MDPI

Article

HRFP: Highly Relevant Frequent Patterns-Based Prefetching and Caching Algorithms for Distributed File Systems

Anusha Nalajala [1], T. Ragunathan [2], Ranesh Naha [3,*] and Sudheer Kumar Battula [4]

[1] CSE Department, SRM University-AP, Amaravati 522502, India
[2] Faculty of Engineering and Technology, Sri Ramachandra Institute of Higher Education and Research, Chennai 600116, India
[3] School of Computer Science, The University of Adelaide, Adelaide 5005, Australia
[4] School of Technology, Environments and Design (TED), University of Tasmania, Hobart 7000, Australia
* Correspondence: ranesh.naha@adelaide.edu.au

Abstract: Data-intensive applications are generating massive amounts of data which is stored on cloud computing platforms where distributed file systems are utilized for storage at the back end. Most users of those applications deployed on cloud computing systems read data more often than they write. Hence, enhancing the performance of read operations is an important research issue. Prefetching and caching are used as important techniques in the context of distributed file systems to improve the performance of read operations. In this research, we introduced a novel highly relevant frequent patterns (HRFP)-based algorithm that prefetches content from the distributed file system environment and stores it in the client-side caches that are present in the same environment. We have also introduced a new replacement policy and an efficient migration technique for moving the patterns from the main memory caches to the caches present in the solid-state devices based on a new metric namely the relevancy of the patterns. According to the simulation results, the proposed approach outperformed other algorithms that have been suggested in the literature by a minimum of 15% and a maximum of 53%.

Keywords: frequent patterns; cloud computing systems; prefetching; caching and replacement methods; distributed file systems

Citation: Nalajala, A.; Ragunathan, T.; Naha, R.; Battula, S.K. HRFP: Highly Relevant Frequent Patterns-Based Prefetching and Caching Algorithms for Distributed File Systems. *Electronics* **2023**, *12*, 1183. https://doi.org/10.3390/electronics12051183

Academic Editors: Minseok Choi and Joongheon Kim

Received: 2 February 2023
Revised: 23 February 2023
Accepted: 27 February 2023
Published: 1 March 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Distributed file system (DFS) is a client-server architecture which enables web applications to store, retrieve, and process data from cloud computing platforms. A DFS creates the impression to the user that the data is kept on the same node (computer system) where the user is working even though the data is actually stored on some other nodes. DFS is a crucial component on cloud computing platforms which is utilized as storage at the back end. Cloud computing platforms are commonly used to deploy data-intensive applications due to their enormous storage and processing capacities. The users of such data-intensive applications that are deployed in the cloud perform read access requests (read requests) more frequently when compared to write requests [1,2]. Therefore improving the performance of read operations carried out on the DFS is a significant and challenging research problem. The main aim of this research is to improve the performance of read operations by proposing highly relevant frequent patterns-based prefetching, caching, and replacement algorithms based on pattern relevancy.

The prefetching and client-side caching techniques [3,4] are used to fetch the data in advance and cache them in the main memory (primary memory) or solid-state drive (SSD) caches. Therefore, the file blocks can be read from these caches rather than DFS by the web application programs that are executing on that particular nodes. The time required to read the file blocks from the caches (primary and SSD caches) maintained in the nodes is less than the time required to read the same from DFS. Hence, the performance of

read operations can be improved by reading the data from main memory and SSD caches thereby reducing the disk accesses.

Nowadays we can find computer systems with hard disks and SSD for storage. Multiple types of secondary storage devices are used in general, to improve file access performance as SSDs are faster than hard disks [5]. The file system is maintained by the operating system in the hard disk and SSD of modern computers. Therefore, the DFS utilized by cloud computing platforms must be able to effectively utilize both of these storage media. This study addresses the issue of effectively utilising multiple memory devices to reduce read access time.

We have assumed a rack-organised DFS environment with a name node and global cache node in one of the racks, and a set of data nodes in all racks with multiple storage devices. We also introduced a new HRFP-based prefetching algorithm which prefetches the highly relevant frequent patterns based on the relevancy value generated in each data node. The patterns which are prefetched using the HRFP-based prefetching algorithm are stored in the caches of each data node efficiently. Thus, an increase in the speed of serving the read requests in the DFS is observed. For that, we have compared our HRFP algorithm with existing algorithms from the literature in terms of the cache hit ratio and average read access time. We observed that the proposed approach outperformed the algorithms proposed in the literature.

The following are the contributions of this paper.

1. Proposed highly relevant frequent patterns-based prefetching algorithm to prefetch the file blocks based on pattern relevancy.
2. Proposed multi-level caching algorithm to fill the data node caches and caches of global cache node with prefetched file blocks in an efficient manner.
3. Introduced a novel relevancy-based replacement policy to replace the caches of data nodes and global cache node whenever there is a need to create space for storing the incoming patterns.

The structure of remaining paper is organized as follows. The related work is presented in Section 2 and the architecture of the proposed method is discussed in Section 3. In Section 4, the procedure to identify highly relevant frequent patterns and our proposed HRFP algorithm are discussed. Section 5 discusses the results and the conclusions are presented in Section 6.

2. Related Work

The prefetching, caching, and cache replacement techniques addressed in the literature are presented in this section.

2.1. Literature on Prefetching and Caching Techniques

In this subsection, the methods for prefetching and, the caching techniques proposed in the literature are discussed.

The metadata activities are segregated from the data operations in the distributed file system (DFS) environment. The authors of [6] introduced HR-Meta, a technique for prefetching metadata depending on how file access sequences relate to one another. This method prefetches the information of the files that were related to the requested file whenever a certain file was requested by the client. To minimise the number of metadata accesses, related file metadata that had already been prefetched was given to the client. Prefetching and caching file data in client-side caches was not the main concern of the authors in this case, which further hinders the performance of the DFS.

In [7], the authors developed a novel mechanism to merge file names to feature vectors and trained a gated recurrent neural network to provide file prefetching strategies in order to analyse application I/O access patterns and improve the performance of current file systems. They mapped the names of files or directories into a high-dimensional vector space using the proposed embedding technique to show the relationship between files. They used a neural network to determine whether or not a file needed to be prefetched

by providing an access sequence of files as input. The disadvantage here is prediction, which has computational overhead. Furthermore, a lot of data is loaded into the cache, but only a little of it is really used. Storage problems are caused by these excessive data-loading operations.

In [8–10], the authors introduced the initiative data prefetching technique. The data was prefetched by using prediction algorithms based on past disk accesses. Before the prefetched data was requested, it was delivered to the appropriate client. There is often a storage and computation burden, and the predictions may not be accurate in many situations.

In [11], data correlation-based prefetching was implemented. The prefetching method employed in this method was based on text-based syntax analysis. The reference relations used in this approach were used to prefetch the data from the files. This method prefetches the files that were explicitly correlated (in the form of hyperlinks) in the target file when it was requested. Since the correlated files were also prefetched together with the requested file, which has a higher frequency, the files with a lower frequency and correlation to the requested file may not be disregarded in this scenario, and maintaining them in the cache may create a storage problem.

File data was prefetched using the intelligent pipe-lined prefetching strategy for distributed systems known as IPODS based on the hints produced by web applications [12]. There will be more burden on client applications for generating hints to prefetch the data. Hermes is a distributed and hierarchical I/O system [13], in which prefetching of the file data was done using a server push strategy, which results in an increase of burden on servers which may result in low performance. The authors of [14,15] introduced a prefetching technique based on simple support. The data that was prefetched using support value is cached without using multiple storage devices.

The authors of [16,17] presented prefetching methods that were applied at the file system level (disk). The file access patterns that need to be prefetched were found using frequent sequence pattern mining techniques. There may or may not always be a correlation between the access patterns. The proposed methods might not work well if file requests are interleaved or if access patterns vary often.

To increase I/O access performance, recent research [18] has recommended using flash-based storage devices, phase-changing memory devices and dynamic RAMs for caching [19–22]. In [23], a file system was proposed, that consists of solid-state drives (SSDs) and non-volatile RAMs for specific applications that require long execution times. All of these caching techniques addressed in the literature mainly concentrate on data caching in local caches only.

To address flash memory access latency and cost constraints, the authors of [24] presented a hybrid main memory structure that combines dynamic RAM and flash storage with a cluster-based migration mechanism positioned between them. To address performance loss brought on by unforeseen memory access patterns, a regression-based prefetching approach was developed. The drawback with this method is, the prefetching is based on prediction which requires more computation.

