Efficient Prefetching and Client-Side Caching Algorithms for Improving the Performance of Read Operations in Distributed File Systems

ANUSHA NALAJALA, THIRUMALAISAMY RAGUNATHAN, AND RANESH NAHA

1Department of Computer Science Engineering, SRM University AP, Amaravati 522502, India
2School of Computer Science, The University of Adelaide, Adelaide, SA 5005, Australia

Corresponding author: Ranesh Naha (ranesh.naha@adelaide.edu.au)

ABSTRACT Modern web applications are deployed in cloud computing systems because they support unlimited storage and computing power. One of the main back-end storage components of this cloud computing system is the distributed file system which allows massive amounts of data to be stored and accessed. In most web applications deployed in such systems, read operations are performed more frequently than write operations. Consequently, increasing the efficiency of read operations in distributed file systems is a challenging and important research problem. The two main procedures used in distributed file systems to improve the performance of read operations are prefetching and caching. In this paper, we proposed novel prefetching and multi-level caching algorithms based on the Access-Frequency and Access-Recency ranking of file blocks that were previously accessed by client application programs. We also proposed new augmented ranking algorithms for prefetching file blocks by combining the Access-Frequency and Access-Recency ranking of the file blocks. We used rank-based replacement algorithms to replace file blocks in the cache. The simulation results show that, the proposed algorithms improve the performance of read operations on distributed file systems by 29% to 77% in comparison to algorithms proposed in the literature.

INDEX TERMS Distributed file system, multi-level caching, prefetching, hadoop, HDFS, big data.

I. INTRODUCTION

In the emerging big data scenarios, most web-based information systems are deployed in cloud computing systems because of their support for infinite storage and computational capacity. Distributed File Systems (DFSs) such as Google file system (GFS) [1], and Hadoop Distributed File System (HDFS) [2] are used at the back end of cloud computing systems to store and access large data efficiently. A DFS is a client-server application that allows client applications to use the data stored on the nodes (computer systems) connected in the cloud computing environment as if the data is stored on the node where the application program is currently executed.

Most of the client application programs, including big data applications executing on cloud computing environment, perform read operations and only a few client application programs perform write operations on the DFS [3], [4], [5], [6]. There are many applications that require a large number of reads. The e-commerce applications read product images and descriptions from storage for some of them, such as retail websites where customers must view them. The best examples of these read-intensive workloads are social networking applications, where users can access text, pictures, sounds, and videos that have been read from storage. Users of databases typically perform repeated read operations on the already existing data sets but the data is not frequently altered. Note that, write operations are less in such environments than read operations. So, the performance of client application programs is enhanced if read access requests issued by them can be executed quickly. Hence, enhancing the performance
of read operations carried out on the DFS is a significant and challenging research problem. The aim of this study is to improve the efficiency of read operations carried out on the DFS by proposing efficient prefetching and multi-level caching algorithms based on Access-Frequency and Access-Recency of file blocks.

Prefetching file blocks [7], [8] from DFS is an important procedure addressed in the literature for enhancing the efficiency of read operations on DFS. The prefetching technique fetches the file blocks in advance before the client application program issues read access requests. Client-side caching is also the most commonly used technique for improving the efficiency of read operations [9], [10]. The client-side caching technique places prefetched file blocks in the caches of the nodes where client applications are executed. Note that, client-side caches are kept in the main memory devices of the nodes which are connected in the cloud computing environment. Consequently, client applications running in these nodes can read file blocks from the client-side cache instead of the DFS. It should be noted that reading file blocks from DFS will take more time than reading them from the client-side cache kept in the nodes of the main memory and Solid-state Drive (SSD).

Modern computers have multiple storage devices such as hard disks, SSDs, and main memory for storage purposes. In the current scenario, hard disks are replaced by SSDs because reading the data from SSDs is significantly faster than reading the data from hard disks [11]. However, all these memory devices should coexist in a computer system because of the cost factor. Note that, the operating system maintains the file system in hard disks and SSDs in the current computer systems. Hence, the DFS deployed in cloud computing systems must be capable of utilising both of these storage devices efficiently, and this issue is addressed in this study.

We considered that the DFSs follow a master-slave architecture and are deployed in a rack–organized computing environment. In this type of environment, the master node stores the metadata of the files and a group of data nodes (slaves), where all client applications are executed. Apart from these two nodes, we also considered a common cache node (global cache node) that can be accessed by client applications executing in the data nodes, and this node is used for global caching of data. In this paper, we have proposed ranking algorithms for file blocks based on Access-Frequency and Access-Recency. We have proposed the augmented ranking algorithms by combining both the rankings for the file blocks. We also proposed prefetching and multi-level caching algorithm for the DFS we considered. In addition, we introduced a new replacement policy based on the proposed ranking of file blocks.

The remainder of the paper is structured as follows: Section 2 discuses the related work followed by the motivation and contributions of this paper. The list of abbreviations and acronyms are presented in Section 3. In Section 4, the DFS architecture and file block ranking are described. In Section 5, the proposed algorithms are presented. The simulation results are summarized in Section 6, and the conclusion and future work are discussed in Section 7.

II. RELATED WORK

This section discusses the various prefetching algorithms and caching mechanisms explored in the literature. Subsequently, the existing cache replacement techniques are discussed. Lastly, the motivation and contributions of this paper are discussed.

A. PREFETCHING TECHNIQUES

In this subsection, we discuss various prefetching techniques proposed in the literature.

The prefetching techniques have drawn a lot of interest in the research community as they are the possible ways to reduce the average access time of I/O requests in file systems for decades. In [12], the authors introduced HR-Meta, a metadata prefetching approach based on the relationships between file access sequences. The authors did not focus on prefetching and caching file blocks in client nodes, which further reduces the DFS performance. The authors of [13] analyzed the I/O access patterns to predict the files that are to be prefetched based on machine learning approach. The disadvantage of this method is that it relies on prediction, which requires additional computations.

