prefetching thread first gets the block being accessed currently from client_session, and then gets the information of next block from file_block_map. If the data of next block is not in cache, the thread will prefetches the block from the disk to the cache. SSPF then checks whether the previously prefetched block is accessed, and if so, proceeds to prefetch the next block. Otherwise, if the prefetched block is skipped, it will finish this prefetch and clean the prefetched blocks.

TABLE I
THE STRUCTURE OF CLIENT_SESSION.

Name	Description
ClientId	The unique identifier of a client node
FileId	The unique identifier of a file
BlockId	The unique identifier of a block
BlockVersion	The version of the block

In addition, file_block_map is responsible for maintaining the state of the prefetching block : $unprefetch$, $prefetching$, $prefetched$, $accessed$. The state of block will be changed to $prefetching$ once the prefetching is started and to $prefetched$ when the prefetching is finished. During prefetching, if the block is already in the cache, it's status is directly changed to $prefetched$ to avoid duplicating disk I/O. The prefetched blocks should be cached with high priority to avoid these blocks being evicted before accessed. Once the client has fetched the block, the state of the block is changed to $accessed$. Then, these $accessed$ blocks will be given low priority because it may be cached in the local cache of client and will not be accessed in the near future. If the block has

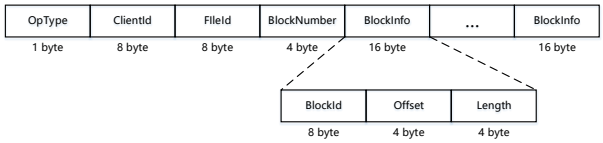

Fig. 4. The internals of the prefetching hint message.

Algorithm 1 The prefetching process.

```
1:  procedure SSPF::PREFETCH
2:      cur_bid = GetCurBlockId(client_session)
3:      next_binfo = GetNextBlock(cur_bid, file_bmap)
4:      existent = CheckExistInCache(next_binfo.blkid)
5:      if existent is false then
6:          DoPrefetch(next_binfo)
7:      end if
8:      MoveIntoPrefetchQueue(next_binfo)
9:      result = CheckPrefetch(client_session)
10:     if result denotes block is accessed then
11:         continue prefetch
12:     else if result denotes block is skipped then
13:         start new prefetch
14:     else
15:         sleep
16:     end if
17:     return
18: end procedure
```

not been accessed for a long time or the file has been closed, the state of the block is also changed to $accessed$ to free the cache.

C. Cache Manager

Besides prefetching blocks, server-side cache also contains other caching blocks, such as read and write blocks. A straightforward approach is to treat all these cached blocks equally and use common caching schemes (e.g. LRU, LIRS) to manage the cache. However, this "uniform" caching approach cannot distinguish the differences between prefetching blocks and other caching blocks, and may evicted the prefetching blocks before they are accessed. Therefore, it has to fetch data from the disk to the cache again on the next access.

To solve this problem, we design a multi-queue based cache management algorithm to coordinate between the prefetching blocks and other cache blocks, improving the efficiency of cache. As shown in Figure 5, SSPF divides the cache into 3 individual queues : $unaccessed$ queue, $caching$ queue, and $accessed$ queue. The $unaccessed$ queue stores prefetched blocks that have not been accessed yet. The $accessed$ queue stores prefetched blocks that have already been accessed by clients. Both of $accessed$ and $unaccessed$ queue are managed by the FIFO policy which evicts the blocks in the order they were added. Once the prefetched block is accessed, it will be immediately moved from the $unaccessed$ queue to the $accessed$ queue. If a block of $unaccessed$ or $accessed$ queue is accessed by other client request, it will be moved to

caching queue. If the prefetched block are not accessed for a long time, these blocks will be evicted from the *unaccessed* queue by the FIFO policy. *Caching* queue stores popular blocks accessed by the read and write requests, which is managed by existing cache schemes (e.g. LFU, LIRS). When the cache is full, the manager first replaces the cache blocks in the *accessed* queue, then the cache blocks in the *caching* queue, and finally the cache blocks in the *unaccessed* queue. If a block in the *accessed* queue is re-accessed, it will be promoted to the caching queue.