Some researchers used the centralised cache approach to track access frequency and notify data nodes to store popular data [25,26]. However, jobs that run on the same node are the only ones that can use the cached data. Big data applications, cannot bring significant performance improvement with those approaches.

Remote memory access was used by the developers of [27] to implement a novel caching system in HDFS. A separate cache node was used to store the information of cached data in each data node. All of the data nodes in the cluster have access to this cache node. The data was cached based on frequency without considering the recentness is the disadvantage of this approach.

In [28], the authors introduced cache performance optimization of the quality of experience framework for cloud computing servers. This paper presents a new cache replacement algorithm for variable video file sizes, analyses the particular requirements for the multi-terminal type of QoC framework, and provides an outline design for the client and server

sides. It then describes the implementation details for the client and server sides and concludes with a thorough functional and performance testing of the entire system. The disadvantage with this approach is increased complexity and memory usage. Optimizing cache performance can involve adding additional layers of complexity to the system or application. Furthermore, it often requires using more memory, which can be a disadvantage in memory-constrained systems.

Several state-of-the-art multi-level caching techniques were proposed in the literature [29–32] which mainly concentrates on caching the data based on prediction by analyzing the characteristics of I/O accesses. The authors of [33] predicted the life time of files by analyzing the access frequency of the files. The authors in [34] proposed WorkflowRL method which manages the data in the multi-level storage systems based on reinforcement learning. These approaches mainly rely on prediction which involves computational overhead. Moreover, the prediction may not be precise for all types of workloads.

Several access frequency based caching methods for big data applications have been proposed in [35–38]. Hyperbolic caching [39] is a priority-based caching technique in which the priority is determined by the frequency of access after entering into the cache. The new file blocks without access frequency cannot withstand with these approaches and the old file blocks with access frequency that are already in cache persists for longer times.

The existing prefetching and caching approaches focus only on prefetching file blocks based on support, hints generation, and machine learning approaches and caching the same without considering multilevel memory devices. In this research, we introduced a new HRFP-based prefetching and caching algorithm based on the relevancy value. This algorithm reduces the number of disk accesses by allowing the majority of read requests made by client application programs to be fulfilled from the client-side caches kept in those specific data nodes. This decrease in disk access will shorten the read access time of the DFS.

2.2. Replacement Policies

Several replacement policies have been introduced in the literature ([40–43]) which are based on machine learning approaches. In [44,45], for web caching, a replacement method combining classification models were presented. These studies used the percentage of objects reuse to decide victim which is to be replaced from the cache. Even though these machine learning replacement methods bring benefits, they require more computational capacity which involves the updation of data for every data access in the memory.

In [4], the authors discussed several replacement procedures based on frequency, size, and weight considerations. Least recently used (LRU) is a recency-based policy that replaces the object that has been accessed the fewest times with a new object that has just arrived while taking into account the frequency of the replaced object's access. Least frequently used (LFU) policies, which are frequency-based, replace cache items whose frequency was lower and don't focus on how recently an object has been utilized. A replacement policy called SIZE replaces an object by taking its size into account; as a result, larger files are replaced first rather than the files which are accessed recently and frequently.

All the replacement policies addressed in the literature mainly focus on replacing the objects based on some factors like frequency, recentness and size. Some replacement policies were also proposed based on machine learning methods which involve complex computations. In this research, we proposed a novel relevancy-based replacement policy for file block access patterns and a migration technique for moving the evicted patterns to the global cache node or SSD cache of the same node based on relevancy value. So, we can save the evicted pattern if its relevancy value is higher than the patterns stored in the global cache node or SSD cache of the same node.

The following Table 1 summarizes the related work.

Table 1. Related work.

Reference	Approach	Limitations
J. Zhang et al. [6]	HR-Meta	Client-side prefetching was not considered.
H. Chen et al. [7]	RNN based prefetching	Computational Overhead is high
G.O. Ganfure et al. [46]	Deeprefetcher	Overhead of tranining neural network models.
G. Cherubini et al. [47]	Prefetching was done based on machine learning models	Computational and storage overhead are high.
Y. Chen et al. [11]	Data correlation based prefetching	Files which are not popular were also prefetched.
M.M. Al Assaf et al. [12]	IPODS	Overhead on clients to generate hints.
A. Kougkas et al. [13]	Hermes	Overhead on servers.
J. Liao et al. [8–10]	Initiative data prefetching	Low-level file system serves the I/O requests.
R. Gopisetty et al. [14,15]	Support-based prefetching	Multi-level memories were not considered.
S. Jiang et al. [16]	DiskSeen	Prefetching errors are high if file requests are interleaved.
Z. Li et al. [17]	C-miner	Prefetching may not be accurate if the file sequence access pattern changes regularly.
H. Herodotou et al. [25] T. Yoshimura et al. [26]	Centralized caching approach	Limited to single computer system.
H. Li et al. [48] approach	Shared Cache The data was cached irrespective of popularity.	
S. K. Yoon et al. [20,21] S. Huang et al. [19] N. Niu et al. [22]	Flash- based caching approach	Not effective for big data applications
D. Akbari bengar et al. [40] M. Sabeghi et al. [41] P. Aimtongkham et al. [42] Y. Wang et al. [43]	Machine learning based replacement policies	Computational overhead is high
W. Ali et al. [4]	LRU, LFU, SIZE	These policies won't consider frequency and recency in a combined manner.

3. Architecture of Distributed File System

We have considered a rack-organized architecture for deploying the DFS as shown in Figure 1. Every rack has a switch attached to it, and all of those switches are linked to one main switch that is connected to the router for internet usage. The metadata information of file blocks stored on the data nodes (Dnodes) is maintained by the name node (Nnode). The Nnode consists of a metadata controller for managing activities related to metadata. In addition, Nnode has DFS server software installed on it. All applications running in the Dnodes can access a global cache directory maintained by the global cache node (Gnode). To store the file blocks, the Gnode consists of a local cache (Gnode_LC) and an SSD cache (Gnode_SC) and it has local and SSD cache controllers for handling the I/O requests. For performing prefetch activities, the Gnode additionally has a prefetch controller. The Dnode has local and SSD caches (Dnode_LC and Dnode_SC), and a hard

disk (disk) to store files and file blocks. The Dnodes have separate local and SSD cache controllers and also have a prefetch controller for handling prefetch-related activities. The Dnodes also have DFS server software installed on it. The Dnodes have a log to store the file and file block ids that are accessed by the client programs. The Nnode, Gnode, and set of Dnodes are arranged in rack organization and the Nnode and Gnode are present in one of the racks.

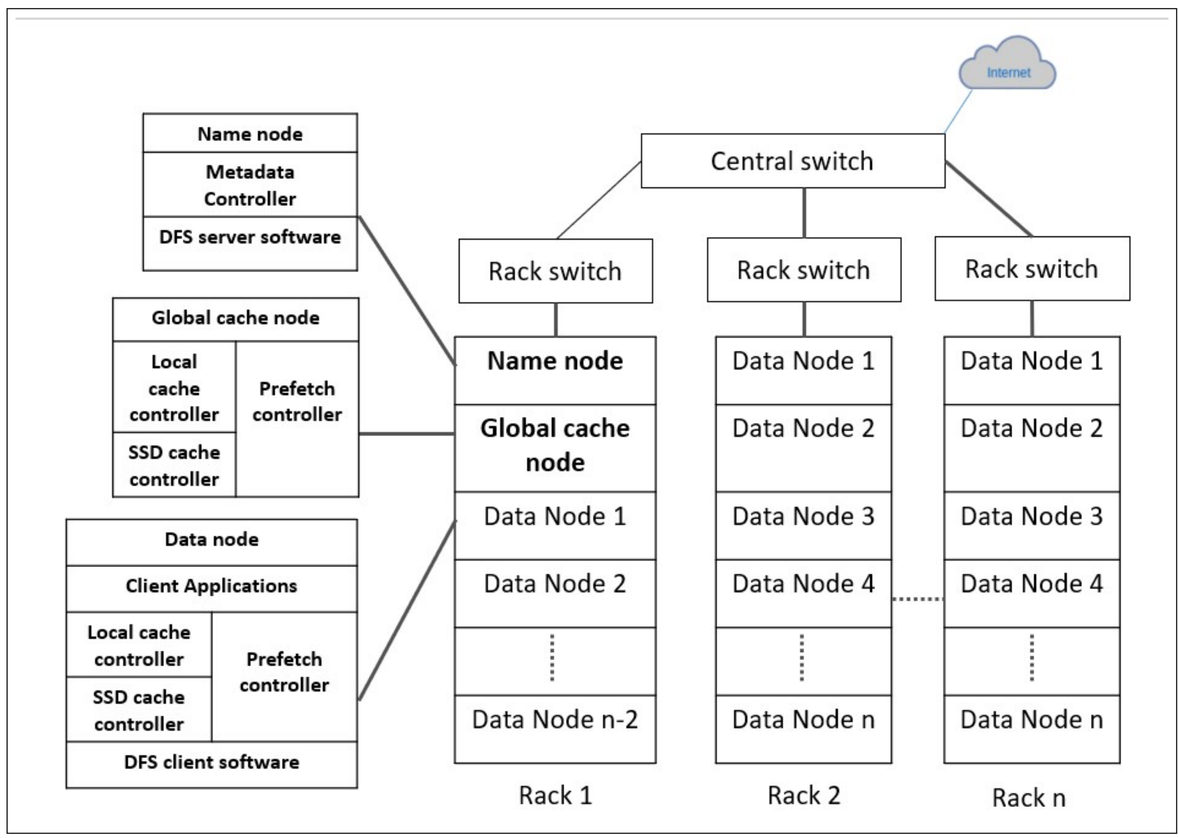


Figure 1. Rack organization of Nnode, Gnode, and Dnodes.

