Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.065090

ARTICLE

Dynamic Metadata Prefetching and Data Placement Algorithms for High-Performance Wide-Area Applications

Bing Wei, Yubin Li, Yi Wu*, Ming Zhong and Ning Luo

School of Cyberspace Security, Hainan University, Haikou, 570228, China

*Corresponding Author: Yi Wu. Email: wuyi@hainanu.edu.cn

Received: 03 March 2025; Accepted: 29 May 2025; Published: 30 July 2025

ABSTRACT: Metadata prefetching and data placement play a critical role in enhancing access performance for file systems operating over wide-area networks. However, developing effective strategies for metadata prefetching in environments with concurrent workloads and for data placement across distributed networks remains a significant challenge. This study introduces novel and efficient methodologies for metadata prefetching and data placement, leveraging fine-grained control of prefetching strategies and variable-sized data fragment writing to optimize the I/O bandwidth of distributed file systems. The proposed metadata prefetching technique employs dynamic workload analysis to identify dominant workload patterns and adaptively refines prefetching policies, thereby boosting metadata access efficiency under concurrent scenarios. Meanwhile, the data placement strategy improves write performance by storing data fragments locally within the nearest data center and transmitting only the fragment location metadata to the remote data center hosting the original file. Experimental evaluations using real-world system traces demonstrate that the proposed approaches reduce metadata access times by up to 33.5% and application data access times by 17.19% compared to state-of-the-art techniques.

KEYWORDS: Metadata prefetching; data placement; wide-area network file system (WANFS); concurrent workload optimization

1 Introduction

Today, considerable scientific data are produced in distributed infrastructures by many organizations. These data must be quickly collected, processed, and linked, which imposes distributed access requirements for scientific applications. In the past, researchers typically used grid computing systems to provide distributed access to applications. With a standard access interface, no differences exist between accessing grid computing systems and local machine resources. Many storage services that support access to geographically distributed data are already available inside grid infrastructures. For example, the Globus Connect service provides fast data migration and sharing across multiple datacenters using GridFTP [1]. However, many studies have shown that entire datasets must be completely transferred to computational nodes prior to the execution of a job in grid computing systems, even if only a small part of each dataset is required. WANFSes, such as NFS [2] and SSHFS [3], offer promising solutions for overcoming the disadvantages of grid computing systems. Transparent data access from any machine can be realized when using WANFSes; data can be accessed anywhere at any time by clients, and jobs can be executed in real time without waiting for the transmission of the entire dataset.

Although the jobs can be executed without transferring the entire dataset, the existing WANFSes can be further improved in terms of the read performance of small files in the wide-area network scenario. The low read performance of small files is caused by metadata fetching. The clients and servers of the WANFSes interact with each other through networks. When accessing a file, the client must ask the server for the file metadata. Interactions between clients and servers can incur severe I/O overheads, particularly for small files.

We tested the metadata access overhead of the different WANFSes for small files. The experiments were conducted at two sites of the Alibaba Cloud. We deployed Lustre [4], NFS [2], and SSHFS [3] at the sites to test the ratio of metadata access time to total access time. The two Alibaba Cloud sites were located in Beijing (BJ) and Shanghai (SH). The network latency (in ms) and the network bandwidth (in MB/s) between the two sites are illustrated in 6.1.1. Each site consisted of three virtual machines. Each virtual machine was configured with two vCPUs, 16 GB of memory, 100 GB of disk storage, and an Ubuntu 18.04 LTS operating system. A single client of each file system was mounted on virtual machines in BJ. The server components of each file system were deployed on the virtual machines in SH. 10,000 files were accessed by the clients of the file systems, with each file being accessed once.

Fig. 1 shows the metadata access time ratio for varied file sizes. The file size is the sum of metadata size and data size. The metadata access time ratio is defined by

$$\frac{access\_time\_of\_metadata}{access\_time\_of\_metadata\_and\_data}.$$

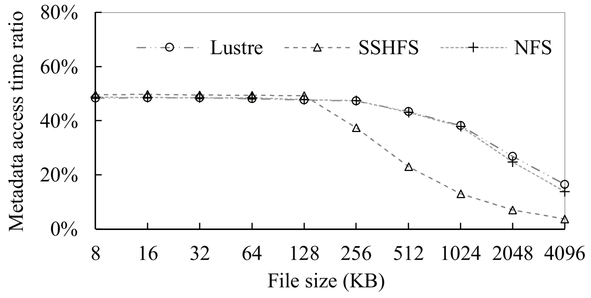


Figure 1: Metadata access time ratio for varied file sizes

The smaller the file, the greater the metadata access time ratio. This is because: 1) the metadata size of each file is fixed, 2) the smaller the file size, the smaller the data size, 3) the smaller the data size, the smaller the data access time, and 4) the smaller the data access time, the greater the metadata access time ratio. The metadata access time ratio can be as high as 49% when the file size is not greater than 128 KB. This demonstrates that reducing the metadata access overhead is the key to improving the access performance of small files.

Efficient metadata management mechanisms are the key to improving the access performance [5–7]. Several file systems prefetch a certain amount of metadata to improve the cache hit rate [4,8]. Existing file systems must provide storage services for multiple applications. However, they typically use a unified prefetching policy for all applications. Consequently, different prefetch requirements cannot be met. Regarding metadata access, existing prefetching policies can be improved in the following three aspects: 1) most non-frequency-mining-based prefetching approaches rely heavily on simple patterns (e.g., temporal locality, sequentiality, and loop references) and do not fully exploit semantic correlations between files [9]; 2) considerable frequency-mining-based prefetching work cannot easily capture file associations imposed by users, particularly when the historical access information is not sufficient to learn associations [10]; and 3) choosing a reasonable prefetching policy in concurrent workload scenarios is not straightforward, as the data of different applications are simultaneously handled by different processes [11]. Therefore, the efficient metadata access approach for small files needs further research.

In addition to the low performance of metadata access for small files, another aspect that the existing WANFSes can be further improved is the low write performance for all files. The low write performance of existing WANFSes is caused by remote writing. When writing a file in a wide-area network environment, the new file fragment is sent to the remote data center, where the original file created by the applications is located. This leads to low-bandwidth and high-cost network communications. Gfarm [12] and CalvinFS [13] placed data replicas throughout multiple datacenters to ensure reliability and reduce communication costs. However, the data placement solutions of existing WANFSes can be improved in the following two aspects: 1) data replicas cannot be placed in the target computing centers prior to computing, as the target computing data centers cannot be known in advance, resulting in remote writing when file data are written; and 2) replicas created for reliability do not benefit data writes because multiple replicas need to be updated simultaneously during writing [12,14].

In this study, our goal was to improve data access performance in a wide-area high-performance computing environments. We designed new approaches to overcome the low read performance for small files and the low write performance for all files. For read performance, using a unified prefetching policy incurs a low level of prefetching accuracy in concurrent workload scenarios. This led us to improve the accuracy by using a non-frequency mining policy and a frequency mining policy. A long-access sequence was split into multiple short subsequences, and each short subsequence was placed into a cutting window. Two independent caches were built into the clients, and the two policies were then applied to the caches. Finally, we applied the prefetching policy with lower cache misses for the next cutting window at the end of the current cutting window. For write performance, it is well known that the traditional data placement approach places a file by storing either a whole file or a sequence of fixed-sized blocks, and the written data must be sent to the data center that holds the original file/block. Consequently, a large amount of data must be transmitted over the network. This led us to reduce the amount of data transmitted over the network by placing a file using the method of storing one or more non-fixed-size data fragments. The written data fragments were placed in the local data center rather than being sent to the remote data center that holds the original file. The main contributions of this study are summarized as follows:

- A metadata prefetching approach is proposed for concurrent workload scenarios in a wide-area network. It dynamically changes prefetching policies based on workload characteristics. It tracks the cache misses of different prefetching policies to determine the prefetching policy; the prefetching policy with lower cache misses is selected as the optimal prefetching policy.
- An efficient data-placement approach is proposed for a wide-area network. It places the written data fragments in the local data center and then sends only the location information of the data fragments to the remote data center that holds the original file. Whenever data are read by clients, the requested data are stored on the client-side data center as a replica. Location information is stored as an extended attribute of the original file.
- Proposed metadata prefetching and data placement approaches were implemented on a newly designed wide-area virtual file system, which builds on existing file systems to provide unified storage. The benefits were evaluated by using different real system traces. The experimental results show that the access times of the file system metadata and the application data could be significantly reduced.

2 Related Work

Metadata access and data placement have long been studied in WANFSes to improve the overall performance of the system. This section reviews related work for prefetching policies and data placement.

2.1 Prefetching Policies

In wide-area computing environments, achieving a high cache hit rate is critical to the performance of storage systems. Several prefetching policies have been intensively explored; however, only a limited number of counterpart research proposals are for metadata in concurrent workload scenarios. These prefetching policies are based on non-frequency and frequency mining.

Non-frequency-mining-based prefetching (NFMBP) relies heavily on simple patterns, such as temporal locality, sequentiality, application informing, and file or block associations imposed by users, to improve storage system performance. ScaleXFS [15] and exF2FS [16] sequentially prefetch data blocks to overlap calculation and I/O. However, this type of prefetching approach may cause read amplification in concurrent workload scenarios including random access.

Lustre [4] prefetches all metadata in the same directory to quickly satisfy subsequent metadata access requests. The metadata server (MDS) pushes all file metadata in the requested directory to clients when directory metadata are requested by clients. However, they cannot capture the potential relationship between files across directories, leading to low access performance [17].

Application-informed prefetching [18,19] generally uses application hints to determine the data to be fetched in advance. However, incorrect prefetching policies may be selected when no appropriate application hints exist. Moreover, this approach generally involves source code revision.

Frequency-mining-based prefetching (FMBP) focuses on mining correlations and exploiting complex semantic patterns to fetch related information. For correlation mining, C-Miner [9] splits long-access sequences into multiple short subsequences and discovers frequent sequences using a data-mining technique to improve prefetching accuracy. Unfortunately, this approach maintains the latency caused by wide-area network communication because the prefetched data remain buffered on the local storage server.

For semantic pattern exploitation, Nexus [20] is a weighted-graph-based metadata prefetching method. In Nexus, a metadata relationship graph is constructed based on the access sequence to prefetch metadata in batches, where the vertices and weights of edges represent files and intimacy between files, respectively. The edges are dynamically inserted or removed by the MDS to improve the prefetching accuracy. Xu et al. [8] adopted a similar approach to mine correlations. Unfortunately, this type of prefetching approach may not be able to make correct decisions when encountering concurrent workload scenarios, particularly when there is insufficient knowledge of the access histories [10].

In summary, several prefetching policies have been proposed to improve the access performance of storage systems. These prefetching policies are efficient for specific scenarios, but most of them have made little attempt to improve the metadata prefetching accuracy for concurrent workload scenarios in wide-area computing.

2.2 Data Placement

In a data center, data can be easily placed with the use of parallel/distributed file systems (PDFSes), such as Tectonic [21] and Orion [22]. Data placement is much more complicated when storage resources are distributed across different institutions without a shared file system. Over the past ten years, typical methods for data transfer have been based on grid transfer, script transfer, and manual transfer [1]. This produces challenges, such as high delay, large network overhead, and more complex operations.

Peer-to-peer systems [23] provide alternative data placement solutions with high scalability. These systems hash pathnames to distribute files across all storage nodes of the file systems. No single-point bottleneck exists because it is eliminated using horizontal sharding. However, files might be placed far away from their most frequent or likely users.

Cloud storage systems provide another promising data-placement solution for wide-area computing. Wiera [24] is a geo-distributed cloud storage system that determines the best replica writing based on network dynamics and access dynamism. Wiera tends to place replicas at data nodes with the lowest round-trip time (RTT) from the client, highest network bandwidth, and shortest distance from their frequent users. Charon [25] is a multi-cloud storage system capable of storing and sharing big data in a secure, reliable, and efficient way using multiple cloud providers. Although partial write is provided by Charon, the written data must be sent to the remote cloud provider that holds the original data block. GeoCol [26] is a geo-distributed cloud storage system with low cost and latency using reinforcement learning. To achieve the optimal tradeoff between the monetary cost and the request latency, GeoCol encompasses a request split method and a storage planning method. In the storage planning method, each datacenter uses reinforcement learning to determine whether each data object should be stored and the storage type of each stored data object. In GeoCol, the access performance of data is compromised because the writing of data mainly considers the storage cost.