The initiative data prefetching method was introduced by the authors of [14], [15], and [16]. Here, prefetching was carried out at storage servers, and prediction algorithms were used to prefetch the data by considering the history of disk access as the input. In many cases, the prediction may be imprecise, and there is an overhead in storage and computation.

Data correlation-based prefetching was introduced in [17]. All correlated files with low frequency were prefetched along with the target file with high frequency, resulting in cache pollution. IPODS is an informed pipe-lined prefetching approach for distributed systems [18] in which file data is prefetched based on hints generated by web programs. To adapt to this approach, applications must be designed to generate hints for prefetching the file data ahead of time, which is an overhead for client applications.

In [19], the authors introduced Hermes, which is a distributed hierarchical I/O system. In this system, a server push approach is used to prefetch the file data, which creates unnecessary overhead on the servers and results in performance degradation. The authors introduced prefetching of popular file blocks based on the simple support value in [20] and [21]. The prefetched data (file blocks) were cached in the memory hierarchy. Multiple storage devices were not used for caching prefetched file blocks.

The prefetching techniques discussed in the literature mainly focus on prefetching popular files, file blocks, and access patterns depending on prediction, either on frequency or on the relevancy of the data. In [22], we proposed a support-based frequent file block access pattern prefetching
B. MULTI-LEVEL CACHING METHODS

Caching is the most critical stage in a system architecture to achieve faster performance. Hierarchical memory models are important for enhancing the system performance. This subsection deals with the several studies conducted in this field.

Several state-of-the-art multi-level caching techniques were proposed in the literature [24], [25], [26], [27] which mainly concentrates on caching the data based on prediction by analyzing the characteristics of I/O accesses. The authors of [28] predicted the life time of files by analyzing the access frequency of the files. The authors in [29] 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.

Some researchers introduced traditional data placement techniques [30], [31] in the multi-tiered storage systems based on the hints generated by the users which is an additional overhead for the user to generate hints.

ECI-cache, a new I/O caching technique, was presented by the authors in [32]. The caching is carried out using the useful reuse distance, which considers the type of request in addition to the reuse distance. SSDs were considered by the authors for caching in virtualized platforms. This algorithm fails to handle the frequently accessed file blocks. The main disadvantage of this approach is that it only considers reuse distance while caching the file blocks and the file blocks with short reuse distance were given higher importance without considering the access frequency of file blocks. So, this approach may not get cache hits if the users are inclined to access file blocks which are more frequent. In [33], the authors introduced a low inter-reference recency set (LIRS) algorithm based on inter-reference recency and the traditional reuse distance. In both the algorithms, caching was done based on reuse distance where the file blocks with long reused distance and infrequently accessed file blocks are kept in the caches which causes cache pollution.

Several access frequency based caching methods for big data applications have been proposed in [34], [35], [36], and [37]. Hyperbolic caching [38] 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 if the access recency of file blocks is not considered.

In [39], the authors introduced ARC (adaptive replacement cache) algorithm based on both frequency and reuse distance. Two LRU (least recently used) cache buffers are maintained, one for frequency and the other for reuse distance which is an additional overhead. The ARC algorithm stores infrequently accessed file blocks in the cache because it maintains the file blocks in frequency buffer even if they are accessed twice. This causes pollution in frequency buffer. The file blocks that have small reuse distance irrespective of time are maintained in the cache. The file blocks with small reuse distance may not be accessed recently. The authors concentrated more on reuse distance of file blocks than the access frequency.

FRD (frequency and reuse distance) [40] is also a caching algorithm based on frequency and reuse distance. This approach can handle the file blocks which are frequent and with short reuse distance efficiently. However, this approach fails to handle the frequent file blocks with long reuse distance. The authors concentrated more on reuse distance of file blocks than the frequency. When a file block which is present in the cache is accessed again, it is moved to the top of the cache irrespective of their frequency. Due to this, the file blocks with high frequency gets placed at the LRU position of the cache which may be evicted soon. So, the infrequently accessed file blocks with short reuse distance may pollute the cache.

The authors of [41] implemented a new caching mechanism for HDFS using remote memory access. A separate cache node was used to maintain a global directory of the cached data in all the data nodes, which is accessible to all data nodes in the cluster. The main disadvantage of this approach is that caching is done based on simple file block frequency.

Most of the caching methods discussed above mainly focuses on caching files and file blocks in multi-level storage based on either prediction using machine learning or based on access frequency or reuse distance or both. Most of the methods rely on reuse distance with less focus on frequency. The methods which use the combination of access frequency and reuse distance fails to address the problem of up to which extent the access frequency and recency of the data is to be considered. To overcome this, we come up with a multi-level caching algorithm which takes both access frequency and recency of the file blocks for admission into the cache. As a result, new file blocks with low access frequency that enter the cache will not be the candidate for removal from the cache, and old file blocks with high access frequency will not remain in the cache for longer periods of time.

C. CACHE REPLACEMENT POLICIES

The cache replacement policy has a significant impact on system efficiency. As a result, various cache replacement policies have varying effects on system performance. As replacement policies play an important role in memory systems, several studies have been conducted in this field.

Many replacement policies based on machine learning [42], [43], [44], [45], have been introduced in the
A. Nalajala et al.: Efficient Prefetching and Client-Side Caching Algorithms

literature. In [46] and [47], a replacement method combining machine learning classification models, such as support vector machine and greedy dual size frequency, was proposed for caching in the web. The object re-access ratio was used to make cache replacement decisions. In comparison to traditional replacement algorithms, all of these strategies provide significant benefits; however, they frequently involve more complex data structures for implementation, and some algorithms need data updates at every memory access.

Least recently used (LRU) is a replacement technique which decides the object to be replaced based on recency of the usage. In [48] the authors mentioned that LRU is mostly used replacement policy which works well either in CPU caches or in virtual storage systems. It may not perform well if the users tend to access the popular objects.

Least frequently used (LFU) [37] is a replacement policy which evicts the objects based on its access frequency. This technique keeps the most popular objects and evicts the rarely accessed objects. This technique cannot handle the file blocks with less frequency that are accessed recently.