4. Proposed Work

This section covers the procedure for identifying the highly relevant frequent patterns (HRFPs) first. Next, the proposed HRFP-based algorithm is discussed. Then, the read and write algorithms followed in the DFS are explained. Lastly, the replacement procedure for HRFPs, is discussed.

4.1. Identification of HRFPs

This section discusses the procedure for identifying the HRFPs based on the relevancy value of the file block access pattern. As mentioned in the previous Section 3, all Dnodes maintain logs with the details of file block requests that are initiated by the client application programs.

The support values for all the files that are present in the log entries of the Dnodes are computed first. The number of entries of a file fid and the total number of entries in the log is divided by each other to determine the support value for that file. The files with 60% of support value [15] are considered popular files and are stored in *popular_files_list*. Next, the file block access patterns are identified for each popular file specified in *popular_files_list* by computing the confidence value. Let us assume that the, session of file blocks in the log is represented as [fid B_x B_y... B_z]. The confidence value for a pattern [B_x B_y] of a file fid is calculated as a fraction of the total number of times the pattern fid[Bf_x B_y] appears in the log of a Dnode and the total number of times fid[B_x] appears in the log [49]. The patterns along with the confidence value are stored in a *localconf_patterns_list*. Then, the relevancy

value for each pattern of a file is calculated by multiplying that file support value with the confidence value of that pattern.

The reason for multiplying the file support value with the confidence value of the pattern is to ensure that the highly popular patterns are prefetched. There may be files with low support values and patterns with high confidence values indicating that the popularity of the file is low and its patterns confidence is high. In this regard, we have multiplied the file support value by the confidence value of the pattern so that we may not lose the patterns with a high confidence value. After calculating the relevancy value for all the patterns, they are stored in *local_patterns_list* in the decreasing order of the relevancy value. The procedure for identifying the local HRFPs is described in the Algorithm 1.

Algorithm 1 Identification of local HRFPs

1. **for** each file fid in the log of Dnode **do**
2. calculate support value for fid
3. if(support(fid) $\geq$ sfid_th) then
4. add fid with support value to *popular_files_list*
5. sort *popular_files_list* in descending order of support value
6. **end for**
7. **for** each pattern of fid in *popular_files_list*
8. calculate confidence value for the pattern
9. add patterns with confidence value to *localconf_patterns_list*
10. calculate relevancy value for the patterns
11. add patterns with relevancy value to *local_patterns_list*
12. sort *local_patterns_list* in descending order of relevancy value.
13. Add patterns to *local_HRFPs_list*
14. **end for**

Next, the globally popular files are identified by calculating the global support value. The global support value of a file is computed using the fraction of the number of times a file ID appears in the log entries of Dnodes and the total number of log entries in Dnodes. The files with 60% support value are called global popular files and these files are saved in *global_popular_files_list*. Then for all global popular files, file block access patterns are identified by computing the global confidence value. The global confidence value for a pattern $[B_x\ B_y]$ of a file fid is computed as a fraction of the total number of times fid$[B_x\ B_y]$ appears in the entries of the logs in all Dnodes and the total number of times fid$[B_x]$ appears in the entries of the logs in all Dnodes. The patterns along with the confidence value are stored in *globalconf_patterns_list*. The relevancy value for all the patterns are calculated globally and are stored in *global_patterns_list* in the decreasing order of the relevancy value. The steps for identifying the global HRFPs are described in the Algorithm 2.

Algorithm 2 Identification of global HRFPs

1. **for** each file fid in the log **do**
2. calculate global support value for fid
3. if(global support(fid) $\geq$ sfid_th) then
4. add fid with global support value to *global_popular_files_list*
5. sort *global_popular_files_list* in descending order of global support value
6. **end for**
7. **for** each pattern of fid in *global_popular_files_list*
8. calculate global confidence value for the pattern
9. add patterns with global confidence value to *globalconf_patterns_list*
10. calculate the global relevancy value for all the patterns
11. add patterns with relevancy value to *global_patterns_list*
12. sort *global_patterns_list* in descending order of global confidence value.
13. Add patterns to *global_HRFPs_list*
14. **end for**

4.2. The HRFP-Based Prefetching and Caching Algorithm

We discuss the procedure for prefetching HRFPs from the DFS and caching them in local Dnodes followed by the procedure for prefetching and caching the global HRFPs in Gnode based on the relevancy value in this subsection.

4.2.1. Prefetching and Caching in Dnodes

Initially, the local HRFPs are identified by following the procedure described in Algorithm 1. After identifying the local HRFPs, they are stored in the *local_HRFPs_list*. Then, the HRFPs that are specified in the *local_HRFPs_list* are prefetched from the DFS and are cached in the multi-level storage devices (Dnode_LC and Dnode_SC) of all the Dnodes that are available in the DFS based on their respective sizes. The steps for prefetching the HRFPs and caching the same based on relevancy value in the Dnodes is presented in Algorithm 3.

Algorithm 3 Prefetching and Caching in Dnodes

1. **for** each Dnode
2. **while** Dnode_LC && Dnode_SC are *not full*
3. **for** each HRFP of fid in *local_HRFPs_list* **do**
4. **for** i=1 to Dnode_LC_MAX
5. prefetch $HRFP_i$ from DFS to Dnode_LC
6. **end for**
7. **if** Dnode_LC is *full*
8. **for** i= Dnode_LC_MAX + 1 to Dnode_SC_MAX
9. prefetch $HRFP_i$ from DFS to Dnode_SC
10. **end for**
11. **end if**
12. **end for**
13. **end while**

4.2.2. Prefetching and Caching in Gnode

The global HRFPs are identified by following the procedure described in Algorithm 2 and are stored in *global_HRFPs_list*. From *global_HRFPs_list*, the global HRFPs are prefetched from the DFS and cached the same in the client-side caches (Gnode_LC and Gnode_SC) maintained in Gnode. The process followed for prefetching the global HRFPs and caching the same in the Gnode is presented in the following Algorithm 4.

Algorithm 4 Prefetching and Caching in Gnode

1. **while** Gnode_LC && Gnode_SC are *not full*
2. **for** each HRFP of fid in *global_HRFPs_list* **do**
3. **for** i=1 to Gnode_LC_MAX
4. prefetch $HRFP_i$ from DFS to Gnode_LC
5. **end for**
6. **if** Gnode_LC is *full*
7. **for** i= Gnode_LC_MAX + 1 to Gnode_SC_MAX
8. prefetch $HRFP_i$ from DFS to Gnode_SC
9. **end for**
10. **end if**
11. **end for**
12. **end while**

4.3. Procedures for Read and Write in the DFS

We discuss the read, write, and replacement procedures followed by the DFS in this section.

4.3.1. Procedure for Reading from DFS

The default procedure for reading a file block (fb) which is initiated by a client process (CP) executing on a Dnode from the DFS is as follows:

1. The DFS client software (DFSCP) installed in Dnode makes communication with Nnode on behalf of CP to get the addresses of Dnodes where the fb was present.
2. The Nnode sends the Dnode addresses where the fb was stored.
3. After receiving the addresses from Nnode, DFSCP communicates with the nearby Dnode for reading fb and delivering it to the CP.

The proposed read procedure for reading fb which is requested by the CP executing in a Dnode is depicted in the Figure 2.