The data placement solutions of WANFSes are specifically proposed for wide-area high-performance computing. Lustre-WAN is a wide-area implementation of local cluster production as implemented in a local-area network [27]. The hierarchical persistent client caching (LPCC) mechanism can be used in Lustre-WAN to improve access performance [28]. LPCC builds a read-write cache for the local storage of a single client. Once a file is attached to LPCC, most data I/O is performed locally, thereby reducing the heavy network traffic. However, it is difficult to deploy LPCC to wide-area file systems that consist of heterogeneous file systems because: 1) Lustre-WAN requires every machine to implement the same file system and have a degree of trust between kernels; and 2) the hierarchical storage management of Lustre must be realized to support the operation of LPCC.

The data placement solution of Gfarm [12] is designed to speed up data access in a wide-area high-performance computing environment, where data center outages do not occur. Gfarm adopts a database to manage the data location information. Replica granularity is designed at the file level to reduce the management cost of data location information, and the replicas can be dynamically placed based on client locations or request rates. However, Gfarm builds replicas only for files that are frequently read, which does not benefit data writes. CalvinFS [13] places multiple data replicas in different datacenters for center-level disaster recovery. When the location of the computing centers is not given, the data replica cannot be placed in advance. Therefore, the CalvinFS client must send written data to a remote data node that holds a data replica.

In summary, several efficient data placement solutions have been proposed to improve remote access performance; however, they have not attempted to minimize the amount of data transmitted over the network for data writes in wide-area computing.

3 Proposed Approach for Metadata Prefetching

In this section, we propose a new approach for fine-grained metadata prefetching that adopts two different prefetching policies (i.e., NFMBP and FMBP) and dynamically applies the better policy based on workload changes. We adopt the existing directory-based prefetching approach [4] as the NFMBP policy. We propose a new prefetching policy based on a relationship graph that is employed as a frequency-mining-based policy. We describe how to select a better prefetching policy based on workload changes.

3.1 Frequency-Mining-Based Prefetching

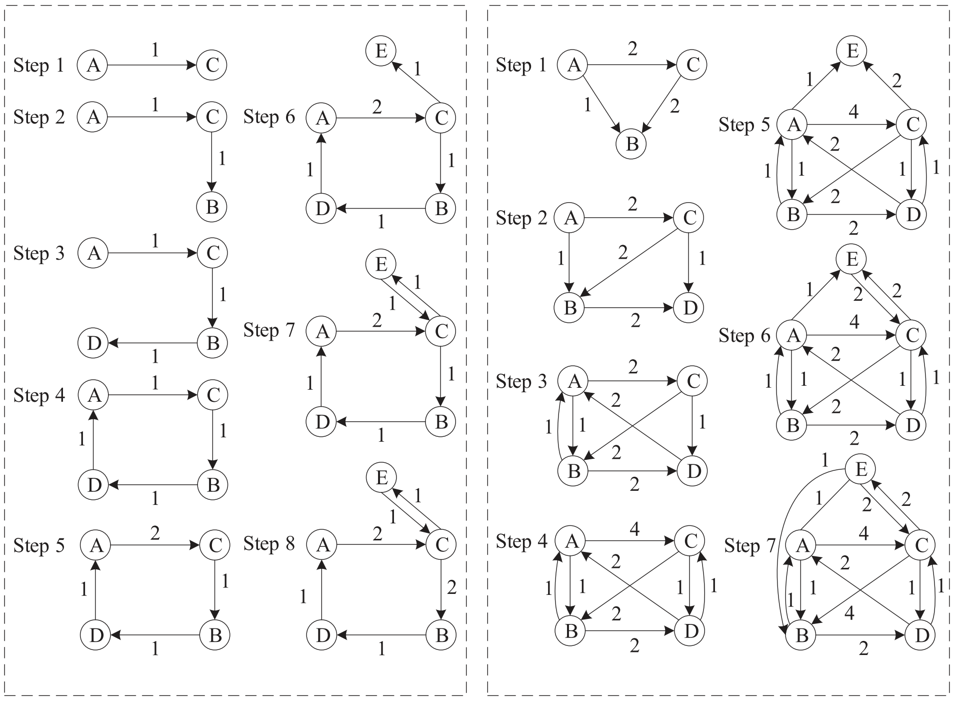
The proposed FMBP uses a relationship graph based on access frequency to capture file metadata correlations for prefetching. A relationship graph is a directed graph because the relationship between predecessors and successors in metadata access streams is essentially unidirectional. A vertex in the relationship graph represents a file and a weighed edge represents the locality strength between a pair of file items.

We describe the construction of a relationship graph to illustrate how it improves prefetching performance. The relationship graph is dynamically built by the clients. A look-ahead history window with a predefined capacity is constructed on the client to store the requests most recently issued by the applications. When no available capacity exists in the history window, the older metadata request is replaced by that of the newest. The locality strength information between the files in the history window is then integrated into the graph on a per-request basis. If all files in the history window can be discovered in the graph, an appropriate weight is added to the corresponding edge. Otherwise, a new edge is inserted.

For a better understanding, Fig. 2 shows the construction processes of the relationship graphs for the incoming metadata stream ACBDACECB. The history window sizes ($hWinSize$) in Fig. 2a,b are set to 2 and 3, respectively. Let *ord* (A) represent the order of A in the history window. The distance between A and B is definited as

$$distance(A, B) = |ord(A) - ord(B)|.$$

The weight values of *A* and *B* can be calculated using $hWinSize$ − *distance* (*A*, *B*).



(a) $hWinSize$ = 2 (b) $hWinSize$ = 3

Figure 2: Case study of relationship graph construction

After the relationship graph is constructed, it is updated online according to the incoming metadata stream. For example, in Fig. 2b, when the metadata stream ACB is incoming, the edges A→C, A→B, and C→B are 2, 1, 2, respectively. When the next metadata value D is incoming, metadata value A is evicted from the look-ahead history window and then D is added to the window, and the weights of the edges C→D and B→D are updated online to 1 and 2, respectively.

A prefetching procedure is evoked when a metadata request identifies a miss in the cache. The prediction items rely on the weights of the outbound edges originating from the currently missing element. The greater the weight of the edge, the higher the prefetching priority. Algorithm 1 shows how the relationship graph is traversed to obtain the prediction items. This algorithm considers the prefetching breadth (*brd*) and depth (*dph*) together. The algorithm prefetches *brd* nodes related to the currently requested item (lines 5–9). Further prefetching works on each item related to the currently requested item (lines 10–16). For each prefetching, the most relevant *brd* nodes are prefetched. In the relationship graph, the larger the weight of the edge, the more relevant the nodes. The weight of the edge is calculated using the distance between any two *hWinSize* items in the history window.

Algorithm 1: GPI (Get Prediction Items)

Input: Relationship graph: *G*; Currently requested item: *r*; Prefetching breadth: *brd*; Prefetching depth: *dph*.
Output: The queue that contains all prediction items: *rstQ*.

```
1: $itQ = \varnothing$;                                  ▷ Initialize the processing queue $itQ$.
2: EnQueue($itQ$, $r$);                                  ▷ Enqueue the requested item $r$.
3: $MaxItems = \frac{1-brd^{dph}}{1-brd}$;               ▷ Calculate the maximum number of predicted items.
4: while $itQ$ != $\varnothing$ do                       ▷ Process items in the queue:
5:     $p$ = DeQueue($itQ$);                             ▷ Dequeue an item.
6:     EnQueue($rstQ$, $p$);                             ▷ Enqueue the item to the result queue.
7:     Let $E$ denote edges from $p$ in $G$;             ▷ Get edges originating from $p$.
8:     Select top $brd$ edges by weight;                 ▷ Select edges with the highest weights.
9:     Let $S$ be the target nodes of selected edges;    ▷ Get target nodes.
10:    EnQueue($itQ$, $S$);                              ▷ Enqueue the selected nodes.
11:    if $rstQ$.count + $itQ$.count $\geq$ $MaxItems$ then   ▷ If the total reaches the limit:
12:        $pNum$ = $MaxItems$ - $rstQ$.count;           ▷ Calculate remaining items to add.
13:        Let $X$ denote the first $pNum$ items in $itQ$;   ▷ Select first $pNum$ items.
14:        EnQueue($rstQ$, $X$);                         ▷ Enqueue the remaining items to the result queue.
15:        break;                                        ▷ Exit the loop.
16:    end if
17: end while
```

3.2 Dynamic Prefetching

The proposed prefetching approach splits a long access sequence into multiple short subsequences and can identify a better prefetching policy. Similar to C-miner, it also uses a cutting window to store short subsequences. The storage system client builds two independent caches to determine the better prefetching policy. One of the caches, called *Real Cache* (*R-Cache*), stores real metadata information to satisfy the metadata requests of applications. The other cache, called *Comparison Cache* (*C-Cache*), does not store metadata information and serves only as a comparison with *R-Cache*. Both *R-Cache* and *C-Cache* use the key-value model to organize data. For *R-Cache*, the key denotes the file path, and the value contains complete metadata information. For *C-Cache*, the key also denotes the file path; however, the value contains only a small piece of metadata information, such as the file type.

The proposed approach determines the better prefetching policy for the next cutting window by comparing the cache misses of *R-Cache* and *C-Cache* in the current cutting window. At the end of the current cutting window (when all requests in the window have been sent), the prefetching policy with fewer cache misses is selected as the better prefetching policy for the next cutting window. The metadata prefetched by the better prefetching policy are cached in *R-Cache*, and the file path information fetched by the other prefetching policy is stored in *C-Cache*.

Algorithm 2 shows the fine-grained management of multiple prefetching policies. This algorithm splits a long access sequence into multiple short subsequences, and places each short subsequence into a cutting window. The parameter *rolls* denotes the number of the metadata request, and also represents the length of the long access sequence. When a metadata request arrives, *rolls* is incremented by 1. The parameter *cWinSize* denotes the cutting window size and also represents the size of each short subsequence. The cutting window is full when *rolls* % *cWinSize* is equal to 0 (line 1). This implies that the length of the current short subsequence reaches the threshold value. In this situation, the better prefetching policy is selected for the next cutting window (line 2). If FMBP is selected as the better prefetching policy, then the FMBP algorithm is performed; otherwise, the DBP algorithm is performed (lines 4–8). The cache misses of *R-Cache* (denoted as *rMiss*) can be calculated on the client side by counting the cache misses for each metadata request. If FMBP is selected as the better prefetching policy, the cache misses of *C-Cache* (denoted as *cMiss*) cannot be calculated in real time. This is because no prediction path information is available on the client side, and the client must interact with the I/O providers to obtain the path information.