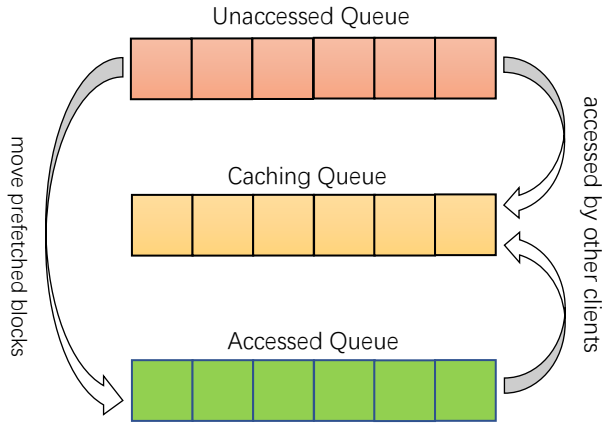

Fig. 5. A multi-queue based cache manager.

A key design of the cache manager is how to allocate cache among the three queues to achieve optimal performance. The size of *unaccessed* queue is determined by the client concurrency and the size of the data block, which is mostly much less than that of the *caching* queue. The length of *accessed* queue is fixed. If the client does not receive the response, it will resend the read request to the data server. In this case, if the prefetched block is still in the cache, it can be directly fetched from *accessed* queue and save a disk I/O. Therefore, the length of the *accessed* queue is calculated based on the retry time of the message to improve the hit ratio of retry requests.

When the cache is too small to store all the prefetched blocks, we introduce a benefit-aware cache reassignment scheme to determine how to allocate cache blocks between *unaccessed* queue and *caching* queue. One approach is to first build accurate model for cache blocks and then solve the issue by formulating an optimization problem to maximize the overall performance. However, this approach depends on heavy computation. Besides, the optimal solution needs to be frequently recalculated to follow the change of access pattern. In contrast, SSPF proposes a lightweight adjustment scheme to solve this issue. The allocation of *unaccessed* queue and *caching* queue is dynamically adjusted by their benefits. The benefit can be defined as the number of saved disk I/O, which can be expressed as follows:

$$\mathrm{E[SavedIO]} = \sum_{i=1}^{n} h_i \times f_i + \sum_{j=1}^{m} h_j \times f_j \qquad (1)$$

where n means the size of *unaccessed* queue, m means the size of *caching* queue, h_i denotes the cache hit ratio of $block_i$, f_i denotes the access frequency of $block_i$. In practice, the cache manager records the cache hit number of each queue for a given time period to reflect its benefits. The cache of low-benefit queue will be periodically reassigned to the high-benefit queue. To reduce the overhead of tracing, the cache manager further divide each queue into multiple regions, called chunk, and only traces and adjusts one chunk per period. For *unaccessed* queue, the cache manager periodically traces the caching hit number of the header chunk and uses it as the benefit of reassignment. For *caching* queue, we use the LRU as an example to explain the adjustment. The LRU policy puts the least recently used blocks into the head of caching queue, so the cache manager always chooses the tail chunk as victim for reassignment. For LRU policy, since the most recently used blocks are placed at the tail of *caching* queue, the cache manager chooses the tail chunk as victim. Therefore, the number of cache hits in the tail chunk is used as the reassignment cost of the caching queue. In order to avoid frequent reassignment, similar to the cache thrashing problem, the cache manager reassigns the cache chunk only when the difference exceeds the threshold. The default threshold is empirically set to 20% and the hit rate should be greater than 5.

D. Load-Aware Request Distribution

Request distribution strategy plays a critical role in the overall system performance. For example, if client distributes two sequential read requests to the same data server, it is possible that the latter request has to wait for prefetching to complete as the slow speed of disk, increasing access latency.Besides, the access latency of read requests on overloaded server is significantly larger than that on the underload server. Therefore, a load-aware distribution strategy not only increases the parallelism between data servers, but also decreases the latency of read requests, thus increasing the overall system throughput.

