SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

March 2022, Vol. 65 132101:1–132101:17
https://doi.org/10.1007/s11432-019-2833-1

A self-tuning client-side metadata prefetching scheme for wide area network file systems

Bing WEI[1,2], Limin XIAO[1,2*], Yao SONG[1,2], Guangjun QIN[3], Jinbin ZHU[1,2], Baicheng YAN[1,2], Chaobo WANG[1,2] & Zhisheng HUO[1,2]

[1]*Laboratory of Software Development Environment, Beihang University, Beijing* 100191*, China;*
[2]*School of Computer Science and Engineering, Beihang University, Beijing* 100191*, China;*
[3]*Smart City College, Beijing Union University, Beijing* 100101*, China*

Received 28 September 2019/Revised 14 December 2019/Accepted 17 March 2020/Published online 22 February 2021

Abstract Client-side metadata prefetching is commonly used in wide area network (WAN) file systems because it can effectively hide network latency. However, most existing prefetching approaches do not meet the various prefetching requirements of multiple workloads. They are usually optimized for only one specific workload and have no or harmful effects on other workloads. In this paper, we present a new self-tuning client-side metadata prefetching scheme that uses two different prefetching strategies and dynamically adapts to workload changes. It uses a directory-directed prefetching strategy to prefetch the related file metadata in the same directory, and a correlation-directed prefetching strategy to prefetch the related file metadata accessed across directories. A novel self-tuning mechanism is proposed to efficiently convert the prefetching strategy between directory-directed and correlation-directed prefetching. Experimental results using real system traces show that the hit ratio of the client-side cache can be significantly improved by our self-tuning client-side prefetching. With regards to the multi-workload concurrency scenario, our approach improves the hit ratios for the no-prefetching, directory-directed prefetching, variant probability graph algorithm, variant apriori algorithm, and variant semantic distance algorithm by up to 15.22%, 6.32%, 10.08%, 11.65%, and 10.73%, corresponding to 25.24%, 18.11%, 23.53%, 24.94%, and 24.19% reductions in the average access time, respectively.

Keywords wide area network file systems, multiple workloads, metadata prefetching, correlation-directed prefetching, directory-directed prefetching, self-tuning prefetching

Citation Wei B, Xiao L M, Song Y, et al. A self-tuning client-side metadata prefetching scheme for wide area network file systems. Sci China Inf Sci, 2022, 65(3): 132101, https://doi.org/10.1007/s11432-019-2833-1

1 Introduction

In a wide area environment, heterogeneous storage resources owned by different organizations are geographically distributed, resulting in barriers between applications and data. Network-based file systems offer promising solutions to address this problem. In network-based file systems (such as Onedata [1] and GFFS [2]), the client and server are decoupled and interact with each other through network communications. Several network-based file systems use client-side metadata caching to reduce the number of network communications and achieve better access performance [1–5]. The client caches a certain amount of metadata and periodically refreshes the cached metadata [1].

As cache hit ratios are crucial for the performance of network-based file systems [6], several prefetching schemes [3–5, 7–11] have been proposed to improve cache hit ratios. These approaches can generally be classified into two categories: directory-directed or correlation-directed prefetching. Directory-directed prefetching is commonly used to alleviate access latency in several network-based storage systems [3–5]. It prefetches all file metadata in the same directory with a network communication. This type of approach can be used to prefetch metadata without knowing the semantic correlations between files [11]. Directory-directed prefetching is effective because it can capture the natural organization imposed by

* Corresponding author (email: xiaolm@buaa.edu.cn)

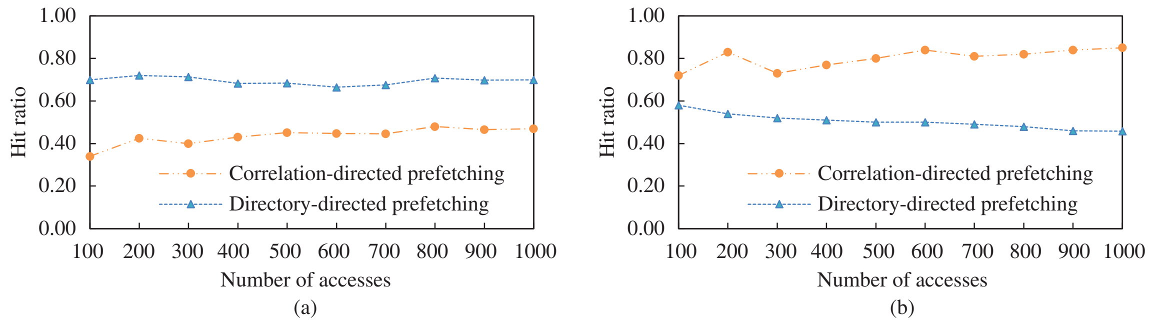


Figure 1 (Color online) Variations of cache hit ratios for metadata fetching when running two different application workloads. (a) Financial1 Trace; (b) Exchange Server Traces.

users [11]. Correlation-directed prefetching is a promising approach that exploits increasing intelligence in file systems [5,11,12]. File correlations are useful for improving the effectiveness of client-side caching [7, 11]. For example, the hyperlink of html file B is embedded in html file A, and B has a high probability of being accessed once A is accessed. Therefore, we can cache A and B together to improve the cache hit ratio. As another example, consider a C program that is built from the source file HelloWorld.c, and the source file refers to the head file stdio.h. Clearly, stdio.h is likely to be accessed once HelloWorld.c is accessed. The prefetching module is usually configurable in storage systems, and a prefetching strategy can be loaded as needed before running an application [3–5].

Network-based file systems are expected to serve applications from various fields, and these applications have different workload characteristics, resulting in different requirements for metadata prefetching [13]. Existing prefetching approaches [3–5, 7–11] typically use a unified prefetching strategy to serve I/O requests issued from different applications. This "one-size-fits-all" solution cannot meet the varied prefetching requirements, and the maximum file I/O bandwidth cannot be obtained [13]. For instance, directory-directed prefetching yields a higher cache hit ratio than the correlation-directed prefetching strategy in terms of metadata fetching when replaying Financial1 Trace (see UMass Trace Repository), as shown in Figure 1(a). This is because the directory-directed prefetching strategy can capture the natural organization imposed by users [11]. Even though the files accessed consecutively under this workload are usually in the same directory, the correlation between any two files in the same directory is not close. The files placed in the same directory seem to be read randomly. Correlation-directed prefetching either lacks sufficient file association information to prefetch metadata or the prefetched metadata is incorrect, thereby failing to achieve the same hit-ratio level as directory-directed prefetching. Correlation-directed prefetching achieves a higher cache hit ratio when replaying Exchange Server Trace (see Storage Networking Industry Association), as shown in Figure 1(b). This occurs because there is a close relationship between files across directories for this workload, and correlation-directed prefetching can effectively capture file association information to improve the prefetching accuracy [5, 7, 11]. For this workload, the files placed in the same directory are not accessed relatively close to each other. Directory-directed prefetching is static rather than dynamic; thus, it cannot fully exploit the complex semantic patterns between files in storage systems [7, 11]. We also find that directory-directed prefetching incurs cache thrashing. This is because useless metadata are prefetched, which then pollutes the client-side cache [14].

The test results show that different application workloads have different prefetching requirements. Furthermore, I/O access patterns are more complex when multiple applications with different workload characteristics are run simultaneously [13]. The "one-size-fits-all" solution cannot address this problem [13, 15–17]. In this paper, to address this problem, we present a new self-tuning client-side metadata prefetching scheme that uses multiple different prefetching strategies and dynamically adapts to workload changes. It uses a directory-directed prefetching strategy to prefetch the related file metadata in the same directory and correlation-directed prefetching strategy to prefetch metadata that are frequently accessed together across directories. The proposed file mining strategies are modified as our correlation-directed prefetching strategies to explore file correlations in several storage traces collected in real storage systems. As a result, the proper prefetching strategy can be selected to serve varied I/O requests to improve cache hit ratios. The main contributions of this paper are summarized as follows.

- A new prefetching approach is proposed to adaptively adjust the prefetching strategy based on the variations of access patterns for wide-area file systems. It tracks the cache hit ratios of each prefetching

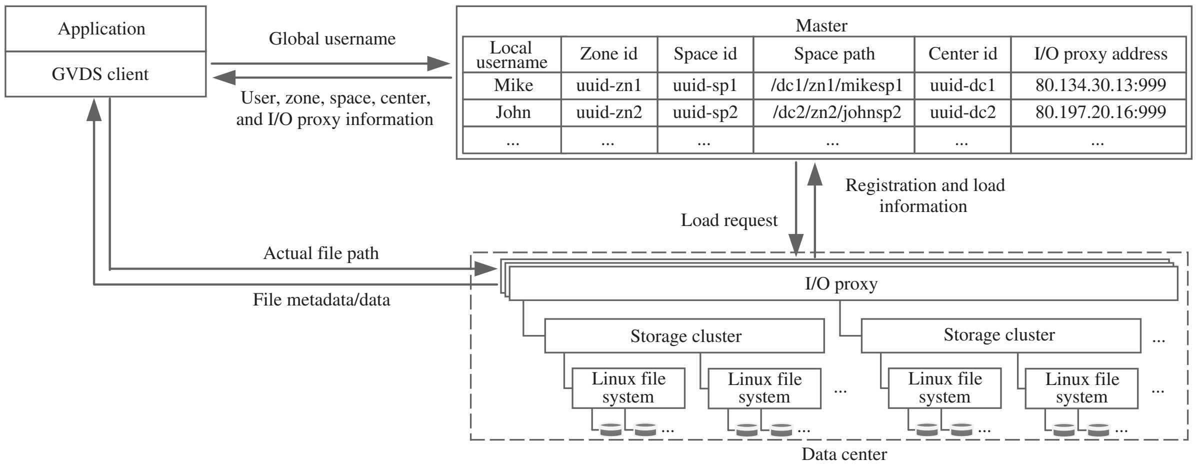

Application
GVDS client
Global username
User, zone, space, center, and I/O proxy information
Master
Local username | Zone id | Space id | Space path | Center id | I/O proxy address
Mike | uuid-zn1 | uuid-sp1 | /dc1/zn1/mikesp1 | uuid-dc1 | 80.134.30.13:999
John | uuid-zn2 | uuid-sp2 | /dc2/zn2/johnsp2 | uuid-dc2 | 80.197.20.16:999
Load request
Registration and load information
Actual file path
File metadata/data
I/O proxy
Storage cluster
Linux file system
Data center

Figure 2 GVDS architecture.

strategy in the current sliding window, and applies the prefetching strategy with a higher hit ratio to the next sliding window. It can meet the requirement of fine-grained control on prefetching strategies to apply the correct prefetching strategy when running different concurrent applications.