In [21], the authors proposed a rank-based cache replacement policy. They calculated the rank of the file block based on the support and access timestamp ranks of the file blocks. Whenever there was a need for cache replacement, the file block with the least rank was evicted from the cache. The authors demonstrated that the rank-based cache replacement policy outperformed the LRU replacement policy in terms of the performance. Based on this observation, we proposed an augmented replacement policy that combines proposed ranking methods to replace file blocks when cache replacement is required.

Table 1 summarizes the related works with their limitations.

### D. MOTIVATION AND CONTRIBUTIONS

Recent studies suggested various prefetching and caching techniques based on frequency and reuse distance. Such studies neglect to handle the new data which has less access frequency and frequent data which has not accessed recently. The approaches based on reuse distance may not get good hit ratio if the users tend to access data which is more frequent. Moreover, the distance between two consecutive requests is considered as reuse distance irrespective of time and the file blocks with smaller reuse distance were given high importance without considering the access recency of the file blocks. So, the file blocks which have short reuse distance are stored in the cache by ignoring access frequency and access recency of file blocks. The frequency-based approaches may also get less hit ratio if the users are inclined to access more frequently the data which is recently accessed by them. The approaches based on both access frequency and reuse distance lacks in handling the infrequently accessed file blocks that are more recent and frequently accessed file blocks that are not accessed recently. Due to this the new file blocks which are accessed first time will always be the candidates for removal from the cache and the old file blocks with high frequency will persist for longer times in the cache.

To avoid this, we come up with new Access-Frequency and Access-Recency based ranking for file blocks. We have found that Access-Frequency based ranking of file blocks is preferable than Access-Recency based ranking of file blocks by thorough evaluation of the simulated workload. Furthermore, we proposed augmented rank algorithms for file blocks that combine Access-Frequency and Access-Recency based ranking, with the rank value determined by taking certain percentage of the Access-Frequency and Access-Recency rank values. The file blocks in the cache are ordered in decreasing order of the newly computed rank value. This type of ranking approach allows the new file blocks with less access frequency to survive in the cache and considers the old file blocks with high frequency to be victims of the cache.

The following are the contributions of this paper.
TABLE 2. List of abbreviations and acronyms.

| Acronyms/Abbreviations | Definition |
|------------------------|------------|
| DFS                    | Distributed File System |
| GPS                    | Google File System |
| HDFS                   | Hadoop Distributed File System |
| SSD                    | Solid-state Drive |
| ARC                    | Adaptive Replacement Cache |
| LIRS                   | Low-inter-reference recency set |
| FRD                    | Frequency and Reuse Distance |
| LRU                    | Least Recently Used |
| LRU_L                  | Least Recently Used |
| MND                    | Master Node |
| CCN                    | Common Cache Node |
| CCN_L                  | Local cache of common cache node |
| CCN_S                  | SSD cache of common cache node |
| DND                    | Data Node |
| DND_L                  | Local cache of data node |
| DND_S                  | SSD cache of data node |
| iid                    | File id |
| bsd                    | File Block id |
| blockid                | Block id of file id |
| PF                     | Frequent File Block |
| AF_rank                | Access Frequency Rank |
| AR_rank                | Access Recency Rank |
| AFPMC                  | Access Frequency based ranking Algorithm |
| ARPMC                  | Access Recency based ranking Algorithm |
| AFRBP                  | Augmented Frequency Rank based ranking Algorithm |
| ARRBp                  | Augmented Recency Rank based ranking Algorithm |
| Avg_freq_rank          | Augmented Frequency Rank |
| Avg_freqency_rank      | Augmented Frequency Rank |
| Avg_rank               | Augmented Rank |
| SMPC                   | Support-based prefetching and Multi-level caching Algorithm |
| DFS_CF                 | DFS client program |
| CP                     | Client application Program |
| tb                     | file block |
| ARAT                   | Average Read Access Time |
| LCC                    | Local Cache |
| SCC                    | SSD cache |

1) Proposed Access-Frequency and Access-Recency based ranking and augmented ranking algorithms to prefetch the file blocks.
2) Proposed multi-level caching algorithm which fills the caches of data nodes and the global cache node with prefetched file blocks in an efficient manner.
3) Introduced a replacement policy based on the proposed ranking of the file blocks whenever the caches of data nodes and global cache are full and there is a need to create space for storing the incoming file block.

III. ABBREVIATIONS AND ACRONYMS
The acronyms, abbreviations and their definitions used in this paper are listed in the Table 2.

IV. DFS ARCHITECTURE AND RANKING OF FILE BLOCKS
This section describes the DFS architecture first. The Access-Frequency and Access-Recency ranking measures are described next.

A. DFS ARCHITECTURE
We considered that the DFS is deployed in a rack-organized system with ten racks and ten data nodes (DNDs) in each rack. Each rack is connected to a switch and all switches of the racks are connected to a central switch. This central switch is connected to a router for accessing internet. So, all the nodes in the cluster are connected by using a local area network. We also consider that one master node (MND) and one common cache node (CCN) are available in the DFS environment. The MND maintains the metadata information of the files and their respective file blocks stored in the DNDs. The MND also has a metadata controller to handle the metadata activities and a DFS server program is installed on this node. The CCN maintains a global cache directory and is accessible to the client application programs executing in DNDs. The CCN also maintains a local cache (CCN_L) and SSD cache (CCN_S) to store the file blocks. For handling I/O requests, the CCN have controllers for the local and SSD caches. In addition, a prefetch controller is also maintained in the CCN, which is responsible for carrying out the prefetch operations based on the details of the logs preserved in DNDs. Note that, all client applications are executed in the DNDs and the DND maintains a local cache (DND_L), SSD cache (DND_S), and hard disks for the storage of file blocks. The DNDs also maintain logs in which the names of the files and identification (ids) of file blocks accessed by the client applications are recorded. The DNDs has its own controllers for local and SSD caches. The DND also has the DFS client software, which runs on the top of the file system. We assumed that 100000 files are available in the DFS, and each file consisted of 10000 blocks. The size of each file block is 32 KB, and the file block replication factor is considered to be 3 based on the replication policy proposed in [2]. The architecture of the proposed method is shown in Figure 1.