However, it is challenging to obtain optimal distribution among multiple data server because load balancing problem is known to be NP-hard. Instead, we propose a near-optimal approach to mitigate the performance impact in I/O path, whose goal is to find an acceptable result with negligible overhead. The idea of this approach is to reduce the number of candidate data server in distribution using the following filtering criteria. 1) First, the data servers are classified into two categories : underloaded or overloaded. Next, the strategy selects underloaded data servers as the candidate. However, if all replicas are overloaded, candidates are selected according to the load on data servers. 2) For a sequence of read requests, this strategy only guarantees that the selected data server does not overlap with the n previous data servers, where n should be large enough to ensure that most of prefetching can be completed before read request arrives.

III. Evaluation

We have implemented SSPF in the RacoonFS to verify its performance. RacoonFS is a HDFS-like file system written in C++ for high performance. It has the same architecture as HDFS and also supports write-once-read-many semantics on files. A file once created, written, and closed will then become immutable. We conducted various experiments to evaluate SSPF's performance and memory usage. In this section, we first present the hardware and software environments used in the experiments, and then present and discuss the results of the experiments.

A. Experimental Setup

Figure 6 shows the topology of the SSPF test environment whose system configurations are shown in table II. The cluster consists of five client nodes, three data servers, and one metadata server. The local cache of all client nodes is set to 2GB. All of these nodes are connected by 40Gb/s InfiniBand or 1Gb/s Ethernet. To evaluate the effect of SSPF, we tested the performance of RacoonFS with and without SSPF, where the performance without SSPF was used as a baseline. To evaluate performance with more realistic workloads, we run the Sysbench benchmark [11] on all experiments.

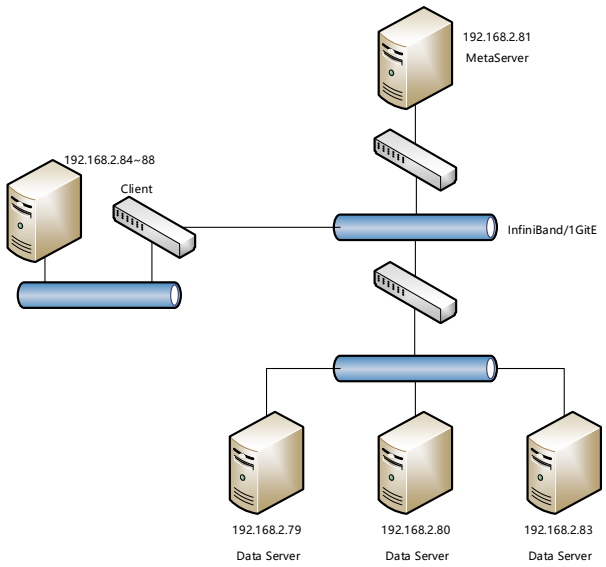

Fig. 6. The topology of the SSPF test environment.

TABLE II
Machine configuration.

CPU	Intel(R) Xeon(R) CPU E5620 @ 2.4GHz*16
Memory	DDR3REG 12GB
Disk	SATA 300GB @ 7200R/M
Operating System	Red Hat Enterprise Linux Server release 5.4
Kernel Version	Linux 2.6.18-163.el5
Network	InfiniBand 40Gb/s 1Gigabit Ethernet

B. Different Network

In this chapter, we deployed SSPF in InfiniBand and Ethernet networks and evaluated its effectiveness in both kind of networks.The bottleneck of the system is the disk in 40Gb/s InfiniBand and the network in 1Gb/s Ethernet respectively.In summary, we tested the following versions of SSPF:

1) IB-prefetch : it runs in the IB network and enables the prefetching scheme;
2) IB-no-prefetch : it runs in the IB network and disables the prefetching scheme;
3) Ethernet-prefetch : it runs in the Ethernet network and enables the prefetching scheme;
4) Ethernet-no-prefetch : it runs in the Ethernet network and disables the prefetching scheme.