• To reduce the overhead introduced by the self-tuning prefetching approach, in which the advantages and disadvantages of multiple prefetching strategies need to be evaluated in real time, we propose the following three mechanisms: (1) asynchronous fetching, (2) data compression, and (3) calculating the hit ratio based on assumptions.

• We implemented our self-tuning prefetching strategy on our newly designed wide-area file system and evaluated the benefits of selecting the correct prefetching strategy at runtime. We evaluated the benefits of self-tuning prefetching by replaying the real system traces. The experimental results show that our proposed scheme can effectively improve the hit ratio of the client-side cache in both the single and multi-workload concurrency scenarios.

The remainder of this paper is organized as follows. Section 2 describes the background and motivation. Section 3 describes our design for metadata prefetching. Section 4 presents the implementation of our self-tuning prefetching method. Section 5 presents and discusses the experimental results. Section 6 presents the related work. Section 7 provides conclusion and future work.

2 Background and motivation

To achieve unified access to geographically distributed data, we designed a global integrated file system referred to as global virtual data space (GVDS), which is under development and will be described in detail in a future study. GVDS is a wide area network (WAN) file system that federates heterogeneous file systems (such as distributed/parallel and local file systems). It virtualizes globally distributed storage resources to construct a unified storage space.

GVDS consists of a single manager node, multiple I/O servers, and multiple clients, as shown in Figure 2. To hide the complexity of the data distribution in GVDS, we introduce the concept of spaces. Spaces are logical containers that hold data stored by associated users. A space corresponds to a directory that is located in a file system. The manager node manages information regarding the spaces and I/O servers. The I/O server provides the file access for distributed/parallel and local file systems. For the purpose of load balance, an institute usually deploys multiple I/O servers. Every I/O server is able to mount one or multiple distributed/parallel and local file systems to access files. Sometimes, hot data may need to be accessed quickly, and an I/O server with a large memory capacity can be deployed to cache all of the data to avoid disk operations for low access latency. Our proposed approach is mainly applicable to this scenario. When accessing a file, the client connects to the manager node and requests the space path and I/O server address. The space and relative file paths are spliced by the client to form an absolute file path. Client-specific lookup routines are invoked when the virtual file system (VFS) layer performs a path lookup.

We found that metadata fetching may incur severe I/O overheads, especially when the file size is small

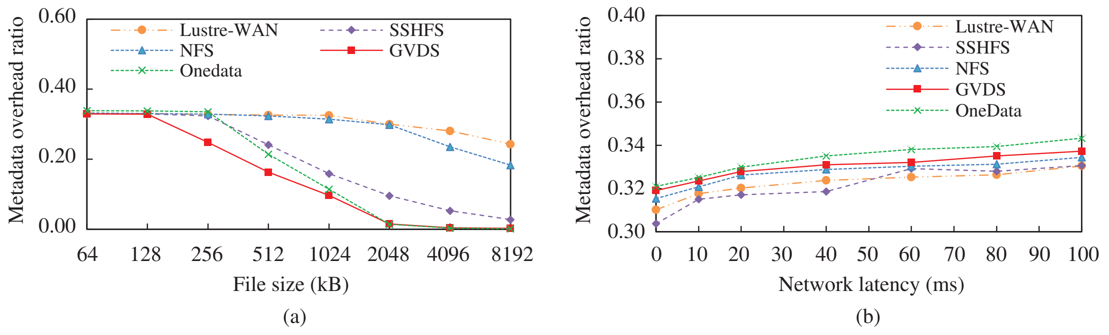
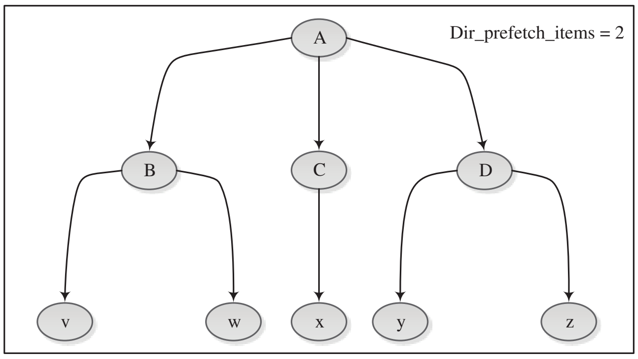


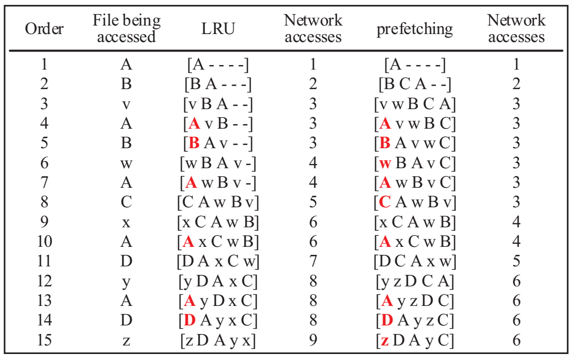
Figure 3 (Color online) Execution states: functional state for single and self-tuning prefetching-schemes. Metadata overhead ratios of the different network-based file systems when varying (a) the file size with a network latency under 60 ms and (b) the network latency with a file size under 128 kB.

and network latency is high. Figure 3 shows the metadata overhead ratios of Lustre-WAN [18], SSHFS (see Github), NFS (see Wikipedia), GVDS, and Onedata [1]. We used fio (see Wikipedia) to generate the desired types of I/O actions to evaluate the impact of file size and network latency on the metadata overhead ratio (ratio of the metadata access time and total access time). We used the traffic control (TC) tool to control the network latency. Figure 3(a) shows the metadata overhead ratios of the different network-based file systems when varying the file size with a network latency under 60 ms. The metadata access overhead ratio is relatively high when the file size is small. This behavior occurs because each metadata access introduces a high-latency network communication, and the data transferred by WAN is too small to offset the overhead of the metadata fetching. The metadata overhead ratios of all systems dropped dramatically as the block size was increased from 64 to 8192 kB. Figure 3(b) shows the metadata overhead ratios of the different network-based file systems when varying the network latency with a file size under 128 kB. The metadata overhead ratios of all systems increased to some extent, as the network latency was increased from 0 to 100 ms. This is because the metadata/data access time is comprised of the network and disk time. Data and metadata access have the same network time. The higher the network latency, the higher the network time proportion, and the higher the metadata overhead ratio. Furthermore, numerous studies have shown that small files receive the majority of file references and several workloads contain numerous small files with an average size under 1024 kB or less [19]. It has been shown that more than 80% of files being accessed are less than 32 bytes and approximately 40% of the small file access time can be attributed to metadata fetching [5]. Thus, a better access performance can be achieved by reducing the metadata fetching overhead.

To reduce metadata fetching overhead, WAN file systems, such as OneData [1] and GFFS [2], cache a certain amount of metadata on the client side to reduce network and I/O overheads. Client-side cache accuracy is of critical importance in a wide area environment because a cache miss leads to a low-bandwidth and high-cost network communication [11]. The WAN file systems mentioned above use the least recently used (LRU) algorithm for cache replacement. However, LRU is insufficient to guarantee availability when facing multiple workloads [11]. Several prefetching approaches [3–5, 11, 12] have been proposed to improve cache effectiveness in file systems. These approaches can be classified as directory-directed prefetching [3–5] and correlation-directed prefetching [5, 11, 12]. Most of these only provide applications with a single prefetching strategy regardless of the variations in the access patterns. Thus, the existing prefetching approaches in file systems adopt the “one-size-fits-all” solution to provide services for all applications. However, file systems are expected to serve applications in various fields [13]. This “one-size-fits-all” solution fails to meet various application needs and hinders the full exploitation of the potential performance [13]. Zhang et al. [5] used both directory-directed and correlation-directed prefetching to improve the overall system performance based on the access patterns of workloads. However, these prefetching strategies are configured by the users prior to running applications, rather than adaptively selected by the system. Unfortunately, it is not an easy task for users to select the correct strategy to exploit the potential performance of the file systems. One primary reason is the narrow I/O interface between applications and file systems. In such a simple interface, applications perform only file read or write operations without any indication of access patterns or data semantics [7]. Another reason is that once the user selects a prefetching strategy, there is still only one prefetching strategy throughout the running of the applications. Thus, the “one-size-fits-all” problem still exists.



(a)



Order	File being accessed	LRU	Network accesses	prefetching	Network accesses
1	A	[A - - - -]	1	[A - - - -]	1
2	B	[B A - - -]	2	[B C A - -]	2
3	v	[v B A - -]	3	[v w B C A]	3
4	A	[**A** v B - -]	3	[**A** v w B C]	3
5	B	[**B** A v - -]	3	[**B** A v w C]	3
6	w	[w B A v -]	4	[**w** B A v C]	3
7	A	[**A** w B v -]	4	[**A** w B v C]	3
8	C	[C A w B v]	5	[**C** A w B v]	3
9	x	[x C A w B]	6	[x C A w B]	4
10	A	[**A** x C w B]	6	[**A** x C w B]	4
11	D	[D A x C w]	7	[D C A x w]	5
12	y	[y D A x C]	8	[y z D C A]	6
13	A	[**A** y D x C]	8	[**A** y z D C]	6
14	D	[**D** A y x C]	8	[**D** A y z C]	6
15	z	[z D A y x]	9	[**z** D A y C]	6

(b)

Figure 4 (Color online) Case study of directory-directed prefetching.