B. RANKING OF FILE BLOCKS
We name the files and file blocks which are accessed frequently as frequent files (FF) and frequent file blocks (FFBs). The FFs and FFBs can be found by analysing the logs maintained in the DNDs. We determined FFs and FFBs based on two ranking measures: Access-Frequency and Access-Recency. First, the support value for each file is calculated. FFs are the files with support value greater than the specified threshold. Then, for each file block of the FFs, a support value is calculated, and the file blocks with a support value higher than the specified threshold are referred to as FFBs. The list of FFBs of the FFs are stored in decreasing order of their support values to calculate the Access-Frequency rank for FFBs. FFBs with the same support value are arranged in decreasing order based on their access times. The position values for the file blocks are assigned from 0 to n-1, if there are n blocks in the list. Suppose that, the position of a file block is p; then, the Access-Frequency rank for that file block is computed as ((n-p)/n). Similarly, to compute the Access-Recency rank for the file blocks, they are arranged in descending order of access time, as recorded in the log. Note that, the position values for the file blocks are assigned from 0 to m-1, if m blocks are specified in the log. Suppose

126236

VOLUME 10, 2022
that, the position of a file block is $t$; then, the Access-Recency rank for that file block is computed as $((m-t)/m)$.

1) EXAMPLE

The log records details such as the names of files and the ids of file blocks requested by client applications executing in a DND. A sample log is presented in Table 3. Every row in the log is called as file access session. We can find the rack number, data node number, file name followed by the set of file block ids and timestamp value in each file access session entry. From these entries, the support, Access-Frequency rank and Access-Recency rank are computed for every file block.

The file support values are first calculated and the support for file fid specified in the log maintained at DND is calculated using equation (1), as shown at the bottom of the page. Using Equation (1), the support values for files fid1 and fid2 specified in the log of the DND are calculated as follows:

$$
support(fid1) = \frac{5}{10} = 0.5$$
$$
support(fid2) = \frac{5}{10} = 0.5$$

The files with a support value greater than the fixed threshold are considered as FFs. For example, if we consider the threshold to be 0.4, files fid1 and fid2 are considered as FFs. Next, the support values for the file blocks of FFs are calculated. The support value for a particular file block fid(bid) specified in the log of a particular DND is calculated using Equation (2), as shown at the bottom of the next page.

The support values for the blocks of the file fid1 and fid2 are calculated as follows:

$$
support(fid1(bid1)) = 0.8$$

$$
DND\_support(fid) = \frac{\text{Number of times fid appears in the log of the DND}}{\text{Total number of entries in the log of the DND}}$$  \hspace{1cm} (1)
TABLE 4. The list of FFBs.

| File id | Block id | Support | Position |
|---------|----------|---------|----------|
| fid2    | bid2     | 1.0     |          |
| fid1    | bid2     | 1.0     | 1        |
| fid1    | bid3     | 1.0     | 2        |
| fid2    | bid4     | 0.8     | 3        |
| fid1    | bid1     | 0.8     | 4        |
| fid2    | bid1     | 0.6     | 5        |
| fid2    | bid3     | 0.6     | 6        |

Note that all bids with a support value greater than the fixed threshold (0.4) are FFBs. In this example, fid1(bid1), fid1(bid2), fid1(bid3), fid2(bid1), fid2(bid2), and fid2(bid3) are FFBs. Similarly, the global support values for the files and file blocks are calculated using Equations (3) and (4), as shown at the bottom of the page.

Global FFs and global FFBs are files and file blocks with a support value higher than the specified threshold.

The following procedure is followed to calculate Access-Frequency ranks for both local and global FFBs. The FFBs in the list are sorted in decreasing order according to their support values. Note that, if there are 'n' FFBs in this list, the position of the FFBs in the list is assigned from 0 to n-1, and the same procedure is applied for global FFBs.

The Access-Frequency rank (AF_rank) for each fid(bid) stored in local FFBs list and global FFBs list is calculated as shown in Equation (5).

\[ AF_{\text{rank}}(\text{fid}(\text{bid})) = \frac{\text{list size} - \text{position(\text{fid}(\text{bid}))}}{\text{list size}} \]  

(5)

Consider the FFBs extracted from the entries in Table 3. These FFBs are sorted in decreasing order by support, and FFBs with the same support value are sorted by the access time recorded in the log. Next, a position is assigned to them, as shown in Table 4.

Considering a file block fid1(bid2) from table 4, the Access-Frequency rank for that file block is calculated using Equation 5.

\[ AF_{\text{rank}}(\text{fid1}(\text{bid2})) = \frac{7 - 1}{7} = 0.8 \]

Next, we discuss how to calculate the Access-Recency rank (AR_rank) for a file block fid(bid). This procedure is applicable to file blocks referred to in both local and global logs. The file blocks specified in the log are sorted in decreasing order of the access time. File blocks with the same access time are arranged in reverse order of the requests recorded in the session entries of the log. If a file block is referred repeatedly, only the most recently accessed block will be considered. Note that, if there are 'm' number of unique file blocks (log size) specified in this log, the positions of the file blocks are assigned one of the values from 0 to m-1. The same procedure is applied for the file blocks listed in the global log.

Access-Recency rank (AR_rank) is calculated using equation (6).

\[ AR_{\text{rank}}(\text{ fid ( bid ))} = \frac{\text{log size} - \text{position(\text{fid(bid))}}}{\text{log size}} \]  

(6)

We arranged the file blocks specified in Table 3 following the procedure discussed above, and the results are shown in Table 5.