1) Read Throughput: In this experiment, files were created with different sizes, whose sizes were 1MB, 50MB, 100MB, 500MB, 1GB, 2GB and 3GB, respectively. The data block size was fixed to 64MB. All experiment used throughput as the performance metrics of the system. The results of read performance are shown in Figure 7. As shown in Figure 7, there is little performance improvement when deploying prefetching scheme for distributed file systems in 1Gb/s Ethernet. The reason is that the throughput of network is less than the throughput of disk in the slow network. Therefore, even if the data is prefetched to the cache before accessed, it still needs to wait for the network transfer to complete. As shown in Figure 7, in IB network, the performance of distributed file system with prefetching scheme had significant improvement compared to that without prefetching scheme. Especially, the throughput of IB-prefetch is increased by up to 41.9% when the file size is 3GB. However, SSPF has very limited performance improvements for small files. It can be seen from Figure 7 that the throughput of IB-prefetch is increased by only 3.89% when the file size is 10MB. The reason is that small file has only one block and will access the data server immediately after sending the prefetching hint, thus causing the read request to wait for the completion of prefetching.

Figure 8 plots the throughput of SSPF varying with the number of data server nodes. In this experiment, the file size was set to 1GB. As shown in Figure 8, the throughput gains of both Ethernet-prefetch and Ethernet-no-prefetch are very small as the number of data servers increase. As we explained above, the bottleneck of a distributed file system in the slow network is the network communication. So, the improvement of disk performance has a negligible impact on the overall performance. In IB network, for both IB-prefetch and IB-no-prefetch, throughput of RacoonFS increases with the number of data servers. As the number of data server increases, the throughput of RacoonFS is improved by 2.87%∽8.05% in IB-prefetch and 2.44%∽6.03% in IB-no-prefetch. The reason is that the access parallelism increases as the number of data server nodes increases, reducing the overhead of disk.

2) Write Throughput: In this experiment, it tested the write throughput of SSPF. Files were created with different sizes, whose sizes were 1MB, 300MB, 400MB, 500MB, 800MB, 1GB,

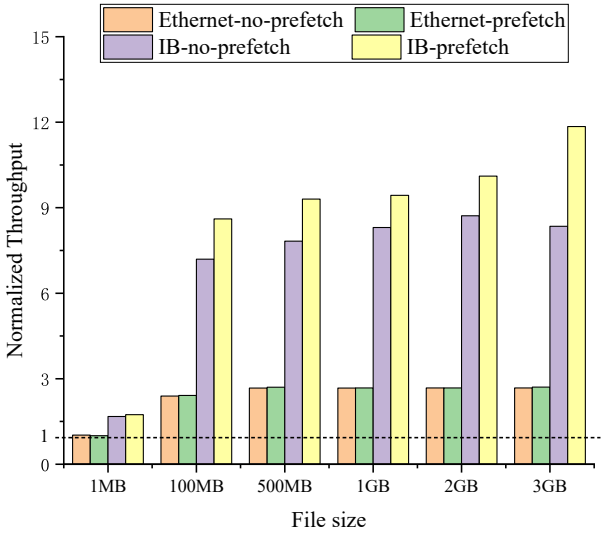

Fig. 7. The read throughput of the SSPF.