The last reason is that as the interrelationships between files and the increasing intelligence in storage systems are highly complex, users cannot discover complex patterns on modern file systems to select a correct prefetching strategy [11]. Therefore, selecting prefetching strategies by users does not address the "one-size-fits-all" problem. This motivates our proposal of a more powerful prefetching approach to address this problem by exploiting the potential performance in GVDS.

3 Design of metadata prefetching

In this section, we introduce directory-directed prefetching, present the correlation-directed prefetching used in our scheme, and describe the self-tuning prefetching for efficient prefetching in WAN file systems.

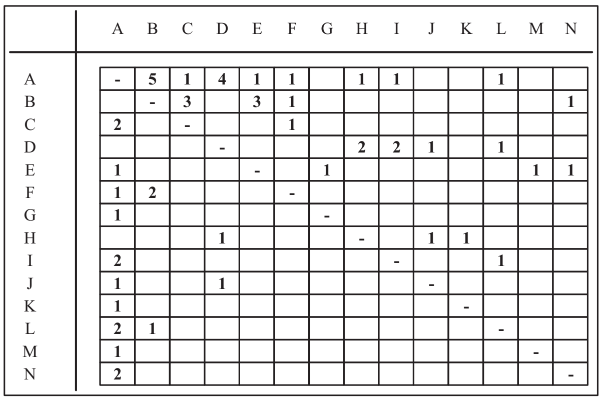
3.1 Directory-directed prefetching

Metadata requires fetching prior to file data accessing whenever a client accesses a file. If the client-side metadata cache misses, the client needs to connect the remote I/O server to fetch metadata. The I/O server returns the currently requested file metadata and metadata of all files (including directories) in the same directory. If the number of files in a directory is significantly large, aggressive prefetching occurs. The file prefetching number threshold (dir_prefetch_items) needs to be set. The client caches the metadata that is returned from the I/O server and uses the LRU for cache replacement. Prefetched metadata are compressed before sending. The compression mechanism is described in detail below.

Directory-directed prefetching can reduce the number of network accesses. Let us consider an example in which a client opens the files A/B/v, A/B/w, A/C/x, A/D/y, and A/D/z. The directory structure is shown in Figure 4(a) for dir_prefetch_items = 2. The client-specific lookup routines are invoked when the VFS layer performs path lookup. For file A/B/v, the metadata of A, B, and v are fetched sequentially. A similar method is performed when accessing other files. Figure 4(b) shows that the cache with directory-directed prefetching is more effective than the traditional cache without prefetching (LRU replacement algorithm), where the cache size is 5 metadata items. The cache hit ratios for the no-prefetching and directory-directed prefetching are 40% and 60%, respectively. Directory-directed prefetching reduces 3 network accesses compared to no-prefetching. Thus, directory-directed prefetching can significantly improve the cache hit ratio without tracking and analyzing file references. However, it cannot identify related files accessed across directories.

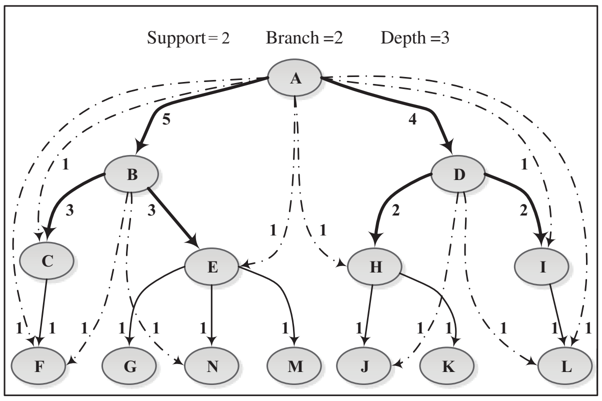
3.2 Correlation-directed prefetching

The correlation-based approach is used to infer file correlations inside file systems. It dynamically analyzes the access stream to infer file correlations without the knowledge of users and any assumptions regarding applications. This type of approach infers file correlations based on the fact that files that are always accessed together within a short access distance are likely to be accessed together in the future. To investigate correlation-based prefetching, we proposes variants of the probability graph [12] and a semantic distance [11] algorithms.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
A	-	5	1	4	1	1		1	1			1		
B		-	3		3	1								1
C	2		-			1								
D				-				2	2	1		1		
E	1				-		1						1	1
F	1	2				-								
G	1						-							
H				1				-		1	1			
I	2								-			1		
J	1			1						-				
K	1										-			
L	2	1										-		
M	1												-	
N	2													-

(a)



(b)

Figure 5 Case study of prefetching based on a variant probability graph. (a) Variant probability graph example; (b) prefetching tree constructed by the variant probability graph.

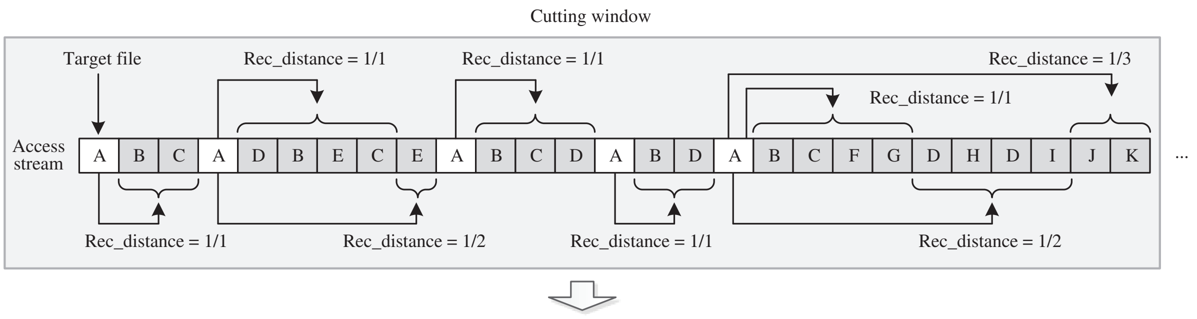
3.2.1 *Prefetching based on the variant probability graph algorithm*

In network-based file systems, providing data in advance of the request can effectively hide access latency and improve system performance. A probability graph [12] is used to capture file correlations for prefetching. We used a probability graph to find the correlated files. The probability graph is a directed graph, which consists of vertices and arcs. Each vertex represents a unique file, and each arc represents an access relationship between two files. File B is said to be related to file A if the access of file B is next to the access of file A; therefore, file B is likely to be accessed after file A is accessed. Each time the sequence AB appears in an access stream, the weight of arc A$\rightarrow$B is incremented by 1. The larger the weight of the arc, the more relevant the files. The parameter support determines whether any two files are relatively close to each other. File B can be prefetched only if the weight of arc A$\rightarrow$B is greater than or equal to the support when accessing file A.

Figure 5(a) shows the construction of a probability graph based on the following access stream: ABCFABEGABENABEMAEADHJADHKADILADIABCACAFBFBNAHDJDLAIALBC. The graph is traversed to construct a prefetching tree, which consists of the currently requested and prefetched files. Each node of the tree is a file, and the currently requested file is the root of the tree. The branch and depth parameters are the threshold values for the prefetching breadth and depth, respectively. The branch specifies the maximum prefetching number of files related to an individual file. The prefetching priority is proportional to the weight of the arc. The depth specifies the maximum path length from the root to the leaf node. Figure 5(b) shows an example of how a prefetching tree is constructed using the probability graph, where support = 2, branch = 2, and depth = 3. The number on an edge is the support value for the corresponding correlation between two files. The dashed lines represent the correlation between a node and its descendants other than its children. The highlighted lines are the correlations with support $\geqslant 2$. The number of highlighted lines originating from a parent node is no more than the branch. For example, if A is currently requested by the client, the two arcs with the highest weight for A are arcs A$\rightarrow$B and A$\rightarrow$D. Therefore, B and D become the children of A. The weights for arcs B$\rightarrow$C and B$\rightarrow$E are larger than the support; hence, C and E become the children of B. The weights for arcs D$\rightarrow$H and D$\rightarrow$I are equal to the support; hence, H and I become the children of D. As a result, the nodes linked by the highlighted lines eventually become a subtree with a branch of 2 and a depth of 3. Some arcs, such as N$\rightarrow$A, M$\rightarrow$A, L$\rightarrow$A, and L$\rightarrow$B, are not plotted as the edges because the target nodes A and B have already been fetched.

3.2.2 *Prefetching based on the variant semantic distance algorithm*

In our exploration of correlation-based prefetching, we also used a variant of the semantic distance algorithm [11]. The key observation is that if a set of files is frequently accessed together within a certain range, these files are closely related. The parameter range_size determines the size of an access range in which the files share the same semantic distance for a specified file. The parameter rec_distance is the reciprocal of the distance from the specified file to an access range. To discover the access patterns in storage systems, we split the access stream into access sequences of equal size; each access sequence



	AB		C	D	E	F	G	H	I	J	K
A	-	5	4	1	3/2	1	1	1/2	1/2	1/3	1/3

Reference reciprocal-distances between file A and other files

Figure 6 Case study for the calculation of the reference reciprocal-distances.

is placed into a cutting window, and the parameter cut_size is the window size of the cutting window. Figure 6 shows a case study for the calculation of the reference reciprocal distances between a target file (file A) and other files, where the range_size and cut_size are 4 and 27, respectively. When the number of files in a cutting window reaches the cut_size, the reference reciprocal distances between the target file and other files in the window are calculated. The closer the access range is to the target file, the greater its reciprocal distance. The target file may repeatedly appear in the access stream of the cutting window. For this case, the latest target file is used instead of the previous one to calculate the reciprocal distances for the files in the subsequent sequence. A reciprocal-distance graph that is similar to a probability graph is constructed to infer file correlations. In the graph, the value between any two files is the sum of all the reciprocal distances between them. As seen in Figure 6, B has the largest reciprocal-distance for the target file because it is always within the first access range of A. Figure 6 shows the calculation of the reciprocal distances between file A and the other files. The reciprocal distances between any two files can be calculated in the same way. A reciprocal-distance graph is constructed when initializing the system, and the graph is updated in each cutting window. The reciprocal-distance graph is also traversed to construct a prefetching tree for prefetching related files.