Using Equation 6, the Access-Recency rank for a file block fid1(bid2) specified in Table 5 is calculated as follows:

\[ AF_{\text{rank}}(\text{fid1(bid2)}) = \frac{8 - 4}{8} = 0.5 \]

V. PROPOSED ALGORITHMS AND PROCEDURES

The ranking algorithms for assigning ranks to the file blocks that are requested and accessed by the client application programs are discussed in this section. Next, prefetching and multi-level caching algorithms are discussed. Subsequently, we covered the read and write procedures of DFS. Finally,
we describe the cache replacement policy used to replace the file blocks and the procedure for re-initiating the prefetch task.

A. RANKING ALGORITHMS

In this subsection, we first discuss the Access-Frequency based Ranking Algorithm (AFPMC), followed by the Access-Recency based Ranking Algorithm (ARPMC). Finally, we explain the Augmented ranking algorithms.

1) AFPMC RANKING ALGORITHM

We calculated the support values for files and their respective file blocks from the log entries recorded in the local DNDs following the procedure discussed in Section IV-B. Note that, the threshold for finding popular files and file blocks is fixed at 0.6 based on [20]. DND_FF_list is used to store files with support values higher than the specified threshold. The support value is calculated for each file block of the files stored in the DND_FF_list, and the file blocks with a support value higher than the specified threshold are saved in the DND_FFB_list. Similarly, the global support values are calculated and stored in CCN_FF_list and CCN_FFB_list.

Next, for all FFBs in DND_FFB_list and CCN_FFB_list, a position is assigned from 0 to n-1 if there are n blocks in the list. If q blocks have the same support value, they are ordered according to their access time while assigning the position. After assigning the position to the FFBs in both lists, the Access-Frequency rank for each FFB is calculated using Equation (5). The procedure for local and global ranking of FFBs is described in Algorithm 1.

Algorithm 1 The Procedure for Access-Frequency Based Ranking

```
for each fid(bid) in DND_FFB_list do
    Assign a position value from 0 to n-1 for all fid(bid)s in the list
end for
for each fid(bid) in DND_FFB_list do
    Calculate AF_rank(fid(bid)) //equation (5)
    Add fid(bid) to DND_FFB_Freq_ranklist
end for
for each fid(bid) in CCN_FFB_list do
    Assign a position value from 0 to n-1 for all fid(bid)s in the list
end for
for each fid(bid) in CCN_FFB_list do
    calculate AF_rank(fid(bid)) //equation (5)
    Add fid(bid) to CCN_FFB_Freq_ranklist
end for
```

FFBs along with their Access-Frequency rank values are stored in DND_FFB_Freq_ranklist and CCN_FFB_Freq_ranklist. These lists are sorted in descending order and named DND_FFB_Freq_ranksortlist and CCN_FFB_Freq_ranksortlist.

2) ARPMC RANKING ALGORITHM

Each row of the log (session) contains the file id and block id of the file blocks accessed by the client applications, as well as the access time. The file blocks of the files listed in the DND log are ordered in descending order of the access time. Hence, the file blocks which are accessed more recently will appear at the top of the list, and the file blocks which are accessed least recently will be placed at the bottom of the list. If a file block appears multiple times, then only block with the recent access timestamp value is considered, and the remaining ones are ignored. After ordering the file blocks in this manner, the file block ids along with their access time values are stored in a separate list named DND_AR_list. The same procedure is used to sequentially order the file blocks with respect to all DNDs and file block ids along with their access time values are stored in a list named CCN_AR_list.

We then calculated the Access-Recency rank for each file block in both lists using the Equation (6). The procedure for local and global ranking of file blocks using the ARPMC algorithm is discussed in Algorithm 2.

Algorithm 2 The Procedure for Access-Recency Based Ranking

```
for each fid(bid) in the DND_AR_list do
    Calculate AR_rank(fid(bid)) //equation (6)
    Add fid(bid) to DND_FFB_ARranklist
end for
for each fid(bid) in the CCN_AR_list do
    calculate AR_rank(fid(bid)) //equation (6)
    Add fid(bid) to CCN_FFB_ARranklist
end for
```

After computing the Access-Recency rank value for every file block, the ids of the file blocks, along with their rank values, are recorded in the DND_FFB_ARranklist and CCN_FFB_ARranklist lists. These lists are sorted into new ones called DND_FFB_ARranksortlist and CCN_FFB_ARranksortlist respectively in descending order depending on their Access-Recency rank value.

3) AUGMENTED RANKING ALGORITHMS

In this sub section, we discuss three augmented ranking algorithms: Augmented Frequency Rank (AFRPB), Augmented Recency Rank (ARRBP) and Augmented Rank (ARBP) algorithms. In these algorithms, the augmented rank is computed by combining the access frequency and recency rank values of the file block. The idea behind these augmented algorithms is that a file block with the highest frequency, which is accessed very recently, will be accessed in the near future.

a: AUGMENTED FREQUENCY RANK

The Access-Frequency and Access-Recency ranks are calculated for the file blocks and stored in DND_FFB_Freq_ranksortlist and DND_FFB_ARranksortlist. Now, the augmented frequency rank (Aug_freq_rank) is computed
for every file block in both lists by combining the Access-Frequency and Access-Recency rank values. The three-fourth value of the Access-Frequency rank and one-fourth value the of Access-Recency rank are considered for calculating the augmented frequency rank. Aug_freq_rank for a file block fid(bid) is calculated using Equation (7).

\[
\text{Aug}_\text{freq}_\text{rank}(\text{fid}(\text{bid})) = \frac{3}{4}[F_R] + \frac{1}{4}[A_R] \quad (7)
\]

where,

\[F_R - \text{AF\_rank\_value\_of\(\text{fid}(\text{bid})\)}\]

\[A_R - \text{AR\_rank\_value\_of\(\text{fid}(\text{bid})\)}\]

Similarly, Aug_freq_rank is calculated for every file block stored in the CCN_FFB_Freq_ranksortlist and CCN_FFB_ARranksortlist.