Algorithm 2: FGP (Fine-Grained Prefetching)

Input: Currently requested item: *r*.
Output: Finish metadata prefetching.

```
1: if rolls % cWinSize == 0 then        ▷ Make a prefetching decision when the number of rolls is a
   multiple of cWinSize.
2:     Make prefetching decision and set bPol;
3: end if
4: if bPol == "FMBP" then               ▷ Use FMBP if policy is set to FMBP:
5:     bufQ = FMBP(r);
6: else
7:     DBP(r);
8: end if
9: if bufQ.count ≥ reqThreshold then    ▷ If the prefetching queue has enough items:
10:     if (rolls + 1) % cWinSize == 0 then    ▷ Perform synchronous prefetching:
11:         Access bufQ and prefetch metadata items if r is file;
12:         Insert items into C-Cache;
13:     else                            ▷ Asynchronous prefetching:
14:         Access bufQ asynchronously and prefetch metadata items;
15:         Insert items into C-Cache;
16:     end if
17:     Update cMiss with miss count from bufQ;
18:     sur = cWinSize - (rolls % cWinSize);
```

(Continued)

Algorithm 2 (continued)

```
19:     if rMiss > cMiss + bufQ.count + sur then          ▷ Assert if prefetching policy needs to change:
20:         bPolAssert = true;
21:     else if cMiss > rMiss + sur then                   ▷ Assert if prefetching policy needs to change:
22:         bPolAssert = true;
23:     end if
24: end if
25: rolls++;                                           ▷ Increment the roll counter.
26: H denotes the hWinSize items in history window;
27: for all items h in H do                            ▷ For each item h in history window:
28:     d = distance between h and subsequent items f;
29:     $g_{h,f}$ = weight of edge $h \rightarrow f$ in graph G;
30:     Update $g_{h,f}$ with $\frac{1}{d}$;     ▷ Update the relationship strength based on proximity.
31: end for
```

In Algorithm 2, a certain number of missed requests are buffered in *bufQ* rather than immediately sent to the I/O providers. If directory-based prefetching is selected, *cMiss* can be calculated in real-time. This is because prediction path information can be obtained from the relationship graph. If the current metadata request is not the last request of the current cutting window, then the requests buffered in *bufQ* are sent to the I/O providers asynchronously; otherwise, the prediction path information is fetched synchronously (lines 9–20). After the prediction path information is fetched, *cMiss* can be accurately calculated (lines 21–22). If the better prefetching policy is known prior to the end of the current cutting window, the calculation of *cMiss* can be cancelled (lines 24–28). If directory-based prefetching is selected as the better prefetching policy, then no item exists in *bufQ*. When a metadata request is satisfied, the weights of the edges related to the *hWinSize* items in the history window must be updated (lines 31–39). In the history window, the distance between any two items is calculated (line 35). This distance is used to update the weight of the relevant edge (lines 36–37). A relatively small *hWinSize* can lead to the loss of frequent subsequences that are split into two or more sequences, and therefore, can decrease the number of occurrences of some frequent subsequences because some of their occurrences are split. Therefore, it makes sense to increase *hWinSize* to achieve a high level of prefetching accuracy.

Algorithm 3: FMBP

Input: Currently requested item: *r*.
Output: Updated *rMiss*; The queue that contains all the missed request items of *C-Cache*: *bufQ*; Updated *C-Cache*; Updated *R-Cache*.

```
1: if !bPolAssert and !C-Cache.Contains(r) then
2:         if !bufQ.Contains(r) then
3:             EnQueue(bufQ, r);
4:         end if
5:     else if !bPolAssert then
6:         Move the hit item to the end of C-Cache;
7:     end if
```

(Continued)

Algorithm 3 (continued)

```
8:      if !R-Cache.Contains(r) then
9:          if r.filetype == regular file then
10:             rstQ = GPI(G, r, brd, dph);
11:         end if
12:         Synchronously access r;
13:         Asynchronously access all items in rstQ;
14:         Insert r, items, and subpaths into R-Cache;
15:         rMiss++;
16: else
17:         Move the hit item to the end of R-Cache;
18: end if
19: sur = cWinSize - (rolls % cWinSize);
20: if rMiss > cMiss + bufQ.count + sur then
21:         bPolAssert = true;
22: else if  cMiss > rMiss + sur then
23:     bPolAssert = true;
24: end if
```

Algorithm 3 shows the sequence of events for FMBP. If a requested item fails to be hit in *C-Cache*, it is temporarily buffered in *bufQ*. Otherwise, the hit item corresponding to the currently requested item *r* is moved to the end of *C-Cache* to prevent it from being evicted quickly (lines 1–7). Only the cache miss of a file (a regular file, not a directory) can invoke item prefetching, and the prediction items can be fetched according to the relationship graph (lines 8–11). The currently requested item *r* is fetched synchronously, whereas the prediction items are fetched asynchronously (line 12). Each time a miss occurs in *R-Cache*, *rMiss* is incremented by one (line 15). Whether the calculation of *cMiss* can be cancelled is determined at the end of the algorithm (lines 19–24).

Algorithm 4: DBP (Directory-Based Prefetching)

```
Input: Currently requested item: r.
Output: Updated cMiss; Updated rMiss; Updated C-Cache; Updated R-Cache.
1:  if !bPolAssert and !C-Cache.Contains(r) then
2:      cMiss++;                                 ▷ Increment miss count for the requested item.
3:      if r.filetype == regular file then
4:          rstQ = GPI(G, r, brd, dph);
5:          Insert all items of rstQ into C-Cache;
6:          Insert accessed subpaths into C-Cache;
7:          rstQ.clear();
8:      else
9:          Insert r and its subpaths into C-Cache;
10:     end if
11: else if!bPolAssert then
12:     Move the hit item to the end of C-Cache;
13: end if
14: if !R-Cache.Contains(r) then
```

(Continued)

Algorithm 4 (continued)

```
15:         Synchronously access $r$, insert $r$ into $R$-$Cache$;
16:         if $r$.filetype == regular file then
17:             Let $D$ be the directory that holds $r$;
18:             Synchronously access metadata items in $D$;
19:             Insert items and subpaths into $R$-$Cache$;
20:         end if
21:         $rMiss$++;
22:     else
23:         Move the hit item to the end of $R$-$Cache$;
24:     end if
```

Algorithm 4 presents the pseudocode for directory-based prefetching. If the currently requested item r fails to be hit in *C-Cache*, *cMiss* is incremented by 1 (line 2), and r and its related items are immediately inserted into *C-Cache* (lines 3–10). Only a cache miss of a file in *R-Cache* can invoke item prefetching. The related items in the same directory are prefetched, and *rMiss* is incremented by one whenever a miss appears in *R-Cache* (lines 14–21).

The execution of *C-Cache* can introduce additional overhead. Assume the number of metadata requests issued by applications is n and the maximum directory depth of the accessed files is d. For each metadata request issued by applications, Algorithm 3 or Algorithm 4 is employed by Algorithm 2 to satisfy the request. In Algorithm 3, the running time of *C-Cache* is bounded by $O(1)$ with the help of hash tables and linked lists (lines 1–7). In Algorithm 4, the running overhead of *C-Cache* is dominated by lines 4–6. The GPI algorithm is employed to update *C-Cache*, and its running time is bounded by $O(brd^{dph})$. The running time for inserting the relevant items into *C-Cache* is bounded by $O(d \cdot brd^{dph}/brd)$ (lines 5–6). In Algorithm 2, the running overhead of *C-Cache* is dominated by lines 11–15, and the running time of *C-Cache* is bounded by $O(d \cdot reqThreshold)$. Therefore, the running time of *C-Cache* is bounded by $O(n \cdot (brd^{dph} + d \cdot brd^{dph}/brd + d \cdot reqThreshold))$.

Table 1 provides a clear taxonomy of the evaluated methods. Most existing strategies are static in nature and focus on either structural (DBP) or statistical (FMBP, NABP) aspects. FGP is unique in that it integrates both structural and statistical features and dynamically chooses the more effective policy, achieving robust performance across diverse workloads.

Table 1: Overview of prefetching strategies and their characteristics

Method	Graph-based	Dynamic	Freq. mining	Dir-based	Key features
NP (No Prefetching)	No	No	No	No	Baseline approach; no prefetching logic.
WGP	Yes	No	No	No	Uses weighted access graph but static; lacks adaptability.
FMBP	Yes	No	Yes	No	Builds relationship graph and mines frequent metadata correlations.
DBP	No	No	No	Yes	Prefetches all metadata in directory; static policy.

(Continued)

Table 1 (continued)

Method	Graph-based	Dynamic	Freq. mining	Dir-based	Key features
NABP	No	No	Yes	No	Uses Apriori-style itemset mining; not real-time suitable.
FGP	Yes	Yes	Yes	Yes	Dynamically switches between DBP and FMBP based on cache miss stats.

Note: *Summarizes the core characteristics of each metadata prefetching method. As shown, only the proposed FGP approach is both dynamic and capable of combining frequency-mining-based learning and directory-based structure exploitation, while adapting online based on workload changes.

3.3 Rationale for Combining DBP and FMBP

The motivation for combining the directory-based prefetching policy (DBP) and the frequency-mining-based prefetching policy (FMBP) stems from their complementary strengths in handling diverse workload characteristics. DBP is particularly effective when files in the same directory exhibit strong spatial locality and are accessed together within a short time interval. This pattern is commonly observed in workloads generated by structured applications such as software builds or scientific computing, where directory structures are tightly correlated with access patterns.

In contrast, FMBP excels in capturing deeper access correlations that are not necessarily constrained by directory hierarchy. By constructing a relationship graph and mining frequent access sequences, FMBP can identify more complex temporal patterns in metadata access. This makes it especially effective for workloads with weaker or irregular directory-level locality, such as cloud service workloads or user-driven exploratory data access.

However, each strategy also has limitations. DBP may suffer from prefetching low-utility metadata when directory-level correlations are weak, leading to cache pollution. On the other hand, FMBP requires sufficient historical data to accurately model relationships and may underperform at the beginning of an execution phase or when the access patterns are highly dynamic.

To address these issues, our approach dynamically switches between DBP and FMBP based on real-time workload feedback, as described in Algorithm 2. During the early stages of access or when directory-level locality dominates, DBP is selected to quickly leverage known structural relationships. As access patterns evolve and higher-order correlations become more prominent, FMBP is favored due to its ability to exploit deeper associations.

A small case study in Fig. 2 illustrates the effectiveness of FMBP in capturing recurring subsequences in access streams, even when files are scattered across multiple directories. Meanwhile, early results shown in 6.2.1 demonstrate that DBP performs better in the initial stages of execution, while FMBP shows superiority later on. This observation supports the rationale for our adaptive hybrid design.

By integrating both strategies into a fine-grained prefetching framework (FGP), our method can leverage the strengths of each under varying conditions, achieving consistently high hit ratios and reduced execution time across diverse workloads.

3.4 Explanation and Tuning of Key Parameters

To ensure effective metadata prefetching, several key parameters are introduced in the proposed approach. This section explains the physical meanings, selection rationale, and tuning strategies for the parameters *brd*, *dph*, *cWinSize*, and *hWinSize*.

- **Prefetching breadth (*brd*)** determines the number of immediate neighbor nodes selected from the relationship graph during each round of traversal. A larger *brd* allows the prefetching mechanism to capture more potentially related metadata items but may introduce irrelevant data and increase cache pollution. Experimental results (6.2.2) show that setting *brd* between 8 and 12 achieves a good trade-off between accuracy and overhead.
- **Prefetching depth (*dph*)** defines how many levels ahead the graph traversal should go when predicting future metadata access. A small value (e.g., $dph = 1$) focuses on direct correlations but may miss long-range associations. A large value increases the prediction horizon but risks introducing noise. Our evaluations (6.2.2) suggest that a depth of 2 or 3 balances prediction accuracy and runtime overhead.
- **Cutting window size (*cWinSize*)** specifies how many metadata requests are processed before the system evaluates the effectiveness of the current prefetching policy. A smaller *cWinSize* enables finer-grained adaptation to workload changes, while a larger value reduces decision-making frequency and stabilizes performance. As shown in 6.2.2, the optimal value depends on workload dynamics, with a moderate size (e.g., 2000–10,000) being generally effective.
- **History window size (*hWinSize*)** controls the number of recent metadata requests considered when constructing the relationship graph. A small *hWinSize* captures only immediate successors, limiting the mining of frequent access patterns. A large value may introduce noise and reduce specificity. 6.2.2 shows that values between 10 and 20 provide a suitable balance between precision and coverage.

In practice, these parameters can be initially set to default values based on empirical evidence (e.g., $brd = 12$, $dph = 3$, $hWinSize = 20$, $cWinSize = 1000$), and then fine-tuned based on system profiling or offline simulation. Furthermore, adaptive tuning mechanisms could be introduced as future work to dynamically adjust these values based on observed cache hit rates and workload characteristics.

4 Variable-Size Data Fragments

Most WANFSes place a file by storing a whole file or a sequence of fixed-sized blocks. Unlike the traditional approach, the proposed approach places a file by storing one or more data fragments, with the fragment size not being fixed. The fragment size may range from 1 byte to the entire file. In addition, fragments are not immutable once written and any byte in a fragment can be overwritten. When appending data to a file, the proposed approach appends it to the local fragment file if it exists. Otherwise, a new fragment is created in the local data center to store the appended data. Clients can merge small data fragments into large fragments to reduce the number of file fragments, and a garbage collector is created to compact the data fragments.

Fragment Storage and Placement

When a client writes data, the data are stored in the local data center. The client first writes the data synchronously to the local data center and then sends information about the offset, size, and local data center to the remote data center that holds the original file. When receiving information from the client, the location information holder updates the distribution information in byte ranges. The data read by a client are also stored in the client-side data center as a replica.