3.3 Self-tuning prefetching

3.3.1 *Prefetching decision proposal*

The WAN file system client uses different prefetching strategies for metadata prefetching by comparing the cache hit ratios of the two different prefetching strategies. Thus, a prefetching strategy is selected that has a higher recent cache hit ratio to prefetch the metadata. Our method counts the hit ratios of the two different prefetching strategies in the current sliding window. The size of the sliding window is win_size, which can be set by the user. The number of sliding windows is the length of the access stream (including the paths involved in the VFS path lookup) divided by the win_size. The hit ratios for the two prefetching schemes were set to 0 at the beginning of the current sliding window. The prefetching strategy remains unchanged in the sliding window. The hit ratios for the two prefetching strategies must be calculated for comparison. The cache ratio of the selected prefetching strategy is the cache ratio of the current sliding window. A new thin-cache is created to calculate the ratio for the unselected prefetching strategy at the beginning of each sliding window. The cache uses the key-value model to organize cached metadata; the key is the directory or file path and the value is the metadata. Once a new thin-cache is created, all keys and thin-metadata (a small portion of the original metadata, which indicates whether the file is a regular file) in the original cache are copied to the new thin-cache. The thin-cache does not need to satisfy the client's metadata requests. The paths prefetched by the unselected prefetching strategy are stored in the thin-cache. For each client request, the self-tuning prefetching algorithm always keeps track of the hit ratios for the two prefetching strategies. At the end of the current sliding window, the prefetching strategy with a higher hit ratio is selected by the decision maker for the next sliding window. If the two prefetching strategies have the same hit ratio, the prefetching strategy remains unchanged.

The proposed decision maker procedure is shown in Figure 7. Both the directory-directed and correlation-directed prefetching schemes are applied to the same access stream in the proposed decision maker

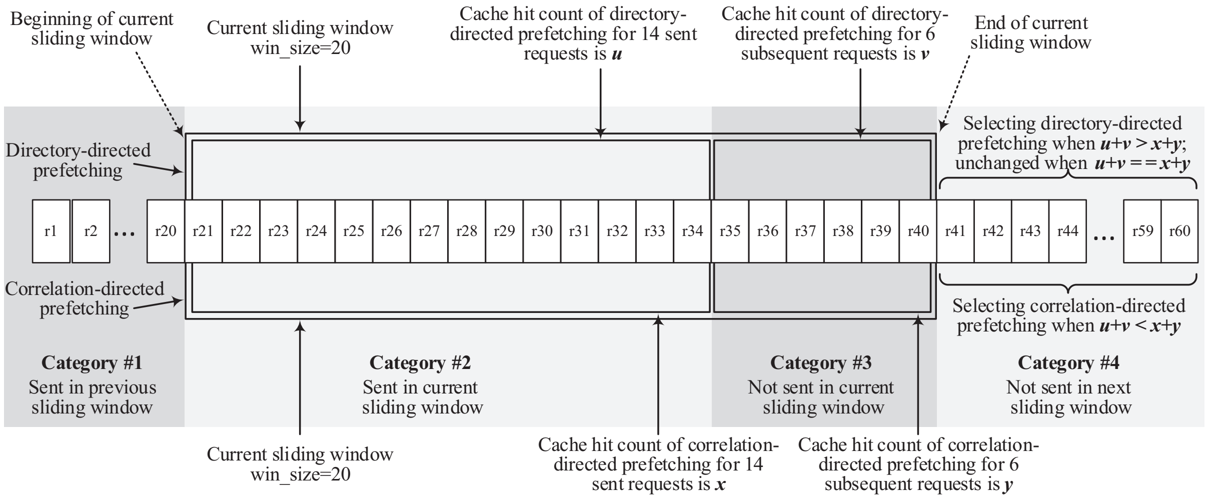

Beginning of current sliding window
Current sliding window win_size=20
Cache hit count of directory-directed prefetching for 14 sent requests is *u*
Cache hit count of directory-directed prefetching for 6 subsequent requests is *v*
End of current sliding window
Directory-directed prefetching
Selecting directory-directed prefetching when *u*+*v* > *x*+*y*; unchanged when *u*+*v* = = *x*+*y*
r1 r2 … r20 r21 r22 r23 r24 r25 r26 r27 r28 r29 r30 r31 r32 r33 r34 r35 r36 r37 r38 r39 r40 r41 r42 r43 r44 … r59 r60
Correlation-directed prefetching
Selecting correlation-directed prefetching when *u*+*v* < *x*+*y*
Category #1 Sent in previous sliding window
Category #2 Sent in current sliding window
Category #3 Not sent in current sliding window
Category #4 Not sent in next sliding window
Current sliding window win_size=20
Cache hit count of correlation-directed prefetching for 14 sent requests is *x*
Cache hit count of correlation-directed prefetching for 6 subsequent requests is *y*

Figure 7 Proposed decision maker procedure.

procedure, and the hit ratios for two prefetching strategies in a current sliding window are tracked by the decision maker. A sliding window can hold 20 metadata access requests (win_size = 20). In a current sliding window, 14 requests (r21–r34) are sent, and the subsequent 6 (r35–r40) requests can be sent in the future. The cache hit counts for either directory-directed or correlation-directed prefetching are u and x, respectively. The cache hit counts of the six subsequent requests for directory-directed and correlation-directed prefetching are v and y, respectively. If $u+v > x+y$, directory-directed prefetching will achieve a higher cache hit ratio than that of correlation-directed prefetching in a current window, and directory-directed prefetching will be selected as the prefetching strategy for the next window. If $u+v == x+y$, the two prefetching strategies have an equal hit ratio in a current window and the prefetching strategy of the current window can be applied to the next window. If $u+v < x+y$, the correlation-directed prefetching achieves a higher cache hit ratio and this will be selected as the prefetching strategy for the next window.

3.3.2 *Self-tuning prefetching refinements*

Once the access stream is received, the self-tuning prefetching scheme mines the access sequence and constructs a probability/reciprocal-distance graph, which can then be used to derive file correlations. Both directory-directed and correlation-directed prefetching are triggered to improve the hit ratio. For the self-tuning prefetching scheme, the hit ratios of the original and thin caches require calculation. The original cache hits can be calculated by simply counting the cache hits for each request. To calculate the hit ratio for the thin-cache, two situations need to be considered in which either correlation-directed or directory-directed prefetching are applied to the thin-cache.

When applying correlation-directed prefetching to the thin-cache, for each metadata request, the correlation-directed prefetching runs after directory-directed prefetching (applied to the original cache) such that it can obtain the information whether the currently requested file is a directory or regular file without additional remote access. This is because the previously run prefetching provides this information through remote access or cache hit. If the currently requested file is a directory and cannot be found in the thin-cache, the requested path needs to be stored in the thin-cache and marked as a directory. It does not need to create a vertex for a directory in a probability/reciprocal-distance graph. If the requested path can be found in the thin-cache, the hit ratio of the thin-cache is updated. If the currently requested file is a regular file and cannot be found in the thin-cache, the requested path needs to be stored in the thin-cache and marked as a regular file. If the associated vertex exists, files that are closely related to the requested file are prefetched by the probability/reciprocal-distance graph. If there is no vertex for the requested file, a vertex is created for the file. If the requested file path can be found in the thin-cache, the hit ratio of the thin-cache is updated.

When applying directory-directed prefetching to the thin-cache, the client must send remote requests to obtain prefetched paths to calculate the hit ratio of the thin-cache, resulting in additional network overhead. To address this issue, three mechanisms are proposed: (1) asynchronous fetching, (2) data compression, and (3) calculating the hit ratio based on assumptions.

Asynchronous fetching. If the currently requested path cannot be found in the thin-cache, it will be stored in the requested queue rather than sent to the remote server immediately. The parameter queue_size is the threshold length of the queue. Once the threshold is reached, the requests are asynchronously sent to the remote server to fetch the paths by directory-directed prefetching.

Data compression. To reduce the amount of prefetched data that is transferred from the server side to the client-side, prefetched paths and metadata are compressed by a compression algorithm before network transmission. The prefetched file paths are concatenated on the server side and compressed prior to sending. After receiving the data, the client decompresses the data and splits the string to obtain the prefetched file paths. Prefetched metadata are compressed in a similar manner. The Facebook Zstandard (see Wikipedia) compression algorithm was used to compress the data.

Calculating the hit ratio based on the following assumptions. In some cases, the prefetching strategy for the next sliding window can be calculated before the end of the current sliding window. Before calculating the hit ratio, the maximum cache hits at the current moment for the current sliding window can be calculated by

$$\begin{aligned} \text{tcache\_max\_hits} &= \text{tcache\_hits} + (\text{win\_size} - \text{cnum}),\\ \text{ocache\_max\_hits} &= \text{ocache\_hits} + (\text{win\_size} - \text{cnum}), \end{aligned} \tag{1}$$

where tcache_max_hits is the theoretical maximum of the thin-cache hits based on the current condition for the current sliding window, tcache_hits is the current thin-cache hits, ocache_max_hits is the theoretical maximum of the original cache hits based on the current condition for the current sliding window, ocache_hits is the current original cache hits, win_size is the size of the sliding window, and cnum is the number of requests that have been satisfied in the current sliding window. If tcache_max_hits $<$ ocache_hits, the thin-cache hits can never exceed the original cache hits and the prefetching strategy for the next sliding window remains unchanged. Therefore, the cache hits calculation can be canceled to reduce network communications. If ocache_max_hits $<$ tcache_hits, the prefetching strategy for the next sliding window needs to be changed, and the cache hits calculation can be canceled.