**b: AUGMENTED RECENCY RANK**

The augmented Recency rank(Aug_recency_rank) value for the file blocks is calculated by combining the Access-Frequency rank value and the Access-Recency rank value. Here, the one-fourth value of AF_rank and three-fourths value of AR_rank are used to calculate the augmented Recency rank using Equation (8).

\[
\text{Aug}_\text{recency}_\text{rank}(\text{fid}(\text{bid})) = \frac{3}{4}[A_R] + \frac{1}{4}[F_R] \quad (8)
\]

where,

\[F_R - \text{AF\_rank\_value\_of\(\text{fid}(\text{bid})\)}\]

\[A_R - \text{AR\_rank\_value\_of\(\text{fid}(\text{bid})\)}\]

**c: AUGMENTED RANK**

The augmented rank for a file block is computed using this algorithm, which combines the Access-Frequency and Access-Recency ranks as, indicated in Equation (9). Here, one-half of the Access-Frequency rank value and one-half of the Access-Recency rank value are used to calculate the new augmented rank value of file blocks.

\[
\text{Aug}_\text{rank}(\text{fid}(\text{bid})) = \frac{1}{2}[F_R] + \frac{1}{2}[A_R] \quad (9)
\]

where,

\[F_R - \text{AF\_rank\_value\_of\(\text{fid}(\text{bid})\)}\]

\[A_R - \text{AR\_rank\_value\_of\(\text{fid}(\text{bid})\)}\]

The procedure for ranking file blocks based on Augmented Rank (ARBp) is described in Algorithm 3. The same procedure is followed for the AFRBP and ARRBP algorithms.

Both the DND_FFB_Aug_ranklist and CCN_FFB_Aug_ranklist lists, which store the file block ids together with Aug_rank values, are arranged in decreasing order based on the rank.

**B. PREFETCHING ALGORITHMS AND CLIENT-SIDE CACHING TECHNIQUES**

Prefetching and client-side caching are popular methods used to improve DFS performance. In this study, we introduce novel prefetching algorithms based on file block ranking. We introduced a new client-side caching technique that keeps two caches, local and SSD caches, in the main memory and SSD of the DNDs respectively. These caches are initially filled with file blocks that are prefetched based on the proposed ranking algorithms from DFS. As a result, client applications running on DNDs can efficiently read data and continue their execution.

The procedure for prefetching and caching file blocks using the ARBP ranking algorithm from DFS to the multi-level memories of DNDs and CCN is described in this subsection. It is important to note that all remaining proposed algorithms use the same caching procedure.

1) PREFETCHING AND CACHING IN A LOCAL DATA NODE

Initially, the file blocks specified in the DND_FFB_Aug_ranklist list are prefetched from the DFS, and are filled in the local cache (DND_L) and SSD cache (DND_S) of the DND based on their respective sizes. The procedure for caching file blocks in the caches of local data nodes is described in Algorithm 4.

Algorithm 4 Caching in a Local Data Node

```plaintext
for each DND_i do
    while (DND_L_i) is not full && (DND_S_i) is not full do
        for each fid(bid) in DND_FFB_Aug_ranklist do
            calculate Aug_rank for the fid(bid)
            Add fid(bid) to DND_FFB_Aug_ranklist
        end for
        for each fid(bid) in CCN_FFB_Aug_ranklist do
            calculate Aug_rank for the fid(bid)
            Add fid(bid) to CCN_FFB_Aug_ranklist
        end for
    end while
end for
```

2) PREFETCHING AND CACHING IN COMMON CACHE NODE

The file blocks specified in the CCN_FFB_Aug_ranklist are prefetched from the DFS, and cached in the local cache (CCN_L) and SSD cache (CCN_S) of the common cache node according to their respective sizes. Algorithm 5
describes the procedure for caching FFBs in a common cache node.

**Algorithm 5 Caching in Common Cache Node**

```plaintext
while (CCN_L) is not full && (CCN_S) is not full do
    for each fid(bid) in CCN_FFB_Aug_ranklist do
        for i = 1 to MAX(CCN_L_i) do
            Add file block from DFS to CCN_L_i
        end for
        for i = [1+ MAX(CCN_L_i)] to MAX(CCN_S_i) do
            Add file block from DFS to CCN_S_i
        end for
    end for
end while
```

We have modified the support-based prefetching technique discussed in the literature [20] by considering multi-level memories to cache the prefetched data and named the algorithm support based prefetching and multi-level caching algorithm (SMPC). We also modified the Dcache algorithm (Dcache) discussed in the literature [41] by considering a global cache. We also compared our proposed algorithms with state-of-the-art FRD (Frequency reuse distance) algorithm [40] which outperformed the other classical algorithms like ARC (Adaptive Replacement Cache) [39] and LIRS (Low Inter-reference recency) [33] algorithms.

**C. READ PROCEDURE**

This subsection describes how to read a file block requested by a client application executing on a DND. First, we discuss the default read procedure followed by DFS. Next, we describe the proposed read procedure and replacement policy. Note that, we have assumed that session semantics are followed by the DFS for sharing the files among all client application programs, and this is also applicable to the write procedure explained in the next subsection.

1) DEFAULT READ PROCEDURE
The default read procedure followed by the DFS client program (DFS_CP) running in a DND, where the read request is initiated for a particular file block, is described below.

1) A file block request initiated in a certain DND is communicated to MND.
2) The MND provides the addresses of the DNDs that store the replicas of the file block.
3) After receiving the file block addresses, the DFS_CP running in the DND (cd) locates the nearest DND (nd), where the file block is available, and communicates with the DFS_CP running in nd which reads the requested file block and sends the same to the DFS_CP running in cd.

2) PROPOSED READ PROCEDURE
Suppose that, a client application program (CP) executing in a specific DND request for a file block fid(bid). First, the local cache of DND where the read initiated is checked for the fid(bid). If fid(bid) is available, it is delivered to the CP. If fid(bid) is not available, then local cache of CCN is checked. If fid(bid) is present, it is transferred to the DND where read request is initiated and delivered to CP. If fid(bid) is not present in the local cache of CCN, then SSD cache of DND where request initiated is checked for the availability of fid(bid). If it is available, fid(bid) is delivered to CP. If it is not available, the SSD cache of CCN is checked for fid(bid). If fid(bid) is present, it is transferred to the DND where the request is initiated and delivered to CP. If fid(bid) is not present, then it is read from the file system following the default read procedure.