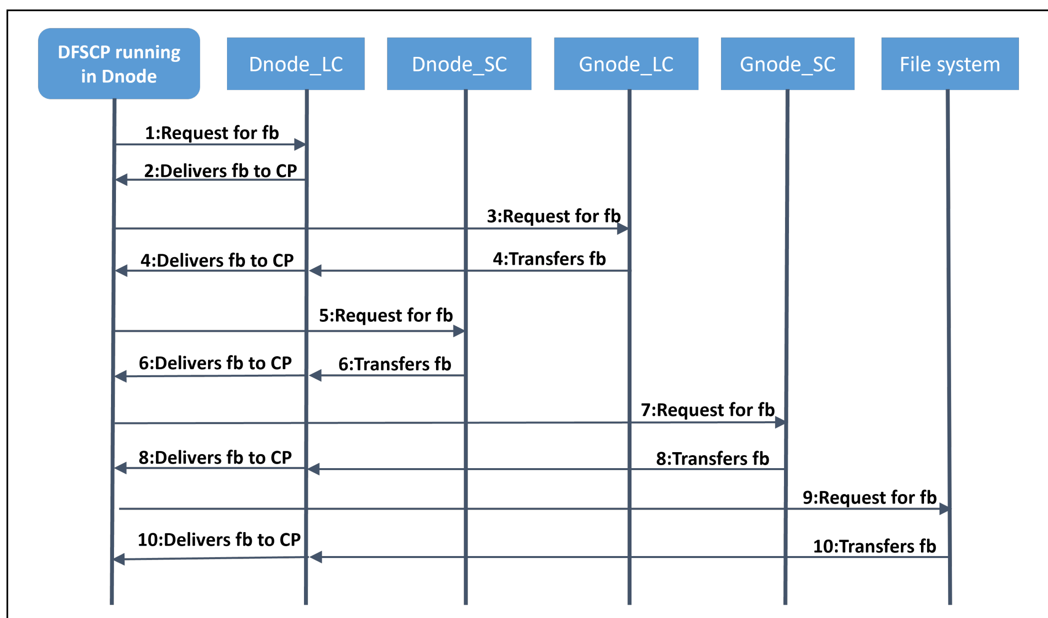


Figure 2. Flow diagram of proposed read procedure.

The steps involved for reading the fb are explained below:

1. Checks for fb in Dnode_LC.
2. fb is delivered to CP.
3. Checks for fb in Gnode_LC.
4. fb is transferred to Dnode_LC and delivered to CP.
5. Checks for fb in Dnode_SC.
6. fb is transferred to Dnode_LC and delivered to CP.
7. Checks for fb in Gnode_SC.
8. fb is transferred to Dnode_LC and delivered to CP.
9. Contacts Nnode and checks in the file system for fb.
10. fb is transferred to Dnode_LC and delivered to CP.

The Algorithm 5 explains the proposed read procedure to read a fb which is requested by the CP executing in a particular Dnode.

Algorithm 5 Proposed read algorithm

1. **if** fb is present in Dnode_LC of Dnode
2. CP reads fb from Dnode_LC
3. **else if** fb is present in Gnode_LC of Gnode
4. fb is read from Gnode_LC of Gnode and provided to CP
5. copy fb to Dnode_LC of Dnode
6. **if** Dnode_LC of Dnode is *full*
7. Follow relevancy-based replacement policy
8. **end if**
9. **else if** fb is present in Dnode_SC of Dnode
10. fb is read from Donde_SC of Dnode and provided to CP
11. **else if** fb is present in Gnode_SC of Gnode
12. fb is read from Gnode_SC of Gnode and provided to CP
13. copy fb to Dnode_LC of Dnode
14. **if** Dnode_LC of Dnode is *full*
15. Follow relevancy-based replacement policy
16. **end if**
17. **else**
18. Follow default read procedure
19. **end if**

4.3.2. Relevancy-Based Replacement Policy

Replacement policies are useful for creating space for the entry of new blocks whenever a miss in the cache appears. We have introduced a relevancy-based replacement policy for patterns that increases the hit ratio and at the same time decrease the read access time of the DFS. Whenever Dnode_LC of Dnode where the request initiated is full, the HRFP with the lowest relevancy value is evicted creating space for the HRFP where the requested fb is a member. The evicted pattern is either placed in the Gnode_LC of the Gnode or Dnode_SC of Dnode based on the relevancy value of the evicted HRFP to avoid losing patterns which have relevancy value higher than the existing patterns already present in the caches (Gnode_LC and Dnode_SC).

The relevancy-based replacement policy for HRFPs whenever the Dnode_LC and Dnode_SC of Dnodes and Gnode_LC and Gnode_SC of Gnode are full, is described in Algorithm 6.

Algorithm 6 Relevancy-based replacement policy

1. **if** Dnode_LC of Dnode is *full*
2. Remove the HRFP which has lowest relevancy value
3. calculate relevancy for removed HRFP globally
4. **if** (global relevancy >= Gnode_LC's lowest relevancy value)
5. add HRFP to Gnode_LC
6. **if** Gnode_LC is *full*
7. Remove the HRFP which has lowest relevancy value
8. **end if**
9. **end if**
10. **else**
11. add HRFP to Dnode_SC of Dnode
12. **if** Dnode_SC of Dnode is *full*
13. Remove the HRFP which has lowest relevancy value
14. **end if**
15. **end if**

4.3.3. Write Procedure

The following is the default write procedure to write fb on to the DFS:

1. The DFSCP on behalf of the CP running in the Dnode communicates with the Nnode to retrieve addresses of Dnodes where it has to write the fb.
2. Based on the replication factor, Nnode gives the addresses of the Dnodes.
3. The CP begins to write data in the Dnode where it is requested after obtaining the Dnode addresses. It transfers the data to the remaining Dnodes for writing purposes in a pipelined fashion.

The proposed write procedure for writing fb in the DFS is as follows:

1. If fb already exists, invalidate all the existing entries in the Dnode and Gnode caches.
2. If fb does not exist, then the default write procedure is followed to write fb.

The following Algorithm 7 explains the proposed write procedure to write fb which is initiated by a CP running in a particular Dnode.

Algorithm 7 Proposed write algorithm

1. **for** each Dnode
2. **if** fb is present in Dnode_LC, Dnode_SC of Dnode and Gnode_LC, Gnode_SC of Gnode
3. invalidate fb
4. **else**
5. write fb in Dnode_LC of Dnode using default write procedure
6. **if** Dnode_LC of Dnode is *full*
7. follow relevancy-based replacement policy
8. **end if**
9. **end if**
10. **end for**

4.4. Re-Initiation of Prefetching

Initially, all the caches present in Dnodes and Gnode are filled with the local and global HRFPs that are prefetched using the HRFP-based prefetching algorithm. The prefetching process is restarted whenever the Lcache and GLcache hit ratio of the Dnodes and Gnode falls below the threshold which is defined based on [14]. The hit ratio of all Dnodes and Gnode is monitored by a background task separately. All the other activities in the DFS environment will run simultaneously with this task. Prefetch re-initiation is done without interrupting other activities. The HRFPs are then filled in the corresponding Dnodes and Gnode caches after prefetching. Fresh requests of client programs are recorded as the latest log entries, and the old entries are deleted.

5. Simulation Experiments

First, we outline the assumptions that are considered for conducting simulation experiments in this section. Next, we discuss the procedure for data set generation and experimental setup for conducting experiments. Lastly, the simulation results are presented.

5.1. List of Assumptions

The list of assumptions made for carrying out the simulation experiments are

1. DFS with one lakh files and ten thousand file blocks per file.
2. File block size was considered as 32 KB.
3. Time to fetch a file block from local cache of Dnode is 0.0008 milliseconds (ms) [50].
4. Time to fetch a file block from SSD of Dnode is 0.0104 milliseconds [51].
5. Reading from disk of Dnode requires 3.5 ms [52].
6. 0.032 ms to read a file block from remote memory of same rack and 0.045 ms to read from different rack.
7. The delay to transfer file block from remote memory is 0.04 ms [53].

5.2. Simulation

In this section, we first explain how to generate a data set that simulates a real workload, and then we discuss the configuration needed to run the simulation tests.

5.2.1. Data Set Generation

A dataset (log) has been generated that mirrors a realistic workload following the technique described in [54]. File and block frequencies are generated using the Zipf distribution [55]. The maximum frequency for files is 500 and file blocks are set to 1000. The daily requests are simulated in intervals using a Poisson distribution. Each interval has a thousand requests. All intervals are merged according to their arrival order and considered as a complete log.