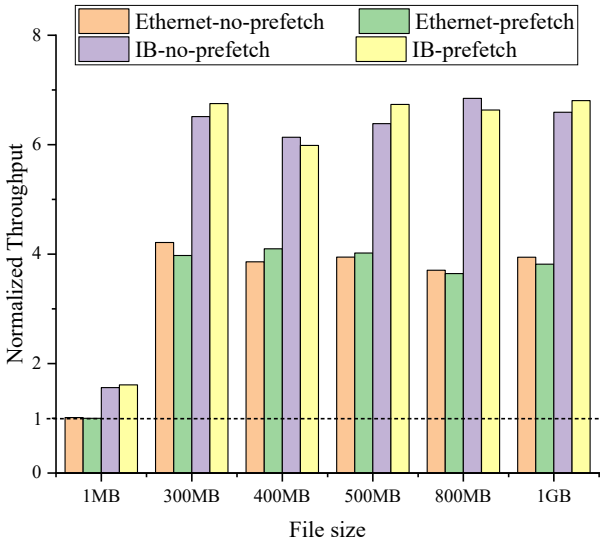

Fig. 8. The throughput of RacoonFS varying with the number of data server nodes.

respectively. The data block size was fixed to 64MB. The results of the write throughput are shown in Figure 9. As shown in Figure 9, the throughput gains for prefetching scheme are in the range of -5.64%∽6.13% in 1Gb/s Ethernet and the average is -0.63%. The throughput gains for prefetching scheme are in the rage of -3.1%∽5.47% in the 40Gb/s IB and the average is 1.66%. It can find that the write overhead introduced by the prefetching scheme is negligible. The reason is that SSPF introduces a novel cache manager to coordinate the prefetching blocks and caching blocks, improving the efficiency of overall cache.

C. Different Workloads

In this chapter, we evaluated the throughput of SSPF under different type of read workload. We simulated concurrent

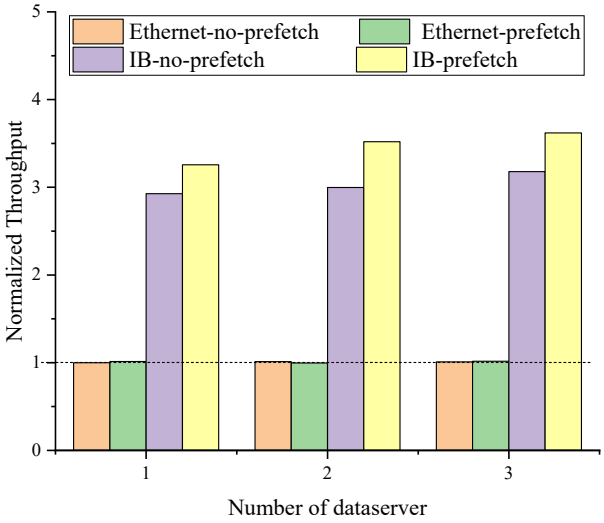

Fig. 9. The write throughput of the SSPF.

read by varying the number of threads in sysbench Files were created with diffrent sizes, whose sizes were 1GB/2GB/3GB respectively.

1) Sequential Read: Table III shows the results of the sequential read experiment. The experimental results show that when the number of threads was greater than 1, the performance improvement of SSPF is higher than when the number of threads is 1. When the number of thread was set 1, the average performance improvement is only 0.96% with the prefetching scheme. When the number of thread was set 2/4/8, the average performance improvement achieves 9.13%/8.88%/12.03% respectively with the prefetching scheme. Parallel file reading can reduce performance due to the interference, such as increasing the randomness of I/O access, reducing the server cache efficiency of the file system, and increasing the overhead of system calls. SSPF prefetchs files to the client node and then directly accesses these files, thus alleviating performance degradation caused by interference on the file system server.

TABLE III
SEQUENTIAL READ THROUGHPUT VARYING THREAD NUMBER AND FILE SIZE.