Algorithm 5: FGDP (Fine-Grained Data Placement)

```
Input: File offset: off; Number of bytes to write/read: size; Center id: cid; Operation type: opType; Byte range graph array: brgArr.
Output: Finish the writing of byte ranges.
1: $Ibr = [off, off + size]$;    ▷ Calculate the byte range Ibr using the offset off and the number of bytes size.
2: if opType == "write" then    ▷ If the operation type is "write":
3:     for all items brg in brgArr do    ▷ Iterate over all byte range graph items in brgArr:
4:         if brg.CenterId == cid then    ▷ If the center id of the byte range graph item matches the given cid:
5:             $brg.Ranges = brg.Ranges \cup Ibr$;    ▷ Add the byte range Ibr to the item's range.
6:         else    ▷ If the center id does not match:
7:             $rBR = brg.Ranges \cap Ibr$;    ▷ Calculate the intersection of the current item's ranges and Ibr.
8:             $brg.Ranges = brg.Ranges - rBR$;    ▷ Remove the intersecting byte range from the item's ranges.
9:         end if
10:    end for
11: else if opType == "read" then    ▷ If the operation type is "read":
12:    for all items brg in brgArr do    ▷ Iterate over all byte range graph items in brgArr:
13:        if brg.CenterId == cid then    ▷ If the center id of the byte range graph item matches the given cid:
14:            $brg.Ranges = brg.Ranges \cup Ibr$;    ▷ Add the byte range Ibr to the item's range.
15:        end if
16:    end for
17: end if
```

Algorithm 5 describes how byte ranges are obtained for a file. Fig. 3 shows an example of fragment storage and placement. The incoming data stream describes the sequence of operations: 1) a 10-MB file is completely written in data center A; 2) a data fragment ranging in size from 4 to 10 MB is read in data center B; 3) a data fragment ranging from 0 to 6 MB is read in data center C; 4) a byte range of 4–6 MB is written in data center D; 5) a byte range of 5–7 MB is written in data center D; and 6) a data fragment ranging from 4 to 7 MB is read in data center A. At T1, the client of data center A writes the complete file content, and no data replica is placed in the other data centers (lines 4–5). At T2, a data fragment is read by the client of data center B, an implicit data replica is placed in data center B, and the data replica is created based on the requested data fragment (lines 13–14). At T3, another implicit data replica is created in data center C when a data fragment is read by the client of data center C (lines 13–14). At T4, the data fragment ranging from 4 to 6 MB is updated by the client of data center D, the data fragment is directly written to the local data center (lines 4–5), and the byte ranges in the other data centers are invalidated (lines 7–8). At T5, the data fragment ranging from 5 to 7 MB is updated by the client of data center D, and a large data fragment ranging from 4 to 7 MB is created in data center D (lines 4–5). At T6, the data fragment ranging from 4 to 7 MB is read by the client of data center A, and then the data stored on data center D is transferred to data center A (lines 13–14).

As we can observe, for the data written locally, the later reads from the clients of other data centers can introduce time-consuming WAN data transfers. The proposed approach works well for the scenario where data are repeatedly written or read multiple times by clients located in the same data centers, as it reduces the need for cross-data-center data transfers. However, it works poorly for the scenario where data are written or read once by clients located on different data centers. In such cases, the overhead of transferring large data fragments over the WAN can significantly impact performance, particularly when the network bandwidth is limited, leading to higher latencies and slower access times.

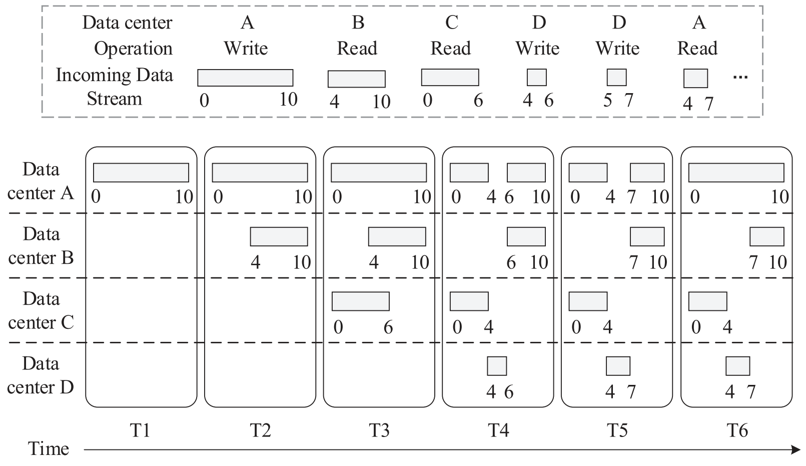


Figure 3: An example for fragment storage and placement

This observation highlights a potential limitation of the proposed approach: while it optimizes data placement and replication within a single data center, it does not fully address the challenge of efficient data transfer across geographically distributed data centers. In scenarios where data is written to one data center and read by clients in another, the current approach could result in unnecessary delays due to the time-consuming WAN transfers. As the network latency increases with geographical distance, the performance benefits of local data placement may be negated by the overhead introduced during cross-data-center data reads.

Moreover, for a given data center, all data fragments of the same file are stored in the same fragment file. For example, at time T4, the two data fragments located in data center A are stored in the same fragment file. This organization simplifies the storage and retrieval process, as the related data fragments are stored together. However, invalid data are not physically deleted immediately; instead, updating location information can bypass the invalidated data. This approach minimizes the immediate overhead of data deletion, allowing for more efficient use of storage space.

When the garbage collector performs compaction, invalidated data are physically deleted, but this process introduces additional overhead. The compaction process is essential for maintaining efficient storage utilization, but it can also impact system performance, especially if the system is under heavy load or if large volumes of data need to be compacted frequently. Future work could explore methods to optimize the garbage collection process to reduce its impact on system performance.

In summary, while the proposed approach provides significant improvements in scenarios with high-frequency data access within a single data center, it faces challenges when dealing with data access across multiple geographically distributed data centers. To address these challenges, future improvements could focus on optimizing cross-data-center data transfer and enhancing the garbage collection process to ensure that the system remains efficient in diverse usage scenarios.

5 Implementation

We implement the proposed metadata prefetching and data placement approaches on the newly designed wide-area virtual file system called WAVFS. The design of WAVFS refers to our proposed GVDS [29] that is designed for wide-area high-performance computing environments. Storage of WAVFS is a federation of PDFSes of geographically distributed data centers. It is built on existing systems such as Lustre [4] and Fisc [30] to provide unified storage. WAVFS actually federates not an entire file system but a particular directory of an existing PDFS. A newly initialized PDFS joins WAVFS to provide storage services by registration. The registration information contains file system locations, exposed storage directories, storage capacity, I/O providers, etc. A single central manager (CM) is used in WAVFS to manage all register information in a database.

5.1 *WAVFS Architecture*

Fig. 4 shows the WAVFS architecture. The CM manages information related to registered PDFS, containers (logical spaces that hold data, as detailed in next subsection), I/O providers, and users by the distributed database, which is deployed in a specified data center. I/O providers provide clients with remote access services for metadata and data. Clients export the system interface.

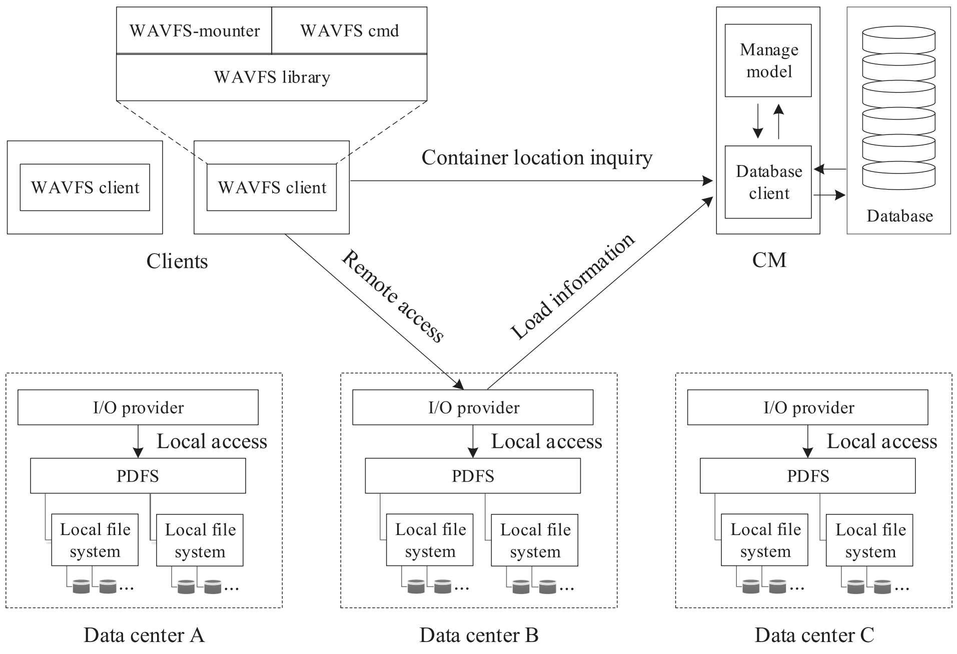


Figure 4: WAVFS architecture

Central manager. During file access, clients must communicate with the CM for container location information. Initially, a client is mounted on a computing node; it immediately interacts with the CM to obtain the location information of the containers that are related to the current user. Later, the client caches the requested information. Clients send the file path to the I/O provider to access metadata and data. I/O providers periodically send heartbeats to the CM to confirm that the I/O provider is alive and that the access services it hosts are available. Heartbeats from an I/O provider also carry information about the load of the I/O provider, total storage capacity, and the amount of storage in use of the registered PDFS. These statistics are useful when allocating I/O providers and storage space for containers. If the CM does not receive a heartbeat from an I/O provider in a certain amount of time, the CM considers the I/O provider to be out of service. The alternate I/O provider is then activated to provide services.

I/O provider. Clients directly connect to an I/O provider when accessing file data. An I/O provider performs real file access for clients. The I/O provider first opens the file at the file path, then seeks the specified offset, and finally performs the actual reads/writes. For load balancing, multiple I/O providers are deployed in every data center. CM chooses an I/O provider for a container based on the load status of the I/O provider. Frequently opening files can seriously degrade access performance. I/O providers use hash tables to cache file descriptors to solve this problem, and the key is a unique string composed of process information and file paths. If the target file descriptor cannot be found in the hash table, then the I/O provider fetches the file descriptor by opening the file and finally inserts the file descriptor into the hash table. The file descriptors that are evicted from the hash table must perform close operations. I/O providers cache a certain amount of data and metadata to reduce the number of disk accesses.

Client. A WAVFS client consists of the following three modules: 1) WAVFS library, 2) WAVFS mounter, and 3) WAVFS command. The WAVFS library is an implementation of the API to access files. Standard file operations such as open, stat, seek, write, read, and close are included. The WAVFS library also contains some extended operations such as container creation and deletion. The WAVFS mounter is a program to mount WAVFS in the user space by using a file system in user space (FUSE) mechanism. WAVFS provides commands such as PDFS management, container management, and data access. Initially, a client connects to the CM and requests the location information of the containers that are related to the current user. When opening a file of a container, the open request is sent to the corresponding I/O provider. The I/O provider opens the file and caches the file descriptor and then returns the open result (success or failure). If the file is successfully opened, then the local file descriptor is pointed to a file description structure by FUSE. Otherwise, an error code is returned. When accessing a file, clients send the file path to the I/O provider to access the data.

Access control

WAVFS achieves secure data sharing and access through access control. The concepts of user, group, and container are introduced in WAVFS to achieve access control. These concepts are described in detail as follows.

User. Each WAVFS user has an account containing basic personal information such as username, password, associated containers, and associated groups. A user can be associated with multiple containers. For example, in Fig. 5, *user3* is associated with *container1* and *container2*.

Group. Groups are introduced to achieve batch settings of access permissions for containers. The creator of a group is the group leader, which is responsible for managing the members. In a group, the group leader has the complete permissions, all members except the leader share the same access permissions. A group can be associated with one or more containers. For example, in Fig. 5, *user1* and *user2* are the members of *group1*, *group1* is associated with *container1*.