4 Implementation

The GVDS client is built on top of the file system in user space (FUSE), which consists of a kernel module and user-level daemon. The FUSE driver is registered to the VFS when loading the kernel module. It acts as a middleman to route requests and data between the VFS and user-level daemon. A dev/fuse block device is also registered by the kernel module to provide an interface between the user-level daemon and kernel. The I/O requests of user applications are read from /dev/fuse, and processed by the user-level daemon before the replies are written back to /dev/fuse. The I/O operations performed by user applications are sent to the VFS, which routes the I/O operations to the kernel driver of FUSE. The driver allocates a FUSE request structure for each operation and places it in the queue. The user application is blocked until the operation is completed. The user-level daemon selects the request from the queue by reading from /dev/fuse. The user-level daemon processes the request and writes it back to /dev/fuse; the driver then marks the request as completed and wakes up the user process.

The architecture of our experimental platform is shown in Figure 8. When a user application performs some operations on a mounted FUSE, the VFS routes the operations to the kernel driver of the FUSE. The request is finally picked by the user-level daemon of the FUSE. The daemon allocates a certain amount of memory space to cache metadata. The parameter cache_size is the threshold number of metadata items that are stored in the client-side cache. The self-tuning prefetching is run by the daemon. Multiple threads in the daemon are created to handle concurrent access requests. To maintain metadata consistency, the cache needs to be locked when the client updates the cache. When accessing metadata, if the cache hits, the FUSE daemon returns the requested metadata to the application; if the cache misses, it needs to send the request to the I/O server to obtain the metadata via a remote procedure call (RPC). The requested data is then cached and returned to the application. The I/O proxy receives the metadata request from the client, and it first finds the server-side cache. If the cache hits, the requested metadata are returned. If the cache misses, the requested metadata are accessed in the local file system, and the server-side cache is updated. The requested metadata are then returned to the client. In this study, only the read-only

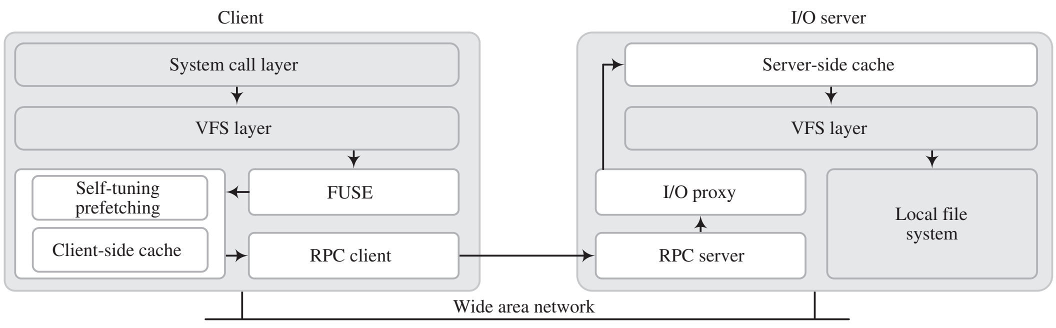


Figure 8 Experimental platform architecture.

workload is considered, and the space of the server-side cache is sufficiently large. In future work, we will consider situations with a write workload and insufficient memory space.

5 Evaluation

Our experiments were conducted on 2-node machines, one of which was the client and the other the I/O server and manager node. The client machine was configured with two 20-core 2.2 GHz Intel Xeon 4114 CPUs, 128 GB of memory, two 7.2 k RPM 4 TB disks, and the Ubuntu 18.04 LTS operating system. The server machine was a virtual machine, configured with 48-core 2.7 GHz CPUs, 126 GB of memory, 900 GB and 10 TB disks, and the Ubuntu 18.04 LTS operating system. The client machine was located in Beijing, and the server machine located in Guangzhou. The round-trip time (RTT) between the two sites was 34 ms.

5.1 Evaluation methodology

We used several large disk traces collected in real systems to evaluate the benefits of self-tuning metadata prefetching. The following four system traces were used in our experiments.

WebSearch2 trace (Web-trace, see UMass Trace Repository) was collected from a machine running a web search engine. This I/O access log was gathered from 0:0:0 to 4:16:36 and consisted of 5 storage devices. Each record in the access log comprises five fields: (1) application specific unit, (2) logical block address (LBA) of the requested data, (3) size of the requested data block, (4) operation type (read or write), and (5) operation time. Each requested data block can be treated as the content of a small file. For each requested data block, a file is created to store it, and the file name is the LBA. For each LBA that is not accessed but in the range of the smallest LBA to the largest LBA, an empty file is also created. To explore the impact of single and multiple directories on prefetching schemes, two mapping rules were used to place all of the files. For the single directory mapping rule (SDMR), all files are placed into a single space directory that corresponds to a space and located in the local file system of the I/O server. For the multiple directories mapping rule (MDMR), several directories are created to store files. All directories are stored together into a single directory that corresponds to the space. The threshold number of files in a directory is file_per_dir. The serial number of the directory for a file is the serial number of the file divided by files_per_dir.

FUJITSU K5 trace (K5-trace, see Storage Networking Industry Association) was created by capturing the K5-Trace cloud service. The format of each record is the same as that of Web-Trace. Files were created and placed in the same way as Web-Trace.

Sequential Trace (Seq-trace) was created by sequentially merging K5-Trace and Web-Trace. This trace simulates the sequential execution of two different applications, which is common in large computing environments.

Concurrent Trace (Con-trace) was created by simultaneously replaying Web-Trace and K5-Trace. The two traces are run simultaneously by two processes. This trace is used to simulate the concurrent execution of two different applications.

Zero-think-time trace replays and skipped trace intervals with no activities were conducted. We also implemented state-of-the-art prefetching schemes to provide a fair comparison. Table 1 provides the experimental parameters used in the experiments.

Table 1 Experiment parameters

dir_prefetch_items	branch	depth	support	win_size	queue_size	files_per_dir	range_size	cut_size	cache_size
120	12	3	1	3000	200	120	10	20	10000

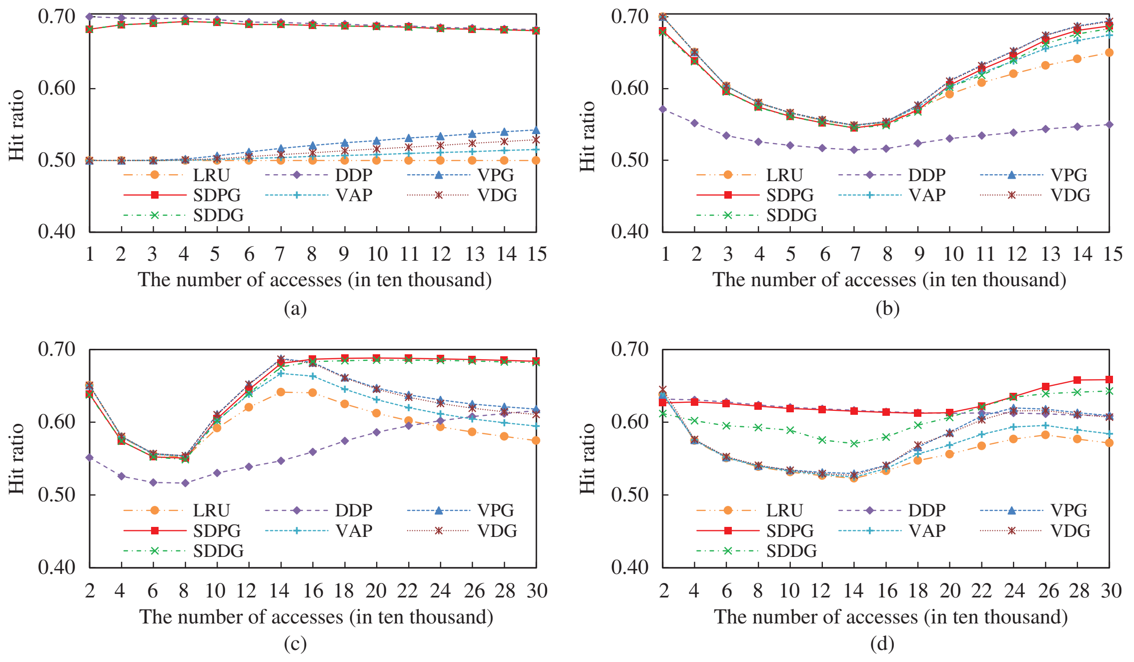


Figure 9 (Color online) Hit ratio variations for seven different schemes when replaying four different traces under SDMR. (a) Web-trace; (b) K5-trace; (c) Seq-trace; (d) Con-trace.

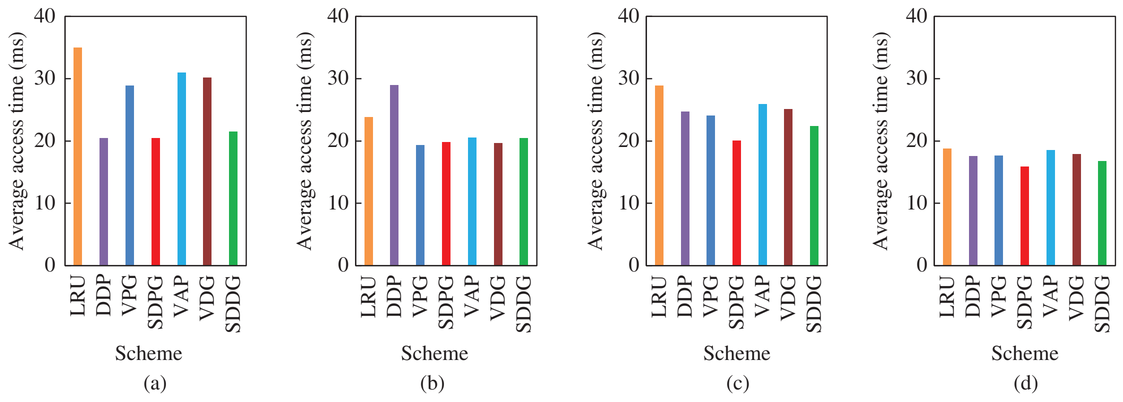


Figure 10 (Color online) Average access time for seven different schemes when completing four different trace replays under SDMR. (a) Web-trace; (b) K5-trace; (c) Seq-trace; (d) Con-trace.