File Size	Prefetch	The Number of Threads			
		1(MB/s)	2(MB/s)	4(MB/s)	8(MB/s)
1GB	Enable	130.81	139.06	137.34	149.45
1GB	Disable	129.04	128.36	126.52	126.35
2GB	Enable	130.38	140.81	139.32	138.45
2GB	Disable	128.91	128.1	126.98	126.16
4GB	Enable	129.75	139.88	137.37	135.78
4GB	Disable	129.27	128.16	126.77	125.64

2) Random Read: Table IV shows the results of the random read experiment. The experimental results show that when the thread number was set to 1, the performance improvement of SSPF was higher than any other thread number settings. When the number of thread was set 1, the average performance improvement achieves 11.16% with

the prefetching scheme. When the number of thread was larger than 1, the average performance improvement is only 3.74% with the prefetching scheme. Because the prefetch accuracy of random reads is generally lower than that of sequential reads, the lower prefetch accuracy will reduce the effectiveness of SSPF. Especially, when serving multiple threads, the accuracy of prefetching will be further reduced due to interference between these threads.

TABLE IV
RANDOM READ THROUGHPUT VARYING THREAD NUMBER AND FILE SIZE.

File Size	Prefetch	The Number of Threads			
		1(MB/s)	2(MB/s)	4(MB/s)	8(MB/s)
1GB	Enable	9.89	19.16	38.79	80.2
1GB	Disable	9.27	18.77	38.21	79.1
2GB	Enable	9.54	17.83	36.37	73.14
2GB	Disable	8.34	16.91	34.02	68.88
4GB	Enable	8.61	16.06	32.99	66.04
4GB	Disable	7.66	15.77	31.63	63.48

D. Cache Usage

In this experiment, we tested the cache usage of SSPF in RacoonFS. Files were created with different sizes, whose sizes were 100MB, 200MB, 500MB, 1GB, 2GB, 3GB, respectively. The data block size was fixed to 64MB. The experiments used the Linux's top command to collect the memory usage and collected it every second. It first collected the average memory usage with deploying SSPF and without deploying SSPF. Then we got the memory overhead of SSPF by subtracting these two values. Table V records the memory overhead of IB-prefetch in each data server when reading different file sizes. As shown in table V, the memory overhead of SSPF is very small. The reasons are as follows: 1) The first is because SSPF prefetches only the next data block of the file, thus reducing the requirement of the prefetching cache; 2) The second is because the entire cache is shared by prefetching and caching blocks and its allocation is adjusted by their efficiency. 3) The third is because the use of cache pooling technology, which allocates cache from the pool and releases them to the pool, avoiding memory fragmentation.

TABLE V
THE MEMORY OVERHEAD OF IB-PREFETCH.

	100MB	200MB	500MB	1GB	2GB	3GB
DataServer1	0.45%	0.56%	0.98%	0.81%	0.23%	1.01%
DataServer2	0.87%	0.32%	0.47%	0.35%	0.53%	0.96%
DataServer3	0.39%	0.62%	0.78%	0.42%	0.92%	0.62%

IV. RELATED WORK

Prefetching is widely used in distributed file systems because it can hide the latency caused by network communication and disk operations. Some researches [12]–[15] have investigated the benefits of local cache, which prefetches remote data to local cache to improve I/O performance for distributed file system. When an I/O request hits in the cache, the data can be fetched directly from the local cache, reducing access latency. In these client-side prefetching schemes, the client is responsible for predicting future accesses by storing and analyzing the history of I/O access requests. Griffioen et al. proposes Automatic Prefetching [15], which takes a heuristic-based approach using knowledge of past file accesses to predict future access requests. Similar as Automatic Prefetching, Kroeger et al. build prediction model based on the sequences of historical file access requests and use this model to predict the next request of the file [12]. FARMER [13] mines file correlations from several typical file system traces and uses it to improve the performance of prefetching scheme. However, as the prediction requires considerable amount of storage and computational power, the client-side prefetching scheme is not well suitable for client machines with limited memory and computing power.