Container. WAVFS uses containers to store user data, containers are logical spaces that hold data. A container is mapped to one or more directories that is placed in different PDFSes. The directories mapped to a container can be classified as primary and secondary. Primary and secondary directories can be used to store file fragments, but the fragment location information is stored only in the extension attribute of the files placed in the primary directory. When a user creates a new directory/file in a container, a new directory/file is also created in the corresponding primary directory. The secondary directory only stores file fragments.

For example, in Fig. 5, *container1* is supported by *I/O proxy1* and *I/O proxy2*, the primary directory and the secondary directory related to *container1* can be created on *PDFS1* and *PDFS2*, respectively.

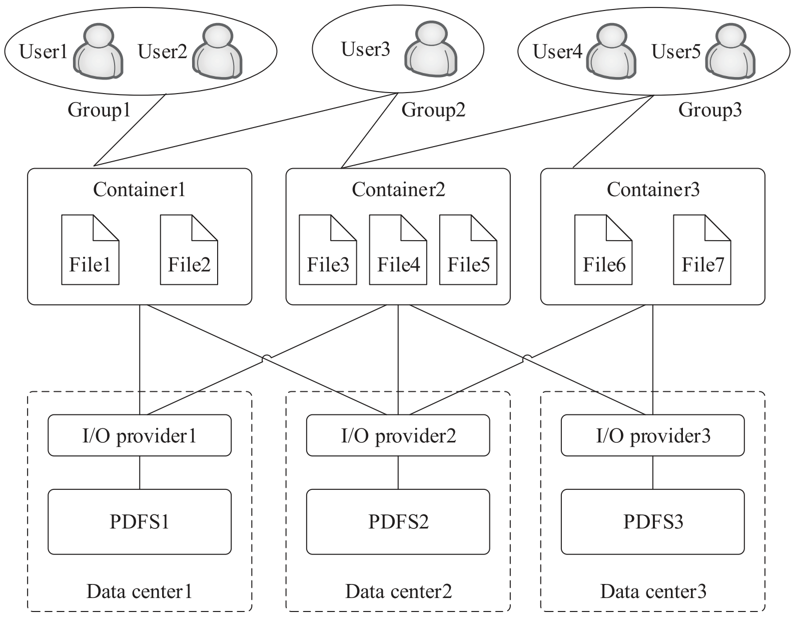


Figure 5: Secure data sharing and access

With the help of user, group, and container, the heterogeneous storage resources distributed in different data centers can be easily shared and access. The users of a group can share the data stored in the container associated with the group, they cannot access the data stored in the container that is not associated with the group. For example, in Fig. 5, *user1* and *user2* are the members of *group1*, they can share the data stored in *container1*; *user1* and *user2* cannot access the data stored in *container2* and *container3*.

5.2 Metadata Access and Prefetching

Fig. 6 shows the metadata access flow and prefetch timing in WAVFS. A metadata request *stat(f1,buf)* is sent from the application. The WAVFS client immediately directs the request to the I/O provider. At the same time, the proposed prefetching approach deployed on the client is invoked to predict metadata access patterns based on historical access information. The client prefetches the metadata asynchronously. Therefore, the previous normal metadata requests can be fulfilled without being blocked.

Both the required and prefetched metadata are cached on the client. The application can perform other operations after receiving the required metadata. When the application sends another metadata access request *stat(f2,buf)* to the client, the client can respond to the request with the cached metadata. As a result, the access request can be handled immediately and network communication cost can be reduced significantly. Although the proposed metadata prefetching approach mainly focuses on the scenarios of data reads, we also consider the consistency maintenance of metadata. In order to ensure the consistency of cached metadata, the I/O provider must record which metadata have been cached. When the cached metadata are updated by a client, the I/O provider must notify the other clients to invalidate the cached metadata.

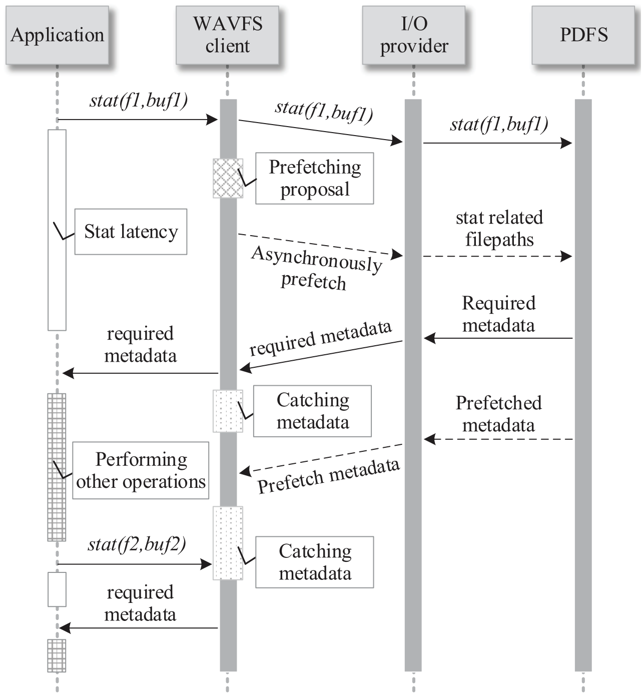


Figure 6: Metadata access flow and prefetch timing

5.3 Data Fragment Placement

To improve write performance, all file data are written into the primary/secondary directory of the local PDFSes that have been registered in WAVFS. Only the location information of the file fragments must be remotely written into the primary directory of the PDFS that holds the original file. If no original file exists in the primary directory, a new empty file (original file) is created in this directory. If no local PDFS is available, the data are written back to the remote file systems synchronously. Under this design, the number of file fragments increases over time. If all location information of these file fragments is held by a single central server, scalability problems will arise.

WAVFS adopts a two-level metadata hierarchy management to address this problem. The first level contains the location information of the container that holds the requested file. The second level contains the location information of the requested file fragment, and the file fragment locations are stored in extended attributes. No scalability limitation nor single-point performance bottleneck exists in the metadata management of WAVFS, because fragment location information is distributed in different PDFSes.

6 Performance Evaluation

6.1 Experimental Setup

6.1.1 Implementation and Configuration Details

All implementations were developed in C++ on Ubuntu 18.04 LTS, using the FUSE (Filesystem in Userspace) framework for WAVFS user-space mounting and metadata access interception. The WAVFS client modules were compiled with `g++` 7.5.0, and the I/O provider and central manager services were implemented as multi-threaded daemons with asynchronous I/O support using `libaio`.

Each virtual machine in Alibaba Cloud is configured with two vCPUs, 16 GB of RAM, and a 100 GB local SSD, running Ubuntu 18.04 with kernel version 4.15.0. The client machines in Guangzhou use Intel Core i7-9700 CPUs, 16 GB of RAM, and 1 TB HDDs, also with Ubuntu 18.04 and kernel version 5.4.0.

The metadata cache on the client side is managed using the LRU (least recently used) eviction policy. The cache is implemented using a hash table with a doubly-linked list to ensure $O(1)$ lookup and eviction operations. The cache size *cSize* is configurable and is set to 10,000 metadata entries in all experiments unless otherwise noted. Each entry contains the path, file type, and necessary stat-like metadata.

Prefetching parameters such as *brd*, *dph*, $cWinSize$, and $hWinSize$ were selected based on empirical testing and micro-benchmarking on the testbed. For example, *brd*=12 and *dph*=3 were chosen to ensure a balanced trade-off between graph traversal cost and prediction coverage. $cWinSize$ was initially set to 1000 based on observed workload fluctuation periods. We also performed parameter sweeps (as shown in 6.2.2) to validate the impact of each parameter and fine-tune accordingly.

WAVFS integrates the proposed metadata prefetching strategy by intercepting metadata-related calls (e.g., `stat`, `lstat`, `open`) through FUSE, and triggering prefetch prediction in parallel with I/O requests. All metadata are stored in extended file attributes in Lustre or local PDFS backends, and the container structure is maintained through a lightweight SQLite database at the CM side.

All experiments are conducted at three sites of the Alibaba Cloud and at a personal site in Guangzhou (GZ). The three sites of the Alibaba Cloud are located in BJ, SH, and Chengdu (CD). Each site in the Alibaba Cloud consists of three virtual machines, where each machine is configured with two vCPUs, 16-GB memory, 100-GB disks, and an Ubuntu 18.04 LTS operating system. The site in GZ consists of two physical machines, each of which is configured with 8-core 3-GHz Intel(R) Core(TM) i7-9700 CPUs, 16-GB memory, 1-TB disks, and an Ubuntu 18.04 LTS operating system. The network latency and the network bandwidth between any two sites are listed in Tables 2 and 3, respectively.

Table 2: Network latency (in ms) between any two sites

Site	BJ	SH	CD
BJ	–	25.42	59.32
SH	24.70	–	37.78
CD	59.32	37.78	–
GZ	39.58	29.86	

Table 3: Network bandwidth (in MB/s) between any two sites

Site	BJ	SH	CD
BJ	–	12.37	12.05
SH	12.31	–	12.25
CD	12.05	12.25	–
GZ	12.14	12.36	

6.1.2 Experimental Setup of Metadata Prefetching

File systems, including local file systems and wide-area file systems, usually provide unified interfaces for applications to access data. The type of the underlying file system is transparent to the application. Each record of a trace is the individual I/O commands issued by the applications. Therefore, there is no strong correlation between traces and the file systems. In this study, we replay typical traces on the wide-area file system to simulate application execution on the wide-area file system. However, the proposed approaches are not limited to specific traces.

We use real-world storage system traces oltp-trace1 [31], oltp-trace2 [31], search-trace [31], and cloud-trace [32] to evaluate the proposed metadata prefetching approach. The first three traces are provided by the storage performance council of Hewlett-Packard and International Business Machines Corporation, and they are collected from online transaction processing applications and a popular search engine. The last trace is provided by a popular cloud service company, and it is created by capturing the cloud service. The fields of these traces are listed in Table 4. Similar to the evaluation methodology described in [33], we replay a portion of each trace (i.e., the first 15,000 records) because the number of trace records is quite large.

Table 4: Fields of oltp-trace1, oltp-trace2, and search-trace

Field	Description
Timestamp	Offset in seconds from the start of the trace
LBA	ASU block offset of the data transfer
Size	Number of transferred bytes
Opcode	"R" is read; "W" is write
ASU	Application specific unit

Considering that the directory entries corresponding to the files placed in the same directory are sequential [4,8], we create an empty file (we focused only on metadata access) for every logical block address (LBA) in the disk, and the filename is the same as the LBA. In other words, we convert the relevance of consecutive data blocks to the relevance of adjacent files in directories. Many studies [34,35] have placed files based on a specific probability distribution. Similar to these studies, we place files into different directories according to the LBA with uniform distributions. Considering that the average directory depth of the fserver-trace [36] is approximately 3, we set the directory depth to 3. Therefore, the access of each file involves three metadata access operations.

We selected four real-world metadata access traces—`oltp-trace1`, `oltp-trace2`, `search-trace`, and `cloud-trace`—to evaluate our metadata prefetching strategies. These traces are chosen to represent diverse and realistic access patterns typically encountered in large-scale storage systems:

- **OLTP traces (oltp-trace1 and oltp-trace2):** These traces are generated by online transaction processing systems, which typically exhibit high concurrency, fine-grained metadata access, and strong locality. They are representative of enterprise database systems or financial transaction workloads. Such workloads often contain repeated access patterns to files in related directories, which are suitable for directory-based prefetching methods.
- **Search trace:** This trace originates from a popular search engine platform. Compared with OLTP workloads, search workloads exhibit complex metadata access sequences with long-range correlations and weaker directory locality. This makes them more challenging for traditional prefetchers, but well-suited to evaluate frequency-mining-based strategies that can capture non-local access patterns.
- **Cloud trace:** This trace reflects cloud service environments where users interact with shared storage in dynamic and less predictable ways. The access pattern is often hybrid, involving both sequential and

random accesses, with varying degrees of locality. This trace helps assess the adaptability of prefetching algorithms under mixed and fluctuating workloads.

These datasets collectively allow us to assess the effectiveness and robustness of the proposed approach under different workload characteristics. By combining OLTP and cloud traces (OC1, OC2) or search and cloud traces (SC), we simulate realistic multi-tenant wide-area file system scenarios with varying degrees of metadata access regularity and complexity.