5.2.2. Experimental Setup

We have used a DFS architecture which we discussed in Section 3 to carry out the simulation experiments. A log (eighty percent read requests and twenty percent write requests) is generated [56], as stated in the preceding subsection. A file access session is represented as a log entry. we have assumed that five to fifteen blocks have been accessed by the CP in each session. From these details, we have extracted HRFPs for every file based on relevancy values. Then, the prefetched HRFPs are cached in Dnode_LC, Dnode_SC, Gnode_LC, and Gnode_SC using the procedure described in the previous section. We considered one lakh sessions of requests for every simulation run to calculate the average read access time (ARAT). We have compared the performance of our HRFP algorithm with Hadoop without caching(HDFS [57]), Hadoop with caching (Dcache) [27], SBFBAP (support-based frequent patterns prefetching and caching) algorithm [58], Support-based prefetching and (SBMS) algorithm [15] by following multi-level caching. Furthermore we also tested the HRFP algorithm performance with LRU (least recently used) [4] replacement policy rather than using a relevancy-based replacement procedure.

5.2.3. Experimental Results

The simulation results using the proposed HRFP algorithm and other existing algorithms from the literature are shown in this section.

Figure 3 shows the performance of ARAT using proposed HRFP and HRFP-LRU algorithms and the existing SBMS, SBFBAP, HDFS, and Dcache algorithms. The results are collected by increasing the size of Dnode_LC from 100 to 500 file blocks and Dnode_SC from 1000 to 5000 file blocks respectively. The size of Gnode_LC and Gnode_SC are fixed to 3000, 5000 and 30,000, 50,000 file blocks respectively.

From Figure 3, we found that the proposed HRFP and HRFP-LRU algorithms are performing better than the existing SBFBAP, SBMS, Dcache, and HDFS algorithms. This indicates that the patterns which have high relevancy value tend to be requested again and again in the future. We also found that the proposed HRFP algorithm which follows a relevancy-based replacement policy works better than the LRU replacement policy.

Next, the hit ratio (Hratio) of the local caches of Dnodes using the proposed and existing algorithms is shown in Figure 4. The Hratio is noted by varying the size of Dnode_LC in the Dnodes from 100 to 500 file blocks and the Dnode_SC size from 1000 to 5000 file blocks. The size of Gnode_LC is set to 3000, 5000 file blocks, and the size of Gnode_SC is set to 30,000, 50,000 file blocks.

We can notice that the Hratio of the local caches in Dnodes is high in the HRFP algorithm than in the HRFP_LRU algorithm, which in turn has resulted in high Hratio than the existing SBFBAP, SBMS, and Dcache algorithms. We have also observed that when the local cache size is increased, the Dnode_LC Hratio is also improved.

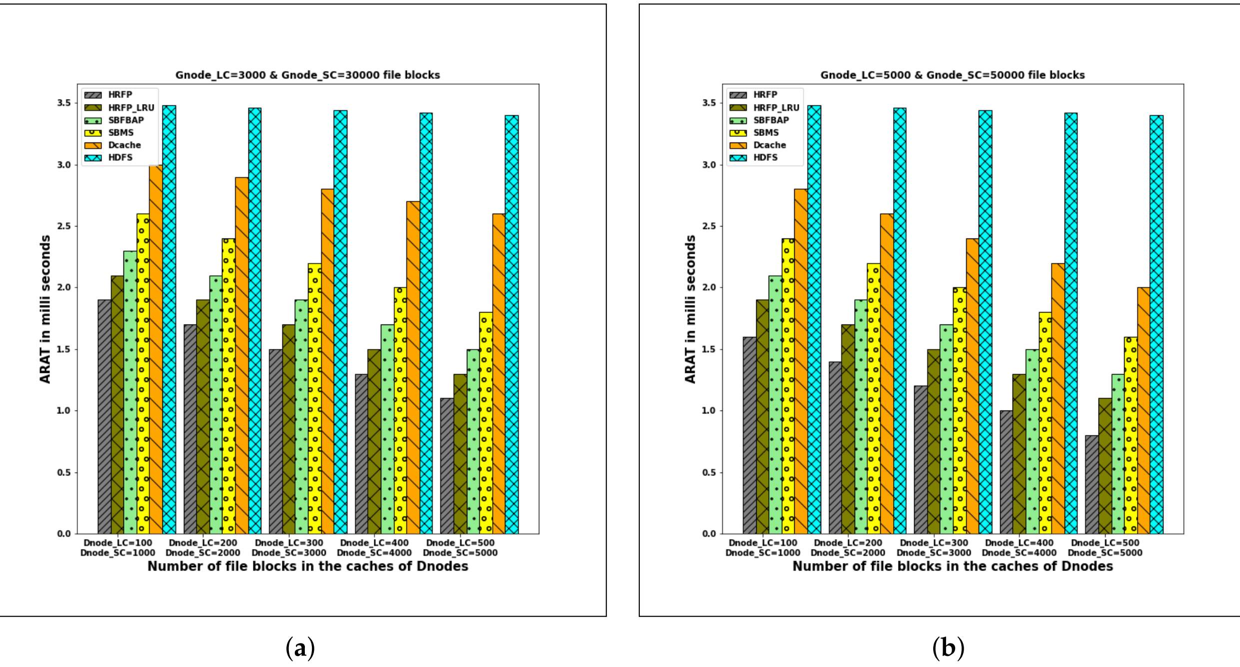


Figure 3. Figures (**a**,**b**) represent Average read access time in milliseconds when the size of Gnode_LC is set to 3000, 5000 file blocks and size of Gnode_SC is 30,000, 50,000 file blocks respectively.

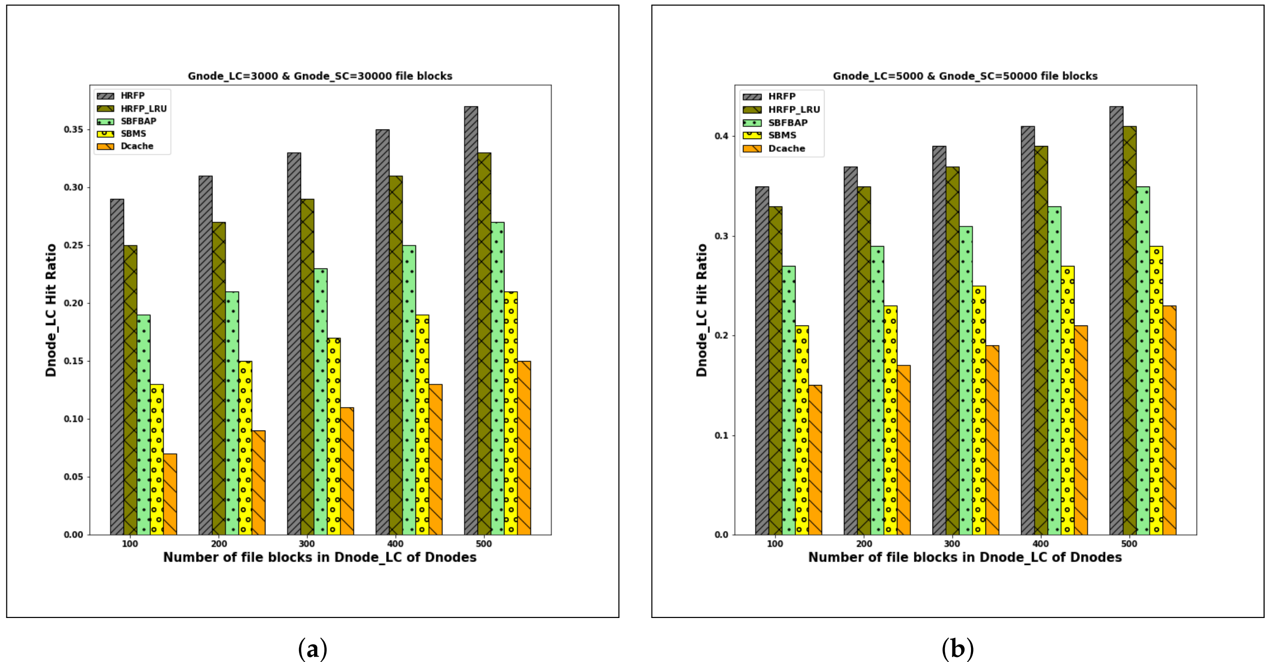
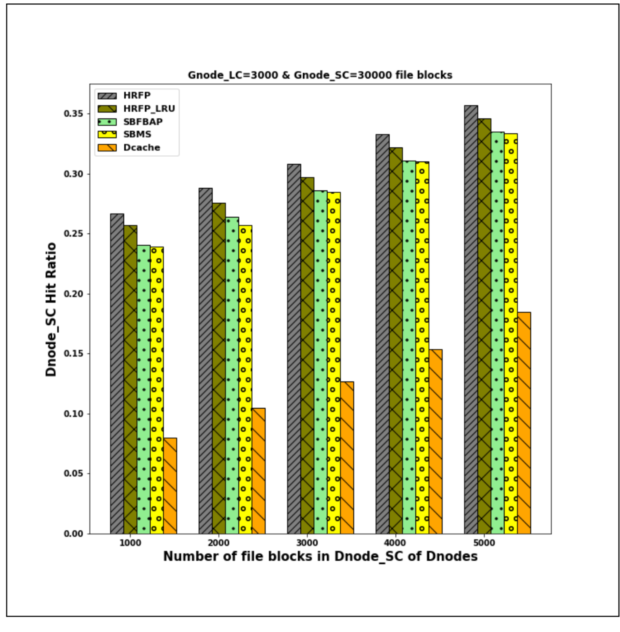
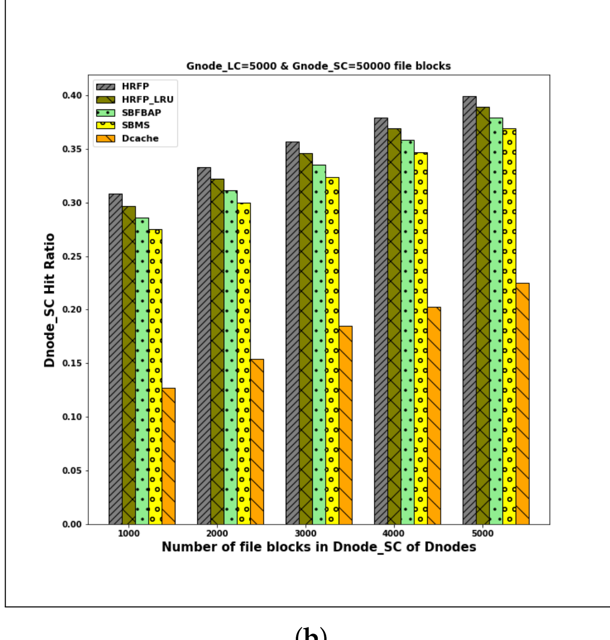