A requirement of client-side caching is that it must guarantee the consistency of the cached data for ensuring all relevant clients have a consistent view of data. However, the overhead of guaranteeing data consistency is so high that it may offset the performance benefits of data caching. There is a lot of researches on how to achieve data consistency with low overhead [16]–[18]. The research [16] uses a write-through approach to ensure data consistency. However, the performance penalty of this approach is so large that it only adapts to the Write-Once-Read-Many workload. GPFS [19] uses the token, which is first revoked when modifying the data, to maintain cache consistency between clients. Similarly, Lustre [20] uses a distributed lock manager, named DLM, to guarantee cache consistency. LPCC [3] proposes an SSD-based hierarchical persistent client cache for Lustre. It integrates with the Lustre Hierarchical Storage Management and the Lustre DLM to guarantee consistent persistent caching. The work [4] further applies hierarchical persistent client cache to NVM. In Ceph [21], when a client updates a file, other clients will be notified to invalidate it's cached copy.

However, the limited power of computing and memory of client nodes make it sometimes impracticable to do prefetching. To solve this problem, several researches [22]–[25] have proposed server-side prefetching schemes. Padmanabhan and Mogul [22] investigated a prefethcing scheme for reducing the user's latency by predicting and prefetching files that are likely to be requested soon in servers. Since the computing node in the cloud always has limited memory resources, chen et al. propose a scheduling-aware data prefetching scheme in the server [23], [26], which prefetches data and release memory resources based on the scheduling information. Similarly, li et al. propose a scheduling-aware data prefetching scheme to enhance the data locality [27]. In the work [28], the storage server first models disk I/O access operations and then prefetches data directly after predicting future I/O accesses. However, the model requires extra space to store logs and its correctness is affected by dynamic changes in workload.

V. Conclusion

We present SSPF, a novel server-side prefetching scheme for distributed file systems. The goal of SSPF is to hide the latency in distributed file systems caused by network communication and disk operations by aggressive prefetching. The core ideas include (1) To minimize the memory requirement, SSPF prefetches only the next block into cache with the help of access pattern hint; (2) A multi-queue based cache manager is proposed to coordinate between the prefetching blocks and other caching blocks; (3) A heuristic-based request distribution strategy is proposed to optimize the balance among data server nodes to improve the overall throughput.

We have implemented and evaluated SSPF on the real distributed file system. Our evaluation results shows that SSPF has 41.9% better read throughput in the fast network, such as 40Gb/s InfiniBand. Since the bottleneck of distributed file systems in the slow network (e.g., 1Gb/s Ethernet) is network communication, it found that the performance gain of the prefetching scheme is very limited in the slow network.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments. This work is supported in part by the Zhejiang Provincial Leadership-type Innovation and Entrepreneurship Team (2024R01006), the National Natural Science Foundation of China (U22A6001), the "Pioneer" and "Leading Goose" R&D Program of Zhejiang under Grant 2024SSYS0015, the National Key R&D Program of China (2023YFB4503005), the National Science Foundation of China (62302465).

References

[1] D. Reinsel, J. Gantz, and J. Rydning, *The digitization of the world from edge to core*. IDC iView: IDC Analyze the future, 2018.

[2] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[3] Y. Qian, X. Li, S. Ihara, A. Dilger, C. Thomaz, S. Wang, W. Cheng, C. Li, L. Zeng, F. Wang *et al.*, "Lpcc: Hierarchical persistent client caching for lustre," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[4] W. Cheng, C. Li, L. Zeng, Y. Qian, X. Li, and A. Brinkmann, "Nvmm-oriented hierarchical persistent client caching for lustre," *ACM Transactions on Storage (TOS)*, vol. 17, no. 1, pp. 1–22, 2021.

[5] Y. Qian, M.-A. Vef, P. Farrell, A. Dilger, X. Li, S. Ihara, Y. Fu, W. Xue, and A. Brinkmann, "Combining buffered {I/O} and direct {I/O} in distributed file systems," in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, 2024, pp. 17–33.

[6] Z. Lin, L. Xiang, J. Rao, and H. Lu, "P2cache: Exploring tiered memory for in-kernel file systems caching," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 801–815.

[7] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "Distcache: Provable load balancing for large-scale storage systems with distributed caching," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, 2019, pp. 143–157.