The seven different schemes evaluated in our experiments are as follows: (1) caching without prefetching (LRU); (2) directory-directed prefetching (DDP); (3) correlation-directed prefetching based on the variant probability graph (VPG); (4) self-tuning prefetching by coupling DDP and VPG (SDPG); (5) correlation-directed prefetching based on the variant apriori algorithm (VAP), which is used in CFFS [5]; (6) correlation-directed prefetching based on the variant semantic distance algorithm (VDG); and (7) self-tuning prefetching by coupling DDP and VDG (SDDG).

5.2 Hit ratio and access time

5.2.1 *SDMR test results*

The hit ratio variations for the seven different schemes when running four different workloads under SDMR are shown in Figure 9. The average metadata access time of the seven different schemes for the four different workloads is shown in Figure 10.

Figures 9(a) and 10(a) provide two aspects of the performance of the seven different schemes when replaying Web-trace under SDMR. SDPG, SDDG, and DDP had the best access performance. This is because a large number of files were accessed sequentially in the same directory, and DDP significantly improved the cache hit ratio. SDPG and SDDG inherited the advantages of directory-directed prefetching; hence, there were no significant hit ratio differences. There was no significant average access time difference between SDPG and DDP, which means that the overhead caused by SDPG was relatively low. When the trace replay was complete, the SDPG improved the hit ratios by up to 13.84%, 16.58%, and 15.23% as compared with those of the VPG, VAP, and VDG, corresponding to 29.26%, 34.03%, and 32.15% reductions in the average access time, respectively. The SDDG achieved the same hit ratio improvement as SDPG, and reduced the average access time by 25.71%, 30.73%, and 28.74% as compared to those of VPG, VAP, and VDG, respectively. VPG, VAP, and VDG improved the cache hit ratios by 4.24%, 1.5%, and 2.85%, respectively, as compared with that of LRU. These results indicate that prefetching can improve the access performance, and directory-directed prefetching achieves a better access performance compared with that of correlation-directed prefetching in Web-trace. Regarding correlation-directed prefetching, the order of the best to worst performance was VPG, VDG, and VAP. This is because the access stream was split into fixed-size short sequences to discover the access patterns in VDG and VAP, which led to a loss of frequent sequences that are split into two or more subsequences.

Figures 9(b) and 10(b) show two views of the performance of the seven different schemes when replaying K5-trace under SDMR. DDP provided the worst access performance because the numerous unrelated files prefetched by DDP consumed a large amount of cache space. This phenomenon demonstrates that a single prefetching scheme cannot meet various prefetching requirements. When the number of accesses was less than 90000, there were no significant hit ratio differences for LRU, VPG, VAP, and VDG. This is because the amount of correlation information in the probability/reciprocal distance graph was insufficient for prefetching. SDPG and SDDG had the same hit ratio, which was slightly less than that of LRU. This is because SDPG and SDDG employed directory-based prefetching by default in the first sliding window and there was no historical access information to select a proper prefetching strategy. Beyond point 90000, the hit ratios of VPG, VAP, VDG, SDDG, and SDPG gradually exceeded that of LRU.

Figures 9(c) and 10(c) show two perspectives of the performance of the seven different schemes when replaying Seq-trace under SDMR. When the number of accesses was less than 140000, there were no significant hit ratio differences between SDPG, SDDG, VPG, VDG, and VAP. The hit ratios of VPG, VDG, and VAP dropped dramatically, while the hit ratios of SDPG and SDDG remained relatively constant. There are three reasons for this behavior. The first is that files in the same directory tended to be accessed together beyond point 140000. Second, the locality of data access decreased significantly. Third, VPG, VDG, and VAP employed a unified prefetching strategy and could not dynamically adjust the prefetching strategy when facing varied access patterns, resulting in a significant drop in their prefetching accuracies.

Figures 9(d) and 10(d) show two aspects of the performance of the seven different schemes when replaying Con-trace under SDMR. When the number of accesses was 20000, the hit ratios of SDPG and SDDG were slightly lower than the hit ratios of the other schemes. This is because an incorrect prefetching strategy was selected in some sliding windows owing to insufficient historical access information to predict the access patterns. Beyond that point, the hit ratios of SDPG and SDDG quickly exceeded those of LRU, VPG, VAP, and VDG. The hit ratios of SDPG and SDDG were always greater than those of LRU, VPG, VAP, and VDG as the number of accesses was increased from 40000 to 300000. This behavior illustrates that our self-tuning prefetching can dynamically detect the cache dominated by a particular workload and then adaptively adjust the prefetching strategy to improve the cache hit ratio for the multi-workload concurrency scenario. When the number of accesses was 300000, SDPG improved the hit ratios of LRU, DDP, VPG, VAP, and VDG by up to 8.69%, 4.98%, 4.94%, 7.44%, and 5.16%, corresponding to 15.63%, 9.78%, 9.98%, 14.49%, and 11.29% reductions in the average access time, respectively. SDDG outperformed LRU, DDP, VPG, VAP, and VDG by 7.15%, 3.44%, 3.40%, 5.90%, and 3.62% in terms of the hit ratio, and improved the average access time by 11.06%, 4.89%, 5.1%, 9.86%, and 6.48%, respectively.

5.2.2 *MDMR test results*

The hit ratio variations at runtime and the average access time for the seven different schemes under MDMR are shown in Figures 11 and 12, respectively. There are more advantages of SDPG and SDDG

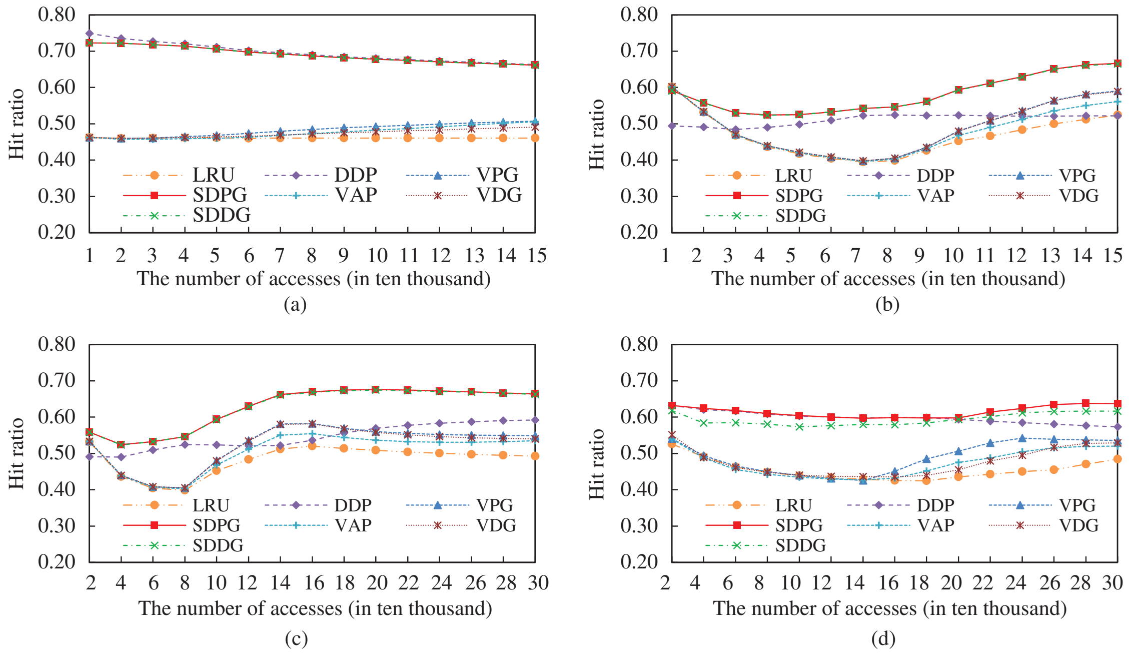


Figure 11 (Color online) Hit ratio variations for seven different schemes when replaying four different traces under MDMR. (a) Web-trace; (b) K5-trace; (c) Seq-trace; (d) Con-trace.

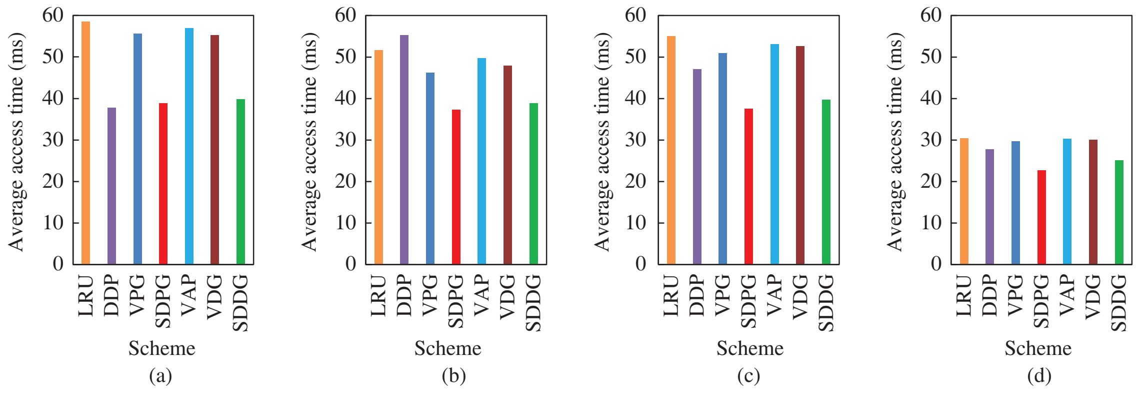


Figure 12 (Color online) Average access time for seven different schemes when completing four different trace replays under MDMR. (a) Web-trace; (b) K5-trace; (c) Seq-trace; (d) Con-trace.

under MDMR than under SDMR. The average access time for the seven schemes increased to some extent owing to the higher levels of the directory tree and the larger VFS path lookup counts.