We evaluate the prefetching performance in the following three scenarios: 1) replaying oltp-trace1 and cloud-trace (OC1); 2) replaying oltp-trace2 and cloud-trace (OC2); 3) replaying search-trace and cloud-trace (SC). Each trace is replayed by a client process, traces are replayed simultaneously by multiple client processes. The experimental parameters are set as follows: $brd = 12$, $dph = 3$, $hWinSize = 20$, $cWinSize = 1000$, $reqThreshold = 100$, $cSize = 10{,}000$, where *reqThreshold* is the threshold number of items stored in *bufQ* and *cSize* is the number of metadata items stored in client-side cache.

The state-of-the-art prefetching approaches are provided for fair comparison. We implement six different prefetching approaches into WAVFS for fair evaluations, including: 1) no prefetching (NP) [2,3]; 2) weighted-graph-based grouping prefetching (WGP) [20]; 3) FMBP; 4) DBP [4]; 5) new apriori-based prefetching (NABP) [17]; and 6) FGP. We use the least recently used algorithm for cache replacement. The server-side software components of WAVFS are deployed in SH and CD, CM is deployed in SH. BJ is deployed with a WAVFS client.

6.1.3 Experimental Setup of Data Placement

We use the real-world trace fserver-trace from Microsoft production servers [36] to evaluate the proposed data placement approach. Writes dominate in this trace because our approach is designed for write-dominant scenarios. The fields of the trace are shown in Table 5. We create a directory for every disk to emulate disk access. The experiments involve the following types of operations: 1) data collection and storage: each physical machine in GZ is deployed with one client process, which writes data into WAVFS. The 27.69 GB of data involved in the first 2,820,213 records of fserver-trace are written into 20,000 files, from which two sets of 10,000 files are created and then written by the two client processes, respectively; and 2) trace replaying: the first 200,000 records (only some of the datasets is required) of fserver-trace are simultaneously replayed by N client processes as N is varied. The machine on which a client process is to be placed is determined by round-robin scheduling. These records are assigned to the N client processes by a central scheduler; this scheduler assigns the next available record to a client process as soon as the previous record assigned to it completed processing.

Table 5: Fields of fserver-trace

Field	Description
Timestamp	Time from start of trace in microseconds
Offset	Offset of request from start of file in bytes
IOSize	Size of request in bytes
DiskNum	Physical disk number
FileName	Name of the file
Operation	File writes/reads

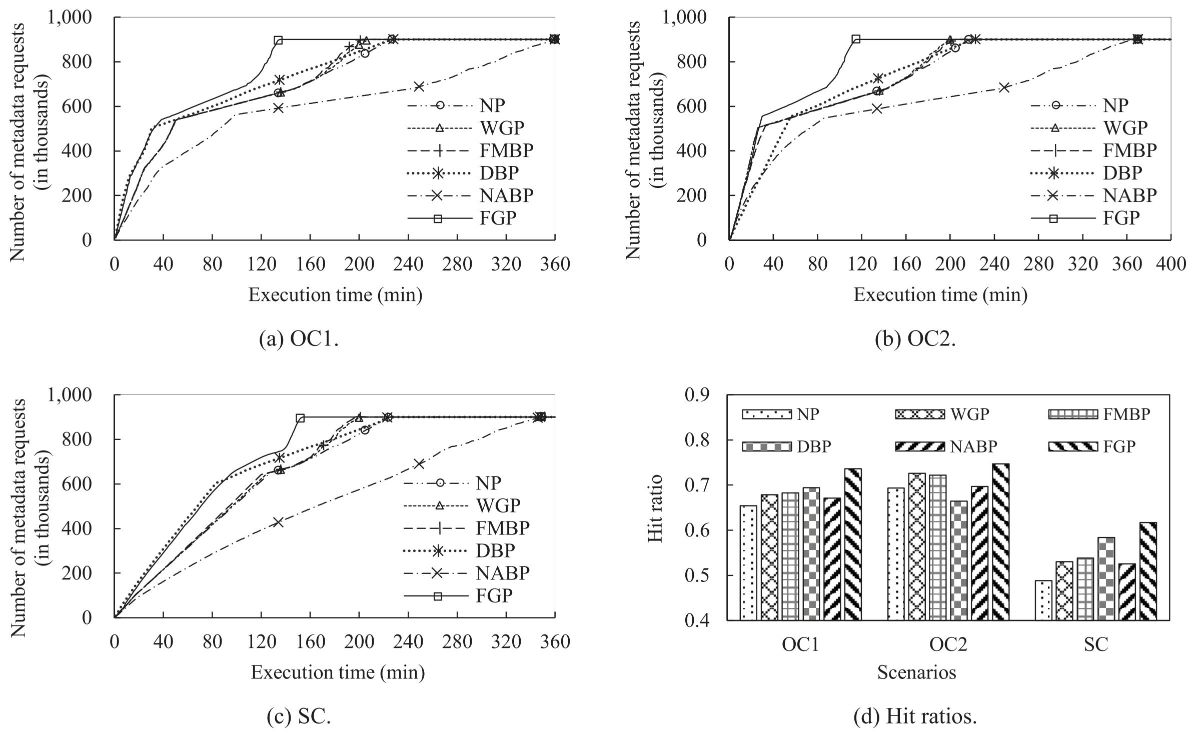
The data placement approaches of the state-of-the-art storage systems are provided for fair comparison. We implement four different data placement approaches into WAVFS for fair evaluations, including: 1) writing a whole file to a specified location (PWFSL) [2,3]; 2) determining the best replica placement based on network dynamics and access dynamism (DORP) [24]; 3) placing multiple data replicas in different data centers (PDRD) [13], we place a data replica in each data center; and 4) FGDP.

For data collection and storage, the server-side software components are deployed in SH and CD. The client-side software components are deployed in GZ. For the trace replaying stage, trace records are replayed in BJ. The server-side software components are deployed in SH, CD, and BJ, and the client-side software components are deployed in BJ.

6.2 Evaluation of Metadata Prefetching

6.2.1 Execution Time

Fig. 7a–c shows the number of metadata requests served by different approaches over time for scenarios OC1, OC2, and SC, respectively. Fig. 7d shows the hit ratios for different trace-replay scenarios. Regarding scenario OC1, when the execution time is less than 36 min, DBP achieves the best prefetching performance, and the number of requests served by DBP is significantly greater than that of NP, WGP, FMBP, and NABP. FGP also achieves a high level of prefetching performance because it inherits the advantages of DBP. When the execution time is approximately 40 min, no significant differences are observed between DBP and NP in terms of the number of requests. This behavior occurs because a large amount of metadata prefetched by DBP is not accessed. For this situation, FGP uses a frequency-mining-based policy to prefetch metadata. Therefore, the number of metadata requests served by FGP at the same execution time is higher than that of DBP.



(a) OC1.
(b) OC2.
(c) SC.
(d) Hit ratios.

Figure 7: Number of metadata requests served by different approaches over time, and hit ratios for different trace-replay scenarios

In Fig. 7a,b, there is a big change around time 40 min. This is because the associativity of the file decreases after 40 min, and many subsequent metadata requests cannot be satisfied by the prefetched metadata stored in the client-side cache. The file system client has to interact with the central manager through wide-area network to access the requested metadata. This leads to long execution time. As the execution time increases from 80 to 140 min, DBP achieves better prefetching performance than NP, WGP, and FMBP. Beyond 160 min, the number of requests served by WGP and FMBP improves significantly. The FGP is able to take advantage of the non-frequency-mining-based and frequency-mining-based policies, thereby achieving the lowest execution time.

When the replaying of OC1 is fulfilled, the order of hit ratios is: FPG > DBP > FMBP > WGP > NABP > NP, where higher is better. The order of execution time is: FGP< FMBP< WGP< NP< DBP< NABP, where lower is better. The FGP achieves the best prefetching performance, improving the hit ratios by as much as 8.17%, 5.76%, 5.31%, 4.16%, and 6.47% as compared to those of NP, WGP, FMBP, DBP, and NABP, respectively. These correspond to 40.27%, 35.1%, 33.5%, 40.53%, and 62.08% reductions in execution time, respectively. FMBP is slightly better than WGP in terms of hit ratio and execution time. Initially, no significant differences are observed between NP, WGP, and FMBP in terms of the number of requests. This is because historical access information is not sufficient to learn. When the relaying is fulfilled, the cache hit ratio and execution time of FMBP and WGP are better than those of NP. The cache hit ratio of DBP is higher than that of NP, but the execution time is lower. NABP achieves a higher hit ratio than NP, but its execution time is significantly higher than that of NP. This is because NABP requires considerable time to perform computations for capturing associated files. This phenomenon also demonstrates that NABP is unsuitable for real-time prefetching.

Regarding scenario OC2, when the execution time is less than 55 min, the number of requests served by FMBP is greater than that of DBP. When the execution time increases from 55 to 183 min, the number of requests served by DBP is greater than that of FMBP. Beyond 183 min, many metadata requests are quickly served by FMBP, and the number of requests served by FMBP is again greater than that of DBP. The execution time of FGP can be divided into the following three time intervals: [0, 35], [36, 86], and [87, 115]. In the first interval, FGP mainly uses the frequency-mining-based policy to prefetch metadata, which is reflected in the fact that the curves of FGP and FMBP are nearly coincident. In the second interval, FGP mainly uses the directory-based policy to prefetch metadata, which is reflected in the fact that the curves of FGP and FMBP have similar slopes. In the third interval, FGP mainly uses the frequency-mining-based policy to prefetch metadata, which is reflected in the fact that many metadata requests are quickly served in a short time. When the replaying of OC2 is fulfilled, the order of hit ratio is: FPG > WGP > FMBP > NABP > NP > DBP. The order of execution time is: FGP < WGP < FMBP < NP < DBP < NABP.

Regarding scenario SC, initially, the number of requests served by DBP is greater than that of FMBP. Beyond 174 min, the number of requests served by the FMBP is greater than that of the DBP. When the execution time is less than 90 min, no significant differences are observed between FGP and DBP in terms of the number of requests served at a time. Beyond 90 min, the number of metadata requests served by FGP at a time is higher than that of DBP. When the replaying of SC is fulfilled, the FGP improves the hit ratios by as much as 12.88%, 8.61%, 7.84%, 3.31%, and 9.09% as compared to those of the NP, WGP, FMBP, DBP, and NABP, respectively. These correspond to 31.56%, 23.76%, 22.22%, 31.25%, and 55.87% reductions in execution time, respectively.

Fig. 8 shows the ratios of C-Cache's execution time to the total execution time of FGP for different trace-replay scenarios. The ratios for OC1, OC2, and SC are 11.42%, 9.11%, and 14.13%, respectively. This demonstrates that the overhead of C-Cache is relatively low. The execution time of C-Cache is relatively high when directory-based prefetching policy is applied to C-Cache. This is because network communications and I/O operations are involved in the execution of C-Cache. The order of hit ratio is: SC > OC1 > OC2. This can be inferred that the ratio of the directory-based prefetching policy being executed is relatively high in SC.

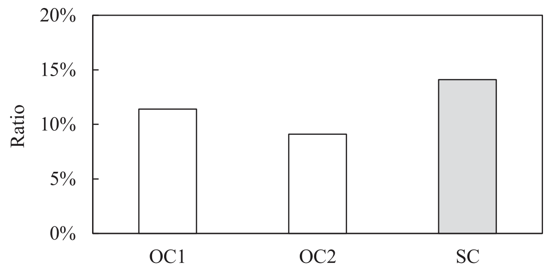


Figure 8: The ratios of C-Cache's execution time to the total execution time of FGP for different trace-replay scenarios

To further analyze the performance overhead introduced by *C-Cache*, we conducted detailed profiling of its resource usage, including memory consumption, CPU cycles, and latency per operation. Table 6 presents the quantitative evaluation results collected during the execution of OC1 scenario.