The following flow diagram Figure 2 explains the how the request is served using the proposed read procedure.

The steps in Algorithm 6 show how this request is served in the DFS.

**Algorithm 6 Proposed Read Procedure**

```plaintext
if fid(bid) is in DND_L then
    CP reads fid(bid) from DND_L
else if fid(bid) is in CCN_L then
    fid(bid) is read from CCN_L and transferred to CP
    copy fid(bid) to DND_L
    if DND_L is full then
        Apply replacement procedure
    end if
else if fid(bid) is in DND_S then
    fid(bid) is read from DND_S and transferred to CP
else if fid(bid) is in CCN_S then
    fid(bid) is read from CCN_S and transferred to CP
    copy fid(bid) to DND_L
    if DND_L is full then
        Apply replacement procedure
    end if
else
    Apply DFS default read procedure
end if
```

3) REPLACEMENT PROCEDURE
While serving the read request, the file block is relocated to the local cache of the node where the read request is initiated. While moving file blocks from one cache to another, there is a need to create space for the incoming file block. To create space for the requested file block, certain blocks must be replaced if the cache is full. In this study, we used a replacement policy that replaces the file block with the lowest ranking to store the incoming file block. Here, the type of rank used is based on the type of ranking algorithm. For example, if we use the APMC algorithm for ranking, we consider the Access-Frequency rank value of the file block for replacement.

Algorithm 7 explains the ranking-based replacement procedure followed for local caches of DNDs.
Algorithm 7 Procedure for Ranking-Based Replacement Policy

```plaintext
Remove the fid(bid) with lowest rank from the DND_L.
if rank[fid(bid)] ≥ least rank value in CCN_L then
  add fid(bid) to CCN_L
if CCN_L is full then
  Remove the fid(bid) with lowest rank
  add fid(bid) to CCN_S
if CCN_S is full then
  Remove the fid(bid) with lowest rank
end if
else
  add fid(bid) to DND_S
  if DND_S is full then
    Remove the fid(bid) with lowest rank
  end if
end if
```

Algorithm 8 Write Procedure

```plaintext
for each DND_i do
  if fid(bid) is present in DND_L,DND_S then
    Invalidate the fid(bid)
  end if
end for
if fid(bid) is present in CCN_L,CCN_S then
  Invalidate the fid(bid)
end if
if fid(bid) is not present in DND_L,DND_S and CCN_L,CCN_S then
  Follow default write procedure
end if
```

D. WRITE PROCEDURE

The procedure for writing file blocks is discussed in this subsection. First, we discuss the default write procedure followed in DFS. Subsequently, the proposed write algorithm is explained.

1) DEFAULT WRITE PROCEDURE

We considered the file replication factor to be three based on [2]. The default write procedure followed by the DFS client program is described as follows:

1) The DFS_CP running in the DND contacts the MND to obtain the addresses of the DNDs where the writing has to be performed.
2) The MND provides the addresses of the DNDs where the requested file block must be written.
3) DFS_CP starts writing data in the DND, where its execution is initiated. After writing the data in the first DND, it is forwarded to the remaining DNDs for writing purposes.

2) PROPOSED WRITE PROCEDURE

A CP executing in a particular DND can initiate a request to write a file block fid(bid). If fid(bid) is already present, all the entries of fid(bid) are invalidated in all the caches of DND and CCN. If fid(bid) is not present, the default procedure for writing the file block in the DFS is followed. Algorithm 8 shows the steps involved in performing the write operation on DFS.

E. PREFETCH RE-INITIATION

A separate background task is initiated to monitor the local cache hit ratios in all the DNDs and CCN. This task runs continuously in parallel with all other applications and user initiated tasks in the DFS environment. Based on [20], we defined a threshold value for the hit ratio of all the specified caches.
Whenever the hit ratio falls below this value, the process of prefetching is re-initiated without affecting the other operations. After prefetching, the FFBs are filled in the respective caches of DNDs and CCN. The earlier log entries are deleted, and fresh client application program requests are added as new log entries in their place.

VI. PERFORMANCE EVALUATION

In this section, we first list the assumptions made to perform the simulation experiments. Next, we discuss the data set generation and simulation experimental setup. Finally, the simulation results are discussed in detail.

A. ASSUMPTIONS

We assumed values for the different parameters to conduct the simulation experiments as shown in Table 6.

We assumed that the DFS consists of 100000 files and 10000 blocks for each file. The size of each file block is 32 KB. The time required for reading 32 KB of data from the main memory of a DND is 0.0008 milliseconds (ms) [49], and from the SSD of the DND is 0.0104 ms [50]. If the file block is to be read from the local disk, it takes 3.5 ms [51]. Reading a file block from remote memory in the same rack requires 0.032 ms and it requires 0.045 ms to read a block from remote memory that is present in a different rack. The average communication delay for moving the 32 KB data from remote memory to the main memory of the local DND takes 0.04 ms [52].

B. EXPERIMENTAL SETUP

In this subsection, we first describe the procedure for generating a log that mimics the realistic workload, and then we cover the setup required to conduct the simulation experiments.

1) LOG GENERATION

We generated a log that is analogous to a realistic workload using the method proposed in [53]. Zipf distribution [54] is used for generating many interesting real-world synthetic workloads [37]. So, we used zipf distribution to generate the frequency of files and file blocks by fixing the frequency parameter as 0.8, and the maximum frequency of files and blocks as 500 and 1000, by fixing the scaling metric to 30. The requests in a day are modelled using a Poisson distribution in the form of intervals, where each interval consists of 1000 requests. All intervals are combined based on the arrival order and are considered a complete log with files and file block requests. We considered 100000 files and 10000 file blocks per file and the file block replication factor is fixed to 3 based on [2]. So, a total of 3000 million file blocks are considered in the proposed DFS architecture. We generated a log with one million file access sessions and each in session we assumed that 5 to 15 file blocks were accessed.