Figure 4. Figures (**a**,**b**) shows Hratio of Dnode_LC in Dnodes when the size of Gnode_LC is set to 3000, 5000 file blocks and size of Gnode_Sc is 30,000, 50,000 file blocks respectively.

In Figure 5, the trend in the Hratio of SSD caches in Dnodes is depicted. The size of Dnode_SC is varied from 1000 to 5000 file blocks and the Dnode_LC size in Dnodes is varied from 100 to 500 file blocks. The sizes of Gnode_LC and Gnode_SC are fixed to 3000, 5000 and 30,000, 50,000 file blocks, respectively.



(**a**)



(**b**)

Figure 5. Figures (**a**,**b**) shows Hratio of Dnode_SC in Dnodes when the size of Gnode_LC is set to 3000, 5000 file blocks and size of Gnode_SC is 30,000, 50,000 file blocks respectively.

We have observed high SSD cache Hratio in the proposed HRFP algorithm than in the HRFP_LRU algorithm. We have also observed that the HRFP_LRU algorithm has high Hratio than the SBFBAP, SBMS, and Dcache algorithms. The SSD cache Hratio increased if we increased the size of the Dnode_SC.

In Figure 6, the Gnode_LC and Gnode_SC Hratio for the HRFP, HRFP_LRU, and the existing algorithms are shown. The Hratio is calculated by varying Dnode_LC and Dnode_SC sizes of Dnodes from 100 to 500 file blocks and 1000 to 5000 file blocks, respectively, while Gnode_LC and Gnode_SC sizes are fixed to 3000, 5000, 30,000, and 50,000 file blocks.

We observe that the Hratio of Gnode_LC and Gnode_SC of Gnode increases with respect to the size of the cache. We have noticed high Hratio for the proposed HRFP algorithm than the HRFP_LRU algorithm. The HRFP_LRU algorithm has high Hratio than the other algorithms. From these observations, we can say that the HRFP algorithm is performing better than the other algorithms.

The ARAT performance improvement of the proposed algorithms in comparison to existing algorithms from the literature is summarised in the Table 2 below. The results shown in Table 2 indicate that the HRFP algorithm outperformed the HDFS, Dcache, SBMS and SBFBAP algorithms by 53%, 41%, 34%, and 22%, respectively. The HRFP_LRU algorithm also showed a performance improvement of 44%, 36%, 23%, 15% in comparison with the HDFS, Dcache, SBMS and SBFBAP algorithms respectively.

Table 2. Improvement of proposed algorithms performance in terms of ARAT compared with existing algorithms from literature.

Proposed Algorithms	Existing Algorithms			
	HDFS	**Dcache**	**SBMS**	**SBFBAP**
HRFP	53%	41%	34%	22%
HRFP_LRU	44%	36%	23%	15%

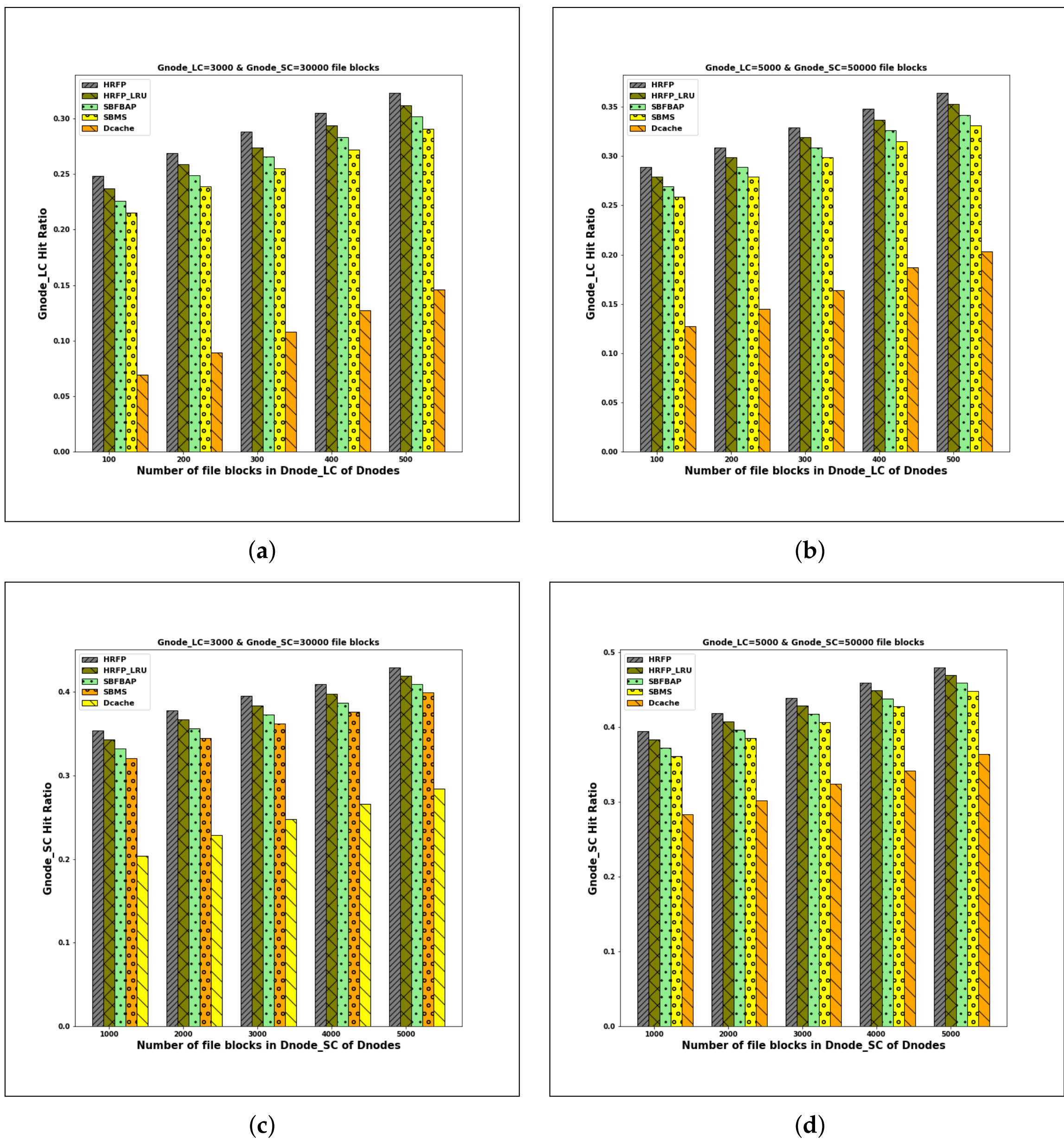


Figure 6. Figures (**a**–**d**) shows Hratio of GLcache and GScache when the sizes of them are set to 3000, 5000 file blocks and 30,000, 50,000 file blocks.