Table 6: Detailed resource overhead of C-Cache in OC1 scenario

Metric	Value
Average memory footprint	12.6 MB
Average CPU cycles per metadata request	8,120 cycles
Average metadata lookup latency (hash table)	0.34 μs
Average metadata insertion latency	0.58 μs
C-Cache hit rate	81.3%
C-Cache eviction rate (per 10k requests)	2.7%

As shown in Table 6, the memory overhead of C-Cache remains modest due to its fixed-size design (10,000 entries by default). Metadata lookups and insertions are both executed within sub-microsecond latencies, supported by a hash table with $O(1)$ access time. The average CPU cost per metadata request is 8120 cycles, which accounts for only a small fraction of total client-side processing time (typically over 200,000 cycles per request in full-stack metadata access including FUSE and I/O stack).

Furthermore, the eviction rate is relatively low (2.7%), indicating that the cache capacity is sufficient to hold most working-set metadata during each cutting window. The high hit rate of C-Cache (over 80%) also confirms its effectiveness in modeling and comparing metadata access patterns between prefetching policies.

Overall, these results demonstrate that while C-Cache introduces additional overhead, its design ensures lightweight, fast, and scalable performance with limited impact on overall system efficiency. The benefits it brings-particularly in enabling dynamic prefetching strategy switching—far outweigh its cost.

6.2.2 Effects of Configurations of FGP

A parameter that might affect the benefits of FGP is *cWinSize*. Fig. 9a shows the effects of varying *cWinSize* from 2000 to 20,000. A smaller *cWinSize* corresponds to a more granular prefetching policy. The figure shows that the hit ratio is maximum when *cWinSize* is 2000. However, the increase in *cWinSize* does not necessarily lead to a decrease in the hit ratio. The hit ratio is improved as *cWinSize* is increased from 16,000 to 20,000 in SC.

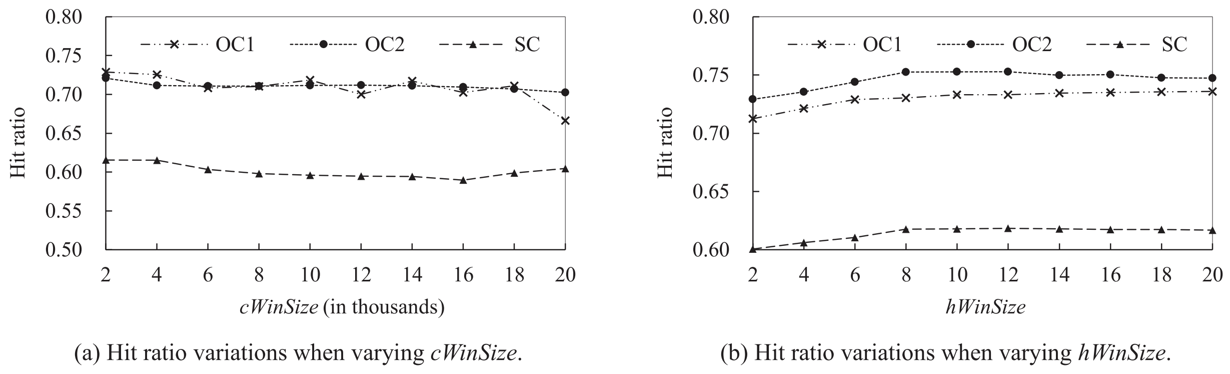


Figure 9: Hit ratio variations for FGP when varying *cWinSize* and *hWinSize*

Fig. 9b shows the effects of varying *hWinSize*. In OC2, the hit ratio initially increases as *hWinSize* is increased. It reaches a maximum when *hWinSize* is set to 10. Beyond 10, the hit ratio decreases as *hWinSize* is increased. This phenomenon can be expected. When *hWinSize* is very small, FGP considers only immediate successors as candidates for future prediction. This phenomenon is also true in SC. When *hWinSize* is very large, several unrelated items are considered as candidates for future prediction.

Parameters *brd* and *dph* significantly affect the benefits of FGP. Fig. 10a shows the effects of varying *brd* from 1 to 10. Initially, the hit ratio increases as *brd* increases beyond a certain number and then fluctuates. This is because the increase in *brd* corresponds to an increased number of related items.

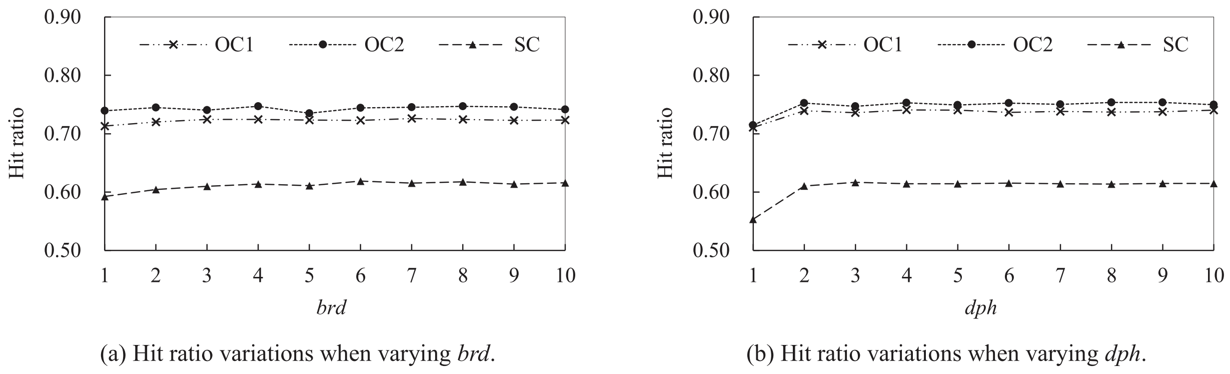


Figure 10: Hit ratio variations for FGP when varying *brd* and *dph*

Fig. 10b shows the effects of varying *dph* from 1 to 10. *Dph* determines the distance to look ahead to prefetch. When *dph* is varied from 1 to 2, the hit ratios for OC1, OC2, and SC increase significantly. This phenomenon shows that prefetching can significantly increase hit ratios. Beyond 2, the hit ratios do not increase with *dph*. This is because looking ahead too far may compromise the algorithm's effectiveness by introducing noise into the relationship graph and client-side cache.

Table 7 summarizes the hit ratios and execution times of all six evaluated metadata prefetching strategies across three representative trace-replay scenarios. As shown, FGP consistently achieves the highest hit ratio and the lowest execution time, demonstrating its adaptability and effectiveness under diverse workload conditions.

Table 7: Comparison of metadata prefetching strategies across trace scenarios

Method	Hit ratio (%)			Execution time (min)		
	OC1	**OC2**	**SC**	**OC1**	**OC2**	**SC**
NP	51.78	49.34	53.12	132.5	147.2	125.1
WGP	54.19	56.18	61.73	121.5	130.6	113.2
FMBP	54.64	55.41	62.56	118.7	128.0	112.3
DBP	55.93	48.12	59.25	135.1	150.9	124.7
NABP	53.10	50.37	54.18	143.7	161.4	140.2
FGP	**59.95**	**58.84**	**66.00**	**79.2**	**105.5**	**85.6**

Note: *Bold values indicate the best performance in each column.

6.2.3 Ablation Study

To assess the individual contributions of key components in the proposed FGP prefetching framework, we perform a set of ablation experiments. Specifically, we test the following four variants of FGP:

- **FGP-w/o-Switch:** Disables the dynamic switching between FMBP and DBP; FMBP is always used.
- **FGP-w/o-FMBP:** Disables the FMBP module entirely; only DBP is used.
- **FGP-brd1:** Sets prefetching breadth *brd* to 1, significantly reducing lookahead width.
- **FGP-dph1:** Sets prefetching depth *dph* to 1, disabling deeper metadata traversal.

Table 8 shows the results on the SC workload. The standard FGP achieves the best performance, and the removal or degradation of any component results in a noticeable performance drop. Notably, disabling dynamic switching (FGP-w/o-Switch) leads to a 4.87% decrease in hit ratio and a 12.3% increase in execution time. This confirms the importance of adaptive policy selection under mixed workloads.

Table 8: Ablation study of FGP under SC workload

Variant	Hit ratio (%)	Execution time (min)
FGP (Full version)	66.00	85.6
FGP-w/o-Switch	61.13	96.1
FGP-w/o-FMBP	59.25	101.4
FGP-brd1	60.52	94.2
FGP-dph1	61.67	91.8

The results show that:

- *Dynamic switching* contributes significantly to adapting across workload phases.
- *FMBP* alone provides high accuracy in identifying non-local metadata correlations.
- Both *brd* and *dph* are crucial in traversing the relationship graph to uncover deeper metadata access patterns.

This ablation study validates that each component in FGP plays a complementary role, and their combination yields the most effective prefetching performance.

6.3 Evaluation of Data Placement

The variations in the number of operations over time for a single client process are shown in Fig. 11, where each operation represents a trace record that is replayed. PWFSL achieves a lower execution time than DORP. One reason is that the written data are rarely read later, indicating that dynamically placing a new data replica close to the clients does not significantly benefit the reading. Another reason is that placing a new data replica consumes the precious network bandwidth can then be harmful to data reads and writes. PDRD has the highest execution time when the trace replay is complete. One reason is that PDRD must update two data replicas simultaneously for each write, which then incurs serious write amplification. Another reason is that the writing of data replicas has little effect on improving read performance. FGDP achieves the lowest execution time when the trace replay is complete. Compared with PWFSL, DORP, and PDRD, FGDP reduces the execution time by 17.19%, 25.35%, and 70.39%, respectively. These experimental results show that FGDP can significantly improve access performance. This is because the written data can be placed directly in the local data center rather than sent to the remote data center that holds the original file.

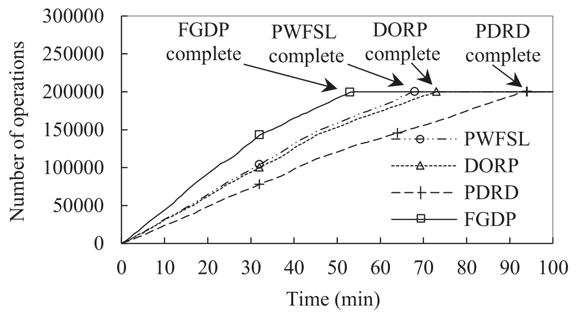


Figure 11: Variations in the number of operations for a single client process

Fig. 12 shows the aggregate throughput for different numbers of client processes when replaying the fserver-trace. FGDP achieves the highest aggregate throughput for different numbers of client processes, and the advantages of FGDP increase as the number of client processes increases. For instance, when the number of client processes is 1, FGDP improves the aggregate throughput for PWFSL, DORP, and PDRD by as much as 25.93%, 35.19%, and 74.07%, respectively. When the number of client processes is 16, FGDP improves the aggregate throughput for PWFSL, DORP, and PDRD by as much as 66.77%, 98.76%, and 145.24%, respectively. This behavior occurs because the contending for network resources becomes more intense as the number of client processes is increased. The data written by FGDP are placed in the local PDFS rather than sent to the remote PDFS, and precious network bandwidth can then be allocated for data reading. This behavior also shows that FGDP achieves a higher scalability than other approaches. For instance, the aggregate throughput of each approach improves significantly as the number of client processes increases from 1 to 64. However, when the number of client processes increases from 64 to 128, only the aggregate throughput of FGDP continues to improve.

Table 9 presents a comparative summary of execution time and aggregate throughput for different data placement strategies. FGDP outperforms the other approaches in both metrics, especially under heavy client concurrency, confirming its high scalability and efficiency.

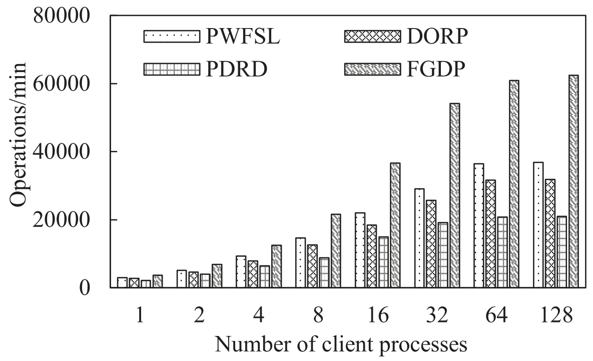


Figure 12: Aggregate throughput for different numbers of client processes when replaying fserver-trace

Table 9: Comparison of data placement strategies

Method	Execution time (s)	Throughput with 128 clients (MB/s)
PWFSL	582.3	192.6
DORP	622.4	174.8
PDRD	984.6	134.3
FGDP	482.0	330.0

6.4 Generalizability and Applicability of the Proposed Methods

While the proposed methods are evaluated using representative real-world traces such as OLTP, search, and cloud service workloads, we believe the observed improvements are not limited to these specific scenarios. The key design principles—relationship-graph-based prefetching (FMBP), dynamic policy switching (FGP), and locality-preserving fragment placement (FGDP)—are applicable to a wide range of file system workloads with different access patterns.