The hit ratio variations and average access time for all schemes when replaying Web-trace under MDMR are shown in Figures 11(a) and 12(a), respectively. The performance of DDP was better than that of the SDMR scenario. For instance, when the number of accesses was 10000, DDP outperformed LRU, VPG, VVAP, and VDG by more than 28% in terms of the hit ratio under MDMR, while there was only a 20% improvement under SDMR. There were no significant differences for DDP, SDPG, and SDDG in terms of the hit ratio and average access time. This behavior indicates that SDPG and SDDG fully inherited the advantages of directory-directed prefetching; hence, they achieved much higher hit ratios than LRU, VPG, VAP, and VDG.

The hit ratio variations and average access time for all schemes when replaying K5-trace under MDMR are shown in Figures 11(b) and 12(b), respectively. The hit ratios of SDPG and SDDG were greater than those of the other schemes when the number of accesses was greater than 10000, providing significantly different results to the experimental results under SDMR. This is because DDP achieved a better performance under MDMR compared with that under SDMR. Our self-tuning prefetching dynamically detects the workload to achieve a fine-grained online variation of prefetching strategies, increasing the performance of SDPG and SDDG.

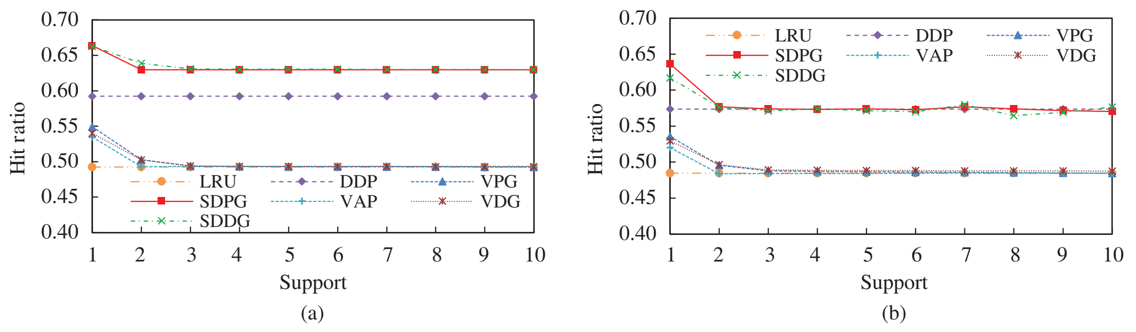


Figure 13 (Color online) Hit ratio variations of seven different schemes for Seq-trace and Con-trace when varying support under MDMR. (a) Seq-trace; (b) Con-trace.

The hit ratio variations and average access time for all schemes when replaying Seq-trace under MDMR are shown in Figures 11(c) and 12(c), respectively. The advantage of DDP under MDMR is more prominent than that under SDMR. This is because a large number of directories are prefetched to improve the hit ratio. SDPG and SDDG efficiently couple directory-directed and correlation-directed prefetching to improve the hit ratios by taking advantage of their advantages and discarding their disadvantages. As a result, the performance of SDPG and SDDG is much better than those of the other schemes.

The hit ratio variations and average access time for all of the schemes when replaying Con-trace under MDMR are shown in Figures 11(d) and 12(d), respectively. SDPG always had the highest hit ratio because it could adaptively adjust the prefetching strategy based on the workload characteristics. The hit ratio of SDDG was slightly less than that of SDPG, which indicates that SDPG was more capable than SDDG when various workloads were run concurrently. DDP had a higher hit ratio than those of LRU, VPG, VAP, and VDG throughout the entire access process. This is because a large number of directories were prefetched and the hit ratio of DDP was significantly improved. At the end of the trace, SDPG improved the hit ratios for LRU, DDP, VPG, VAP, and VDG by 15.22%, 6.32%, 10.08%, 11.65%, and 10.73%, corresponding to 25.24%, 18.11%, 23.53%, 24.94%, and 24.19% reductions in the average access time, respectively. The hit rate of SDDG was 96.88% that of SDPG, and reduced the average access time of LRU, DDP, VPG, VAP, and VDG by 17.38%, 9.5%, 15.5%, 17.06%, and 16.23%, respectively.

5.2.3 *Support parameter effects*

The support parameter may affect the benefits of our proposed self-tuning prefetching. Figures 13(a) and (b) show the effects of varying the support from 1 to 10 for Seq-trace and Con-trace under MDMR. The experimental results show that VPG, VAP, VDG, SDPG, and SDDG achieved their highest hit ratios when the support was one. For Seq-trace, the hit ratios for VPG, VAP, VDG, SDPG, and SDDG decreased by 5.55%, 4.16%, 4.62%, 3.18%, and 3.4%, respectively, as the support was increased from 1 to 3. This indicates that the larger the support parameter, the worse the performance degradation for prefetching based on file correlation.

5.2.4 *Cache_size parameter effects*

The hit ratio variations for Seq-trace and Con-trace under MDMR are shown in Figures 14(a) and (b), respectively. As the cache_size increased from 2000 to 20000 metadata items, the hit ratios for LRU, DDP, VPG, VAP, VDG, SDDG, and SDPG for Seq-trace improved by 3.08%, 7.84%, 3.19%, 2.27%, 2.67%, 4.52%, and 5.12%, and for Con-trace by 2.68%, 7.99%, 2.91%, 1.52%, 2.29%, 5.16%, and 6.44%, respectively. The larger the cache_size, the higher the hit ratio. However, the cache_size cannot be set too large because it can lead to a larger consistency maintenance overhead.

5.2.5 *Prefetch parameter effects*

Figure 15 shows the effects of the prefetch parameters for all schemes when replaying Seq-trace and Con-trace under MDMR. Figures 15(a) and (b) show the effects of dir_prefetch_items variations for Seq-trace and Con-trace, respectively. When dir_prefetch_items varied from 50 to 150, the hit ratios of DDP, SDDG, and SDPG increased dramatically. For example, the hit ratios of DDP, SDDG, and SDPG under Seq-trace increased by 2.98%, 3.51%, and 3.52%, and under Con-trace by 2.73%, 1.86%, and 2.17%,

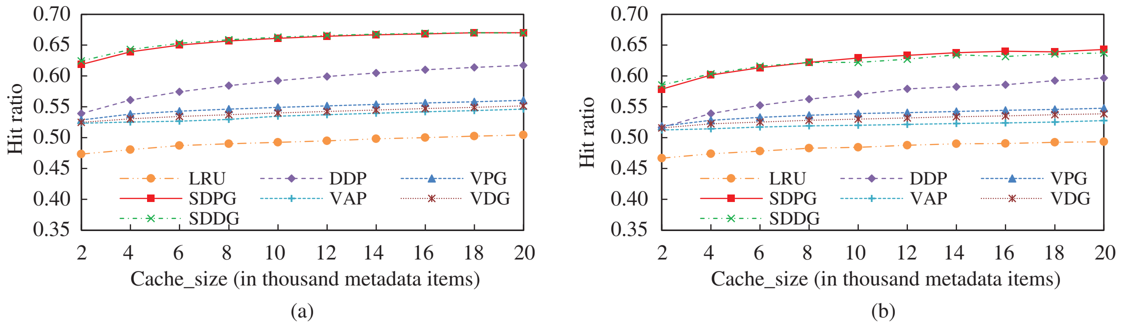


Figure 14 (Color online) Effects of parameter cache_size for Seq-trace and Con-trace under MDMR. (a) Seq-trace; (b) Con-trace.

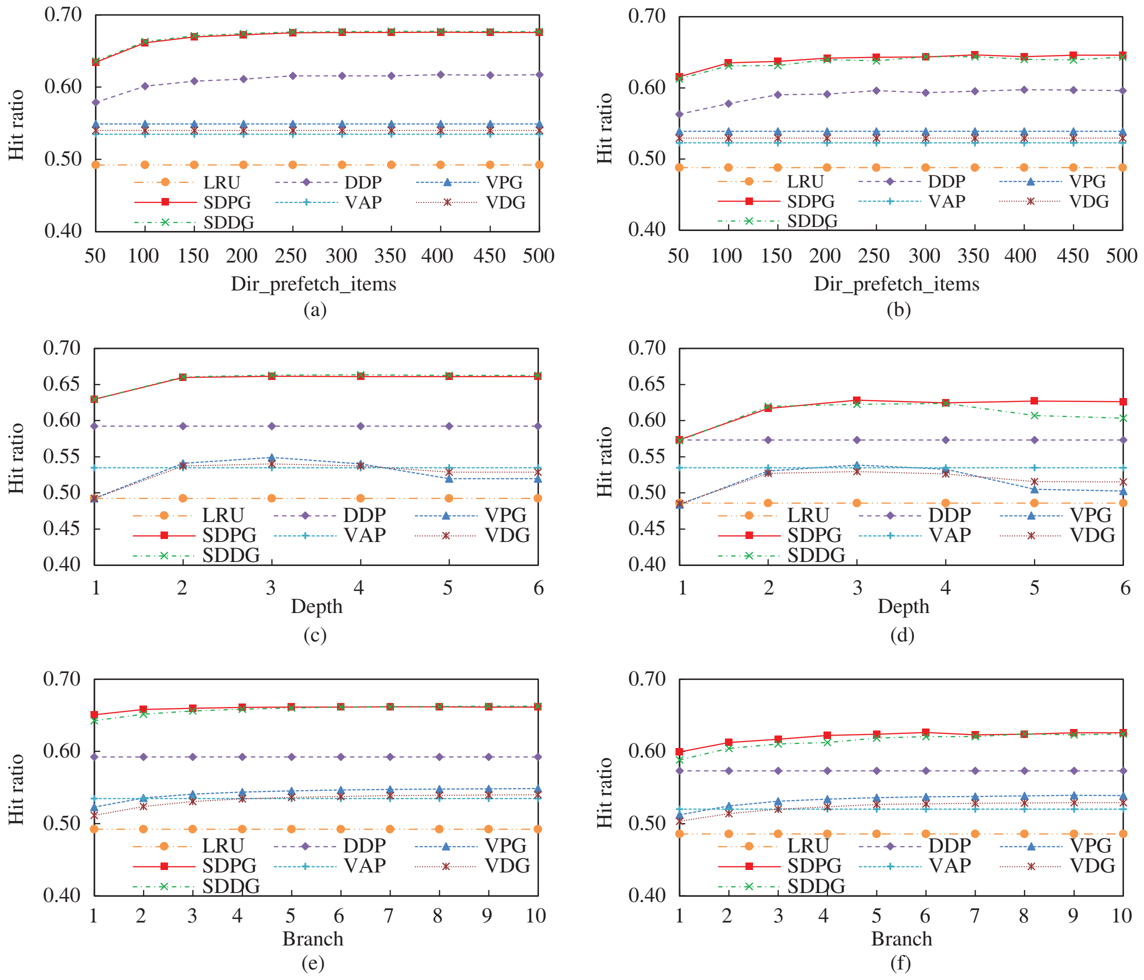


Figure 15 (Color online) Effects of the prefetch parameters for Seq-trace and Con-trace under MDMR. (a), (c) and (e) Seq-trace; (b), (d) and (f) Con-trace.