Overall, we can come to conclusion that the HRFP-based prefetching, caching, and replacement algorithm performs well than the HRFP_LRU, SBFBAP, SBMS, Dcache, HDFS algorithms with different cache sizes in Dnodes and Gnode from the obtained results. The reason for the better performance of the HRFP algorithm is that we have given importance to the highly popular patterns by considering the relevancy value. The HRFP_LRU algorithms achieved the next best performance indicating that the proposed HRFP algorithm works better with relevancy-based replacement than LRU replacement because if we replace a least recently used file block access pattern, we may lose a pattern that may have a higher relevancy value. The SBFBAP algorithm performs better than the SBMS, Dcache, and HDFS algorithms in which simple support is considered for the patterns for prefetching and caching purposes. The SBMS and Dcache algorithms showed less performance because the file blocks are prefetched instead of patterns based on simple support value and these algorithms do not follow multi-level caching. Lastly, the HDFS algorithm achieved poor performance because all the requested file blocks are served from the file system as there is no concept of client-side caching.

5.2.4. Discussion

We believe that the HRFP algorithm can be useful for social media applications like Facebook, LinkedIn, Amazon and so on. For example consider Facebook, where each user accesses this application in different ways. While accessing a page on Facebook some files and file blocks are requested from the node where the application is running. Every user produces a pattern of accessing the pages of an application. Most of the users use some pages more frequently and some pages very less frequently which forms frequent file block access patterns for every user. We generated a log based on this type of workload using medisyn, and Zipf distribution as discussed in Section 5.2.1 to test our proposed algorithms. Finally, we can conclude that the proposed HRFP algorithm with the relevancy-based replacement is performing better than the remaining algorithms as it requires less read access time than the other approaches.

6. Conclusions

In this research, we proposed highly relevant frequent pattern-based prefetching, caching, and replacement algorithms. The support value of the file and the confidence value of the pattern are used for calculating the relevancy value for the file block access patterns. After identifying the highly relevant frequent patterns, they are prefetched from the DFS and cached in the multi-level memories of the DFS environment we considered. We observed that the performance of our proposed algorithms when compared with the existing Support-based frequent block access pattern prefetching and caching algorithm, Hadoop distributed file system with and without caching, and the proposed algorithm with LRU replacement. Our simulation findings show that the proposed highly relevant frequent pattern-based prefetching and caching algorithm outperforms the other techniques addressed in the literature when employing the relevancy-based replacement policy. In the future, we wish to use the Hadoop distributed file system to implement the proposed algorithms and evaluate their effectiveness.

Author Contributions: Conceptualization, A.N. and S.K.B.; Methodology, A.N. and T.R.; Investigation and Writing original draft, A.N.; Writing review & editing, T.R., R.N. and S.K.B.; Visualization, A.N. and S.K.B.; Supervision, T.R., R.N. and S.K.B. All authors have read and agreed to the published version of the manuscript

Funding: This research received no external funding.

Acknowledgments: We want to express our gratitude to the anonymous reviewers who review the manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Stein, T.; Chen, E.; Mangla, K. Facebook immune system. In Proceedings of the 4th Workshop on Social Network Systems, Salzburg, Austria, 10–13 April 2011; pp. 1–8.
2. Wildani, A.; Adams, I.F. A case for rigorous workload classification. In Proceedings of the 2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Atlanta, GA, USA, 5–7 October 2015; pp. 146–149.
3. Mittal, S. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv. (CSUR)* **2016**, *49*, 1–35. [CrossRef]
4. Ali, W.; Shamsuddin, S.M.; Ismail, A.S. A survey of web caching and prefetching. *Int. J. Adv. Soft Comput. Appl* **2011**, *3*, 18–44.
5. Kasavajhala, V. Solid state drive vs. hard disk drive price and performance study. *Proc. Dell Tech. White Pap.* **2011**, 8–9. Available online: https://profesorweb.es/wp-content/uploads/2012/11/ssd_vs_hdd_price_and_performance_study-1.pdf (accessed on 11 November 2022)
6. Zhang, J.; Jiang, B. A kind of Metadata Prefetch Method for Distributed File System. In Proceedings of the 2021 International Conference on Big Data Analysis and Computer Science (BDACS), Kunming, China, 25–27 June 2021; pp. 115–121.
7. Chen, H.; Zhou, E.; Liu, J.; Zhang, Z. An rnn based mechanism for file prefetching. In Proceedings of the 2019 18th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES), Wuhan, China, 8–10 November 2019; pp. 13–16.
8. Liao, J. Server-side prefetching in distributed file systems. *Concurr. Comput. Pract. Exp.* **2016**, *28*, 294–310. [CrossRef]

9. Liao, J.; Trahay, F.; Gerofi, B.; Ishikawa, Y. Prefetching on storage servers through mining access patterns on blocks. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *27*, 2698–2710. [CrossRef]
10. Liao, J.; Trahay, F.; Xiao, G.; Li, L.; Ishikawa, Y. Performing initiative data prefetching in distributed file systems for cloud computing. *IEEE Trans. Cloud Comput.* **2015**, *5*, 550–562. [CrossRef]
11. 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*, 2692–2705. [CrossRef]
12. Al Assaf, M.M.; Jiang, X.; Qin, X.; Abid, M.R.; Qiu, M.; Zhang, J. Informed prefetching for distributed multi-level storage systems. *J. Signal Process. Syst.* **2018**, *90*, 619–640. [CrossRef]
13. Kougkas, A.; Devarajan, H.; Sun, X.H. I/O acceleration via multi-tiered data buffering and prefetching. *J. Comput. Sci. Technol.* **2020**, *35*, 92–120. [CrossRef]
14. Gopisetty, R.; Ragunathan, T.; Bindu, C.S. Support-based prefetching technique for hierarchical collaborative caching algorithm to improve the performance of a distributed file system. In Proceedings of the 2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP), Nanjing, China, 12–14 December 2015; pp. 97–103.
15. Gopisetty, R.; Ragunathan, T.; Bindu, C.S. Improving performance of a distributed file system using hierarchical collaborative global caching algorithm with rank-based replacement technique. *Int. J. Commun. Netw. Distrib. Syst.* **2021**, *26*, 287–318. [CrossRef]
16. Jiang, S.; Ding, X.; Xu, Y.; Davis, K. A prefetching scheme exploiting both data layout and access history on disk. *ACM Trans. Storage (TOS)* **2013**, *9*, 1–23. [CrossRef]
17. Li, Z.; Chen, Z.; Srinivasan, S.M.; Zhou, Y. C-Miner: Mining Block Correlations in Storage Systems. *FAST* **2004**, *4*, 173–186.
18. Theis, T.N.; Wong, H.S.P. The end of moore's law: A new beginning for information technology. *Comput. Sci. Eng.* **2017**, *19*, 41–50. [CrossRef]
19. Huang, S.; Wei, Q.; Feng, D.; Chen, J.; Chen, C. Improving flash-based disk cache with lazy adaptive replacement. *ACM Trans. Storage (TOS)* **2016**, *12*, 1–24. [CrossRef]
20. Yoon, S.K.; Yun, J.; Kim, J.G.; Kim, S.D. Self-adaptive filtering algorithm with PCM-based memory storage system. *ACM Trans. Embed. Comput. Syst. (TECS)* **2018**, *17*, 1–23. [CrossRef]
21. Yoon, S.K.; Youn, Y.S.; Burgstaller, B.; Kim, S.D. Self-learnable cluster-based prefetching method for DRAM-flash hybrid main memory architecture. *ACM J. Emerg. Technol. Comput. Syst. (JETC)* **2019**, *15*, 1–21. [CrossRef]
22. Niu, N.; Fu, F.; Yang, B.; Yuan, J.; Lai, F.; Wang, J. WIRD: An efficiency migration scheme in hybrid DRAM and PCM main memory for image processing applications. *IEEE Access* **2019**, *7*, 35941–35951. [CrossRef]
23. Byna, S.; Breitenfeld, M.S.; Dong, B.; Koziol, Q.; Pourmal, E.; Robinson, D.; Soumagne, J.; Tang, H.; Vishwanath, V.; Warren, R. Exahdf5: Delivering efficient parallel i/o on exascale computing systems. *J. Comput. Sci. Technol.* **2020**, *35*, 145–160. [CrossRef]
24. Yun, J.T.; Yoon, S.K.; Kim, J.G.; Kim, S.D. Effective data prediction method for in-memory database applications. *J. Supercomput.* **2020**, *76*, 580–601. [CrossRef]
25. Herodotou, H. Autocache: Employing machine learning to automate caching in distributed file systems. In Proceedings of the 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), Macao, China, 8–12 April 2019; pp. 133–139.
26. Yoshimura, T.; Chiba, T.; Horii, H. Column Cache: Buffer Cache for Columnar Storage on HDFS. In Proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018; pp. 282–291.
27. Zhang, X.; Liu, B.; Gou, Z.; Shi, J.; Zhao, X. DCache: A Distributed Cache Mechanism for HDFS based on RDMA. In Proceedings of the 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Cuvu, Fiji, 14–16 December 2020; pp. 283–291.
28. Laghari, A.A.; He, H.; Laghari, R.A.; Khan, A.; Yadav, R. Cache performance optimization of QoC framework. *EAI Endorsed Trans. Scalable Inf. Syst.* **2019**. [CrossRef]
29. Shin, W.; Brumgard, C.D.; Xie, B.; Vazhkudai, S.S.; Ghoshal, D.; Oral, S.; Ramakrishnan, L. Data Jockey: Automatic data management for HPC multi-tiered storage systems. In Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, 20–24 May 2019; pp. 511–522.
30. Wadhwa, B.; Byna, S.; Butt, A.R. Toward transparent data management in multi-layer storage hierarchy of hpc systems. In Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E), Orlando, FL, USA, 17–20 April 2018; pp. 211–217.
31. He, S.; Wang, Y.; Li, Z.; Sun, X.H.; Xu, C. Cost-aware region-level data placement in multi-tiered parallel I/O systems. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *28*, 1853–1865. [CrossRef]
32. Ren, J.; Chen, X.; Liu, D.; Tan, Y.; Duan, M.; Li, R.; Liang, L. A machine learning assisted data placement mechanism for hybrid storage systems. *J. Syst. Archit.* **2021**, *120*, 102295. [CrossRef]
33. Thomas, L.; Gougeaud, S.; Rubini, S.; Deniel, P.; Boukhobza, J. Predicting file lifetimes for data placement in multi-tiered storage systems for HPC. In Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems, Online Event, 26 April 2021; pp. 1–9.
34. Shi, W.; Cheng, P.; Zhu, C.; Chen, Z. An intelligent data placement strategy for hierarchical storage systems. In Proceedings of the 2020 IEEE 6th International Conference on Computer and Communications (ICCC), Chengdu, China, 11–14 December 2020; pp. 2023–2027.