In particular, the metadata prefetching approach is beneficial in any application that exhibits temporal or spatial locality in metadata access. For example:

- **Scientific computing** workloads (e.g., HPC applications) often involve repeated sweeps over directory structures or checkpoint files, which could benefit from DBP in early iterations and FMBP in later adaptive phases.
- **Media processing or image/video streaming** applications typically access metadata in predictable sequences (e.g., frames or tiles), which FMBP can efficiently capture.
- **System monitoring or logging frameworks** often generate bursty but highly structured access streams (e.g., accessing daily log files by hour or service ID), making them suitable for our adaptive prefetching framework.

Moreover, the fragment placement mechanism in FGDP can be beneficial beyond write-intensive cloud traces. It can be adapted to any environment where reducing cross-data-center traffic is important, such as edge computing, distributed archival systems, or federated data lakes. Its locality-aware placement can reduce WAN bandwidth consumption and improve performance in scenarios with data affinity to specific geographic locations.

That said, some workloads with extremely random or one-off access patterns (e.g., backup scanning, antivirus scans) may not significantly benefit from metadata prefetching. Similarly, if data are written once and never accessed again, fragment placement optimizations bring little benefit. For such cases, fallback to minimal prefetching and direct-write policies ensures correctness without performance degradation.

In future work, we plan to validate the methods on synthetic traces with configurable locality patterns and deploy them on domain-specific workloads to further assess generalizability.

7 Conclusions

We presented efficient approaches for metadata prefetching and data placement. The access time of file system metadata and application data was successfully reduced by using fine-grained control of prefetching policies and writing of variable-size fragments. For metadata prefetching, the prefetching accuracy has been improved by slicing the access sequence into windows, selecting the better prefetching policy in the current window, and applying it to the next window. During data placement, the amount of data transmitted over the network has been significantly reduced by placing the written data into the local data center and sending the location information of the written data to the remote data center that holds the original file.

The proposed approaches were implemented on a WAVFS, which was also designed in this study. Their performance was evaluated on different real system traces. The experimental results are as follows:1) the proposed metadata prefetching approach reduced the access time of file system metadata by 40.27%, 35.1%, 33.5%, 40.53%, and 62.08% compared to those of NP, WGP, FMBP, DBP, and NABP, respectively; 2) the proposed data placement approach reduced the access time of application data by 17.19%, 25.35%, and 70.39%, respectively, compared to those of PWFSL, DORP, and PDRD, respectively.

Acknowledgement: The authors thank those people or institutions that have helped you in the preparation of the manuscript.

Funding Statement: This research was funded by the National Natural Science Foundation of China under Grant No. 62362019 and the Hainan Provincial Natural Science Foundation of China under Grant No. 624RC482.

Author Contributions: The authors confirm their contribution to the paper as follows: Bing Wei: Conceptualization, Methodology, Data curation, Formal analysis, Writing—original draft. Yubin Li: Conceptualization, Methodology, Formal analysis, Writing—review & editing. Yi Wu: Investigation, Formal analysis, Writing—review & editing. Ming Zhong: Supervision, Validation, Writing—review & editing. Ning Luo: Resources, Supervision, Writing—review & editing. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: https://github.com/Haida-source/WAVFS-client-and-server.git (accessed on 09 June 2025).

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Nadig D, Ramamurthy B, Bockelman B. SNAG: SDN-managed network architecture for GridFTP transfers using application-awareness. IEEE/ACM Trans Netw. 2022;30(4):1585–98. doi:10.1109/tnet.2022.3150000.
2. Handono FW, Nurdin H. Network file system implementation on computer-based exam. Indonesian J Eng Res. 2020;1(1):20–6. doi:10.11594/ijer.01.01.04.
3. Nikolaus R. SSHFS: super easy file access over SSH 2024 [Internet]. [cited 2025 May 28]. Available from: https://github.com/libfuse/sshfs.
4. Rybintsev VO. Optimizing the parameters of the Lustre-file-system-based HPC system for reverse time migration. J Supercomput. 2020;76(1):536–48. doi:10.1007/s11227-019-03054-7.

5. Shen J, Zuo P, Luo X, Yang T, Su Y, Zhou Y, et al. FUSEE: a fully memory-disaggregated key-value store. In: 21st USENIX Conference on File and Storage Technologies (FAST 23); 2023 Feb 21–23; Santa Clara, CA, USA. p. 81–98.
6. Liao G, Abadi DJ. FileScale: fast and elastic metadata management for distributed file systems. In: Proceedings of the 2023 ACM Symposium on Cloud Computing; 2023 Oct 30–Nov 1; Santa Cruz, CA, USA. p. 459–74.
7. Dong C, Wang F, Yang Y, Lei M, Zhang J, Feng D. Low-latency and scalable full-path indexing metadata service for distributed file systems. In: 2023 IEEE 41st International Conference on Computer Design (ICCD); 2023 Nov 6–8; Washington, DC, USA. p. 283–90.
8. Xu W, Zhao X, Lao B, Nong G. Enhancing HDFS with a full-text search system for massive small files. J Supercomput. 2021;77(7):7149–70. doi:10.1007/s11227-020-03526-1.
9. Li Z, Chen Z, Srinivasan SM, Zhou Y. C-miner: mining block correlations in storage systems. In: FAST'04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies; 2004 Mar 31–Apr 2; San Francisco CA, USA. p. 173–86.
10. Chen Y, Li C, Lv M, Shao X, Li Y, Xu Y. Explicit data correlations-directed metadata prefetching method in distributed file systems. IEEE Trans Parallel Distrib Syst. 2019;30(12):2692–705. doi:10.1109/tpds.2019.2921760.
11. Wei B, Xiao L, Zhou B, Qin G, Yan B, Huo Z. I/O optimizations based on workload characteristics for parallel file systems. In: Network and Parallel Computing: 16th IFIP WG 10.3 International Conference, NPC 2019. Cham, Switzerland: Springer; 2019. p. 305–10.
12. Tatebe O, Hiraga K, Soda N. Gfarm grid file system. New Gener Comput. 2010;28(3):257–75. doi:10.1007/s00354-009-0089-5.
13. Thomson A, Abadi DJ. CCalvinFS: consistent WAN replication and scalable metadata management for distributed file systems. In: 13th USENIX Conference on File and Storage Technologies (FAST 15); 2016 Feb 22–25; Santa Clara, CA, USA. p. 1–14.
14. Aral A, Ovatman T. A decentralized replica placement algorithm for edge computing. IEEE Trans Netw Serv Manag. 2018;15(2):516–29. doi:10.1109/tnsm.2017.2788945.
15. Kim D, Min K, Oh J, Won Y. ScaleXFS: getting scalability of XFS back on the ring. In: 20th USENIX Conference on File and Storage Technologies (FAST 22); 2022 Feb 22–24; Santa Clara, CA, USA. p. 329–44.
16. Oh J, Ji S, Kim Y, Won Y. ExF2FS: transaction support in log-structured filesystem. In: 20th USENIX Conference on File and Storage Technologies (FAST 22); 2022 Feb 22–24; Santa Clara, CA, USA. p. 345–62.
17. Zhang S, Roy R, Rumancik L, Wang AI. The composite-file file system: decoupling one-to-one mapping of files and metadata for better performance. ACM Trans Storage. 2020;16(1):1–18. doi:10.1145/3366684.
18. Habermann P, Chi CC, Alvarez-Mesa M, Juurlink B. Application-specific cache and prefetching for HEVC CABAC decoding. IEEE Multimed. 2017;24(1):72–85. doi:10.1109/mmul.2017.12.
19. Al Assaf MM, Jiang X, Qin X, Abid MR, Qiu M, Zhang J. Informed prefetching for distributed multi-level storage systems. J Signal Process Syst. 2018;90(4):619–40. doi:10.1007/s11265-017-1277-z.
20. Gu P, Wang J, Zhu Y, Jiang H, Shang P. A novel weighted-graph-based grouping algorithm for metadata prefetching. IEEE Trans Comput. 2009;59(1):1–15. doi:10.1109/tc.2009.115.
21. Pan S, Stavrinos T, Zhang Y, Sikaria A, Zakharov P, Sharma A, et al. Facebook's tectonic filesystem: efficiency from exascale. In: 19th USENIX Conference on File and Storage Technologies (FAST 21); 2021 Feb 23–25; Online. p. 217–31.
22. Yang J, Izraelevitz J, Swanson S. Orion: a distributed file system for Non-Volatile main memory and RDMA-Capable networks. In: 17th USENIX Conference on File and Storage Technologies (FAST 19); 2019 Feb 25–28; Boston, MA, USA. p. 221–34.
23. Daniel E, Tschorsch F. IPFS and friends: a qualitative comparison of next generation peer-to-peer data networks. IEEE Commun Surv Tutorials. 2022;24(1):31–52. doi:10.1109/comst.2022.3143147.
24. Oh K, Qin N, Chandra A, Weissman J. Wiera: policy-driven multi-tiered geo-distributed cloud storage system. IEEE Trans Parallel Distrib Syst. 2019;31(2):294–305. doi:10.1109/tpds.2019.2935727.
25. Mendes R, Oliveira T, Cogo V, Neves N, Bessani A. Charon: a secure cloud-of-clouds system for storing and sharing big data. IEEE Trans Cloud Comput. 2019;9(4):1349–61. doi:10.1109/tcc.2019.2916856.

26. Wang H, Shen H, Li Z, Tian S. GeoCol: a geo-distributed cloud storage system with low cost and latency using reinforcement learning. In: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS); 2021 Jul 7–10; Washington, DC, USA. p. 149–59.
27. Henschel M, Böttger T, Rümmer D. GridFS: grid file system. In: High Performance Computing: Proceedings of the 18th European Conference. Cham, Switzerland: Springer; 2012; p. 226–34.
28. Qian Y, Li X, Ihara S, Dilger A, Thomaz C, Wang S, et al. LPCC: hierarchical persistent client caching for lustre. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2023 Nov 12–17; Denver, CO, USA. p. 1–14.
29. Wei B, Xiao L, Zhou H, Qin G, Song Y, Zhang C. Global virtual data space for unified data access across supercomputing centers. IEEE Trans Cloud Comput. 2022;11(2):1822–39. doi:10.1109/tcc.2022.3164251.
30. Li Q, Chen L, Wang X, Huang S, Xiang Q, Dong Y, et al. Fisc: a large-scale cloud-native-oriented file system. In: 21st USENIX Conference on File and Storage Technologies (FAST 23); 2023 Feb 21–23; Santa Clara, CA, USA; p. 231–46.
31. Ken B. OLTP application and search engine I/O 2024 [Internet]. [cited 2025 May 28]. Available from: https://traces.cs.umass.edu/index.php/.
32. Oe K, Ogihara K, Honda T. Analysis of commercial cloud workload and study on how to apply cache methods. IEICE Techn Rep. 2018;118:7–12.
33. Shen J, Zhang K, Gu J, Zhou Y, Wang X. Efficient scheduling for multi-block updates in erasure coding based storage systems. IEEE Trans Comput. 2017;67(4):573–81. doi:10.1109/tc.2017.2769051.
34. Chen H, Zhang H, Dong M, Wang Z, Xia Y, Guan H, et al. Efficient and available in-memory KV-store with hybrid erasure coding and replication. ACM Trans Storage. 2017;13(3):1–30. doi:10.1145/3129900.
35. Cao Z, Dong S, Vemuri S, Du DHC. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at Facebook. In: 18th USENIX Conference on File and Storage Technologies (FAST 20); 2020 Feb 24–27; Santa Clara, CA, USA. p. 209–23.
36. STyler H. Production traces collected at Microsoft using the event tracing for windows framework 2024 [Internet]. [cited 2025 May 28]. Available from: http://iotta.snia.org/traces/130/.