respectively. As dir_prefetch_items increased further, the hit ratios increased slightly. This implies that increasing dir_prefetch_items beyond a certain point does not provide further performance benefits under MDMR.

Figures 15(c) and (d) show the effects of varying depth for Seq-trace and Con-trace, respectively. The hit ratios of VPG, VDG, SDDG, and SDPG increased dramatically when depth was varied from one to two. When varying depth from four to five, performance degradation was observed for VPG and VDG. This is because the depths of the most frequently accessed sequences were not greater than four; incorrect metadata was prefetced, which then occupied the cache space.

Figures 15(e) and (f) show the effects of varying branch for Seq-trace and Con-trace, respectively. A large branch had a positive effect on performance. For example, as branch increased from 1 to 10, the hit ratios of VPG, VDG, SDDG, and SDPG under Seq-trace increased by 2.56%, 2.86%, 2.03%, and 1.08%, and under Con-trace by 2.65%, 2.59%, 3.61%, and 2.63%, respectively.

6 Related work

Prefetching and caching have long been studied in parallel applications with intensive I/O and modern file systems for improving I/O system performance. Several sophisticated prefetching and caching schemes have been proposed to reduce overhead caused by high-cost network accesses and disk operations. Most of these are implemented in either the I/O library layer on the client-side of storage systems or the file server layer.

There is no doubt that data prefetching and caching are effective in reducing access latency. Most previous studies rely on hints from the application to determine which data should be fetched beforehand. Better access performance can be achieved using application-controlled prefetching [20] and informed prefetching [21, 22]. However, these prefetching approaches may result in significant system performance degradation when inappropriate information is provided from running applications [13]. Moreover, this type of prefetching has limited applicability as the source code needs to be revised.

Prefetching and caching approaches are used to alleviate file access latency in distributed/parallel file systems. The read cache prefetching technique was employed by Ceph [3] to reduce network communication overhead between clients and metadata servers for a better access performance. The GFS [4] relies on simple heuristics, such as sequentially prefetching data that may be read by the following requests and then achieving a higher level of I/O throughput. However, whether prefetching is enabled depends on the user.

There are several sophisticated prefetching schemes for storage servers. C-Miner [7] uses frequent sequence matching on block I/O data in storage servers to benefit I/O optimization. Hsu et al. [8] proposed an automatic locality-improving storage approach, which automatically reorganizes selected disk blocks by analyzing block correlations such that related objects are clustered and prefetched sequentially. SEER [11] uses semantic distances between files to estimate the degree of similarity between them and prefetches as many related files as possible onto the mobile station such that the files will be available after network disconnection. Zhang et al. [5] proposed a composite-file file system that allows many-to-one mappings of files to metadata. It supports both directory-directed and embedded-reference-based consolidation. This approach is similar to prefetching, for example, the metadata of multiple associated files are prefetched when accessing the metadata of a composite file. Redundant metadata information can be significantly reduced when conducting file consolidation. Therefore, it provides better access performance than common prefetching. Pacaca [23] is a client-side cache management framework that integrates object clustering, parallelized prefetching, and cost-aware caching to achieve a high level of cloud storage performance. However, the current prefetching schemes conducted on file servers cannot dynamically adapt to workload changes; they fail to hide the latency caused by network communications when facing multiple workloads. In summary, no existing schemes respond to evolving and changing access patterns by dynamically and adaptively self-adjusting the prefetching strategy to yield performance enhancements.

7 Conclusion and future work

This paper proposes a new self-tuning client-side metadata prefetching scheme that uses two different prefetching strategies and dynamically adapts to workload changes. A novel decision-making mechanism is proposed to effectively determine the best prefetching strategy; either directory-directed or correlation-directed. An original cache and a thin-cache are constructed using different prefetching strategies to study their performance. The hit ratios for the two prefetching strategies are calculated to provide a comparison at the end of a current sliding window. The prefetching strategy with a higher cache ratio is applied to the next sliding window. We also propose the following three effective mechanisms to reduce the overhead caused by prefetching strategies: (1) asynchronous fetching, (2) data compression, and (3) calculating the hit ratio based on assumptions. Using several real system traces, we show that our scheme is reasonably fast with the same space requirement as other schemes.

Our study has several limitations. First, we only evaluated four system workloads; other workloads, such as those with large-scale file accesses, will be tested in future work. Second, our approach requires the support of the server-side cache, as the running of our scheme would otherwise be highly time consuming, and the disadvantages may outweigh the advantages. In the future, we aim to improve the efficiency of our proposed approach so that it has a small memory space requirement.

Acknowledgements This work was supported by National key R&D Program of China (Grant No. 2018YFB0203901), National Natural Science Foundation of China (Grant No. 61772053), the Fund of the State Key Laboratory of Software Development Environment (Grant No. SKLSDE-2018ZX-10), and Science Challenge Project (Grant No. TZ2016002).

References

1 Wrzeszcz M, Trzepla K, Slota R, et al. Metadata organization and management for globalization of data access with onedata. In: Proceedings of the International Conference on Parallel Processing and Applied Mathematics, Krakow, 2015. 312–321
2 Grimshaw A, Morgan M, Kalyanaraman A. GFFS—the XSEDE global federated file system. Parall Process Lett, 2013, 23: 1340005
3 Weil S A, Brandt S A, Miller E L, et al. Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Washington, 2006. 307–320
4 Ghemawat S, Gobioff H, Leung S T. The Google file system. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, New York, 2003. 20–43
5 Zhang S, Catanese H, Wang A A I. The composite-file file system: decoupling the one-to-one mapping of files and metadata for better performance. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies, Santa Clara, 2016. 15–22
6 Beckmann N, Chen H, Cidon A. LHD: improving cache hit rate by maximizing hit density. In: Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation, Renton, 2018. 389-403
7 Li Z, Chen Z, Srinivasan S M, et al. C-Miner: mining block correlations in storage systems. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, San Francisco, 2004. 173–186
8 Hsu W W, Smith A J, Young H C. The automatic improvement of locality in storage systems. ACM Trans Comput Syst, 2005, 23: 424–473
9 Ding X, Jiang S, Chen F, et al. DiskSeen: exploiting disk layout and access history to enhance I/O prefetch. In: Proceedings of USENIX Annual Technical Conference, Boston, 2007. 7: 261–274
10 Jiang S, Ding X, Xu Y, et al. A prefetching scheme exploiting both data layout and access history on disk. ACM Trans Storage, 2013, 9: 1–23
11 Kuenning G H. The design of the seer predictive caching system. In: Proceedings of the 1st Workshop on Mobile Computing Systems and Applications, New York, 1994. 37–43
12 Griffioen J. Performance measurements of automatic prefetching. In: Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems, New York, 1995. 165–170
13 Li X, Xiao L, Qiu M, et al. Enabling dynamic file I/O path selection at runtime for parallel file system. J Supercomput, 2014, 68: 996–1021
14 Battle L, Chang R, Stonebraker M. Dynamic prefetching of data tiles for interactive visualization. In: Proceedings of the 2016 International Conference on Management of Data, San Francisco, 2016. 1363–1375
15 Wei B, Xiao L M, Wei W, et al. A new adaptive coding selection method for distributed storage systems. IEEE Access, 2018, 6: 13350–13357
16 Lin W, Xu S Y, Li J, et al. Design and theoretical analysis of virtual machine placement algorithm based on peak workload characteristics. Soft Comput, 2017, 21: 1301–1314
17 Patrick C M, Kandemir M, Karakoy M, et al. Cashing in on hints for better prefetching and caching in PVFS and MPI-IO. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Chicago, 2010: 191–202
18 Henschel R, Simms S, Hancock D, et al. Demonstrating Lustre over a 100 Gbps wide area network of 3500 km. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, 2012. 1–8
19 Carns P, Lang S, Ross R, et al. Small-file access in parallel file systems. In: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, New York, 2009. 1–11
20 Cao P, Felten E W, Karlin A R, et al. A study of integrated prefetching and caching strategies. SIGMETRICS Perform Eval Rev, 1995, 23: 188–197
21 Habermann P, Chi C C, Alvarez-Mesa M, et al. Application-specific cache and prefetching for HEVC CABAC decoding. IEEE Multimedia, 2017, 24: 72–85
22 Al Assaf M M, Jiang X, Qin X, et al. Informed prefetching for distributed multi-level storage systems. J Sign Process Syst, 2018, 90: 619–640
23 Hou B, Chen F. Pacaca: mining object correlations and parallelism for enhancing user experience with cloud storage. In: Proceedings of the 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2018. 293–305