35. Cao, T. Popularity-Aware Storage Systems for Big Data Applications. Ph.D. Thesis, Auburn University, Auburn, AL, USA, 2022.
36. Dessokey, M.; Saif, S.M.; Eldeeb, H.; Salem, S.; Saad, E. Importance of Memory Management Layer in Big Data Architecture. *Int. J. Adv. Comput. Sci. Appl.* **2022**, *13*, 1–8. [CrossRef]
37. Kim, T.; Choi, S.; No, J.; Park, S.S. HyperCache: A hypervisor-level virtualized I/O cache on KVM/QEMU. In Proceedings of the 2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN), Prague, Czech Republic, 3–6 July 2018; pp. 846–850.
38. Einziger, G.; Friedman, R.; Manes, B. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Storage (ToS)* **2017**, *13*, 1–31. [CrossRef]
39. Blankstein, A.; Sen, S.; Freedman, M.J. Hyperbolic caching: Flexible caching for web applications. In Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA, USA, 12–14 July 2017; pp. 499–511.
40. Akbari Bengar, D.; Ebrahimnejad, A.; Motameni, H.; Golsorkhtabaramiri, M. A page replacement algorithm based on a fuzzy approach to improve cache memory performance. *Soft Comput.* **2020**, *24*, 955–963. [CrossRef]
41. Sabeghi, M.; Yaghmaee, M.H. Using fuzzy logic to improve cache replacement decisions. *Int J Comput Sci Netw. Secur.* **2006**, *6*, 182–188.
42. Aimtongkham, P.; So-In, C.; Sanguanpong, S. A novel web caching scheme using hybrid least frequently used and support vector machine. In Proceedings of the 2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE), Khon Kaen, Thailand, 13–15 July 2016; pp. 1–6.
43. Wang, Y.; Yang, Y.; Han, C.; Ye, L.; Ke, Y.; Wang, Q. LR-LRU: A PACS-oriented intelligent cache replacement policy. *IEEE Access* **2019**, *7*, 58073–58084. [CrossRef]
44. Ma, T.; Qu, J.; Shen, W.; Tian, Y.; Al-Dhelaan, A.; Al-Rodhaan, M. Weighted greedy dual size frequency based caching replacement algorithm. *IEEE Access* **2018**, *6*, 7214–7223. [CrossRef]
45. Chao, W. Web cache intelligent replacement strategy combined with GDSF and SVM network re-accessed probability prediction. *J. Ambient Intell. Humaniz. Comput.* **2020**, *11*, 581–587. [CrossRef]
46. Ganfure, G.O.; Wu, C.F.; Chang, Y.H.; Shih, W.K. Deepprefetcher: A deep learning framework for data prefetching in flash storage devices. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 3311–3322. [CrossRef]
47. Cherubini, G.; Kim, Y.; Lantz, M.; Venkatesan, V. Data prefetching for large tiered storage systems. In Proceedings of the 2017 IEEE International Conference on Data Mining (ICDM), New Orleans, LA, USA, 18–21 November 2017; pp. 823–828.
48. Li, H.; Ghodsi, A.; Zaharia, M.; Shenker, S.; Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In Proceedings of the ACM Symposium on Cloud Computing, Seattle,WA, USA, 3–5 November 2014; pp. 1–15.
49. Paz, H.R.; Abdala, N.C. Applying Data Mining Techniques to Determine Frequent Patterns in Student Dropout: A Case Study. In Proceedings of the 2022 IEEE World Engineering Education Conference (EDUNINE), Santos, Brazil, 13–16 March 2022; pp. 1–4.
50. Vengeance, C. Corsair Vengeance LPX DDR4 3000 C15 2x16GB CMK32GX4M2B3000C15. 2019. Available online: https://ram.userbenchmark.com/Compare/Corsair~Vengeance-LPX-DDR4-3000-C15-2x16GB-vs-Group-/m92054vs10 (accessed on 11 November 2022).
51. Intel. List of Intel SSDs. 2019. Available online: https://en.wikipedia.org/w/index.php?title=List_of_Intel_SSDs&oldid=898338259 (accessed on 11 November 2022).
52. seagate. Storage Reviews. 2015. Available online: https://www.storagereview.com/seagate_enterprise_performance_10k_hdd_review (accessed on 11 November 2022).
53. 5020, C.N. Switch Performance in Market-Data and Back-Office Data Delivery Environments. 2019. Available online: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-5000-series-switches/white_paper_c11-492751.html (accessed on 11 November 2022).
54. Tang, W.; Fu, Y.; Cherkasova, L.; Vahdat, A. Medisyn: A synthetic streaming media service workload generator. In Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video, Monterey, CA, USA, 1–3 June 2003; pp. 12–21.
55. Nagaraj, S. Zipf's Law and Its Role in Web Caching. *Web Caching Appl.* **2004**, 165–167._19. [CrossRef]
56. Intel. Resource and Design Center for Development with Intel. 2019. Available online: https://www.intel.com/content/www/us/en/design/resource.design-center.html (accessed on 11 November 2022).
57. Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 3–7 May 2010; pp. 1–10.
58. Nalajala, A.; Ragunathan, T.; Rajendra, S.H.T.; Nikhith, N.V.S.; Gopisetty, R. Improving Performance of Distributed File System through Frequent Block Access Pattern-Based Prefetching Algorithm. In Proceedings of the 2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Kanpur, India, 6–8 July 2019; pp. 1–7.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.