[8] X. He, B. Yang, J. Gao, W. Xiao, Q. Chen, S. Shi, D. Chen, W. Liu, W. Xue, and Z.-n. Chen, "Hadafs: A file system bridging the local and shared burst buffer for exascale supercomputers," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 215–230.

[9] D. Koo, J. Lee, J. Liu, E.-K. Byun, J.-H. Kwak, G. K. Lockwood, S. Hwang, K. Antypas, K. Wu, and H. Eom, "An empirical study of i/o separation for burst buffers in hpc systems," *Journal of Parallel and Distributed Computing*, vol. 148, pp. 96–108, 2021.

[10] E. Karrels, L. Huang, Y. Kan, I. Arora, Y. Wang, D. S. Katz, W. Gropp, and Z. Zhang, "Fine-grained policy-driven i/o sharing for burst buffers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–12.

[11] sysbench. [Online]. Available: https://github.com/akopytov/sysbench

[12] T. M. Kroeger and D. D. E. Long, "Predicting file system actions from prior events," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '96. USA: USENIX Association, 1996, p. 26.

[13] P. Xia, D. Feng, H. Jiang, L. Tian, and F. Wang, "Farmer: a novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance," in *Proceedings of the 17th international symposium on High performance distributed computing*, 2008, pp. 185–196.

[14] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris, "Flexible, wide-area storage for distributed systems with wheelfs," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, 2009, pp. 43–58.

[15] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference-Volume 1*, 1994, pp. 13–13.

[16] S. Quinlan, "A cached worm file system," *Software: Practice and Experience*, vol. 21, no. 12, pp. 1289–1299, 1991.

[17] N. R. Paiker, J. Shan, C. Borcea, N. Gehani, R. Curtmola, and X. Ding, "Design and implementation of an overlay file system for cloud-assisted mobile apps," *IEEE Transactions on Cloud Computing*, vol. 8, no. 1, pp. 97–111, 2017.

[18] M. J. Franklin, M. J. Carey, and M. Livny, "Transactional client-server cache consistency: Alternatives and performance," *ACM Transactions on Database Systems (TODS)*, vol. 22, no. 3, pp. 315–363, 1997.

[19] T. Jones, A. Koniges, and R. K. Yates, "Performance of the ibm general parallel file system," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000.* IEEE, 2000, pp. 673–681.

[20] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang, "Understanding lustre filesystem internals," Oak Ridge Nat. Lab., Oak Ridge, TN, USA, Tech. Rep ORNL/TM-2009/117, 2009.

[21] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[22] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 3, pp. 22–36, 1996.

[23] C.-H. Chen, T.-Y. Hsia, Y. Huang, and S.-Y. Kuo, "Data prefetching and eviction mechanisms of in-memory storage systems based on scheduling for big data processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1738–1752, 2019.

[24] T. Luo, V. Aggarwal, and B. Peleato, "Coded caching with distributed storage," *IEEE Transactions on Information Theory*, vol. 65, no. 12, pp. 7742–7755, 2019.

[25] S. Wang, Z. Lu, Q. Cao, H. Jiang, J. Yao, Y. Dong, and P. Yang, "Bcw: Buffer-controlled writes to hdds for ssd-hdd hybrid storage server," in *18th USENIX Conference on File and Storage Technologies*, 2020, p. 253.

[26] C.-H. Chen, T.-Y. Hsia, Y. Huang, and S.-Y. Kuo, "Scheduling-aware data prefetching for data processing services in cloud," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2017, pp. 835–842.

[27] C. Li, J. Zhang, Y. Chen, and Y. Luo, "Data prefetching and file synchronizing for performance optimization in hadoop-based hybrid cloud," *Journal of Systems and Software*, vol. 151, pp. 133–149, 2019.

[28] J. Liao, F. Trahay, G. Xiao, L. Li, and Y. Ishikawa, "Performing initiative data prefetching in distributed file systems for cloud computing," *IEEE Transactions on cloud computing*, vol. 5, no. 3, pp. 550–562, 2015.