2) SIMULATION SETUP

We considered a DFS organized in the form of racks, as discussed in Section IV-A. As discussed in the previous subsection, a log is generated with 80 percentage of read operations and 20 percentage of write operations [55]. The log entries are called sessions, and in each session, 5-15 file blocks are referred. Frequently used files and file blocks are first extracted based on their support values from the log entries. Then, the Access-Frequency and Access-Recency ranks for the file blocks are computed, and the file blocks are prefetched.

We observed the DFS performance by filling the prefetched file blocks in the multi-level memories of the DNDs in DFS. We also analyzed the DFS performance using augmented ranking algorithms. The performance is observed by generating one lakh to one million requests (of which 80% are read requests and the remaining are write requests) for each run of the simulation and by calculating the average read access time (ARAT) of the DFS and hit ratio of the caches. While calculating the ARAT of the DFS, we did not consider the time taken to search for file blocks in the caches of DNDs and CCN, as it is the same for all cases.

3) RESULTS

In this subsection, the results of the simulation obtained using the modified SMPC algorithm [20], Hadoop distributed file system (HDFS) [2], Hadoop with distributed caching (Dcache) [41], Frequency Reuse distance (FRD) [40], and the proposed prefetching algorithms are presented. Subsequently, the results of the augmented algorithms are presented.

a: RESULTS OF PROPOSED ALGORITHMS

The ARAT and hit ratio of the caches in DNDs and CCN are used to assess the performance of DFS using the SMPC, HDFS, and Dcache algorithms presented in the literature, as well as the proposed AFPMC and ARPMC algorithms.

Figure 3 shows the ARAT performance of the proposed AFPMC and ARPMC algorithms, and existing SMPC, HDFS, and Dcache algorithms. The performance is evaluated by increasing the number of file blocks (fbs) in the local cache (LCC) from 100 to 500 and the number of fbs in the SSD cache (SCC) from 1000 to 5000. It is important to note that the size of the LCC and SCC of the CCN are fixed at 2500 and 25000 fbs, respectively. We observed the performance by serving one lakh to one million requests and observed the same trend in each run of the simulation.

In Figure 4, we observed the ARAT performance of the DFS by serving one lakh to one million requests by varying the size of LCCs present in DNDs from 100 to 500 fbs, while the SCC sizes of the DNDs vary from 1000 to 5000 fbs, to measure the performance. In the CCN, sizes of LCC and SCC are set to 5000 and 50000 fbs, respectively.

The proposed algorithms outperformed existing algorithms proposed in the literature in all circumstances. Note that, AFPMC uses both the frequency and the most recent access time to calculate the rank. The file blocks that have the highest support and are accessed recently have the chance to be referred again; hence, these file blocks are given a
TABLE 6. Simulation parameters.

| Parameter Name                        | Value   |
|---------------------------------------|---------|
| Total number of files in the DPS      | 100000  |
| Maximum number of file blocks per each file | 10000  |
| File block size                       | 32 KB   |
| Average communication delay           | 0.04 ms |
| The amount of time necessary to read a file block from a | Hard disk 3.5 ms |
|                                       | SSD     | 0.0104 ms |
|                                       | Main memory | 0.0008 ms |

FIGURE 3. ARAT in milliseconds when the LCC and SCC sizes of CCN are 2500 and 25000 fbs.

It is clear that the hit ratio of LCC is higher in the AFPMC algorithm than in ARPMC algorithm. The ARPMC algorithm achieves a higher hit ratio than the SMPC and Dcache algorithms. We also noted that the hit ratio increased when the LCC size is increased.

Figure 6 shows the hit ratio of SCC by varying the size of SCCs from 1000 - 5000 fbs and the size of LCCs in DNDs from 100 - 500 fbs. In CCN, the LCC and SCC sizes are set to 2500, 5000 and 25000, 50000 fbs, respectively.

The AFPMC algorithm has the highest hit ratio compared to the ARPMC algorithm, which has a greater hit ratio than the SMPC and Dcache algorithms, indicating that the AFPMC algorithm outperforms the other algorithms.
The hit ratio of LCC and SCC of CCN for the SMPC, Dcache, AFPMC, and ARPMC algorithms are presented for one million requests in Figures 7, and 8. The LCC and SCC sizes of the DNDs varied from 100 to 500 fbs and 1000 to 5000 fbs, respectively. The LCC and SCC sizes of the CCN are set to 2500, 5000, 25000, and 50000 fbs.

The hit ratios of the LCC and SCC of the CCN increased as the cache size increased. We can also note that the AFPMC
FIGURE 6. Hit ratio of SCC in DNDs with LCC and SCC sizes of CCN as 2500, 5000 and 25000, 50000 fbs.

FIGURE 7. Figures (a), (b) represents hit ratio of LCC in CCN with the LCC size as 2500, 5000 fbs and SCC size as 25000, 50000 fbs.

FIGURE 8. Figures (a), (b) represents hit ratio of SCC in CCN with the LCC size as 2500, 5000 fbs and SCC size as 25000, 50000 fbs.
method has higher hit ratio than the ARPMC algorithm, which, in turn, has the highest hit ratio than SMPC and Dcache algorithms.

b: RESULTS OF AUGMENTED RANKING BASED ALGORITHMS

We varied the size of LCC in the DNDs from 100 - 500 fbs, and the size of SCC from 1000 - 5000 fbs. The size of LCC and SCC of CCN are set to 2500 and 25000 fbs, respectively. The performance is shown by serving one lakh to one million requests in Figure 9.

Figure 10 shows the performance of the ARAT when the size of LCC in CCN is set to 5000 fbs and the size of SCC is set to 50000 fbs. The size of LCC in DNDs varies from 100 - 500 fbs, and the size of SCCs varies from 1000 - 5000 fbs. The performance is noted by serving the requests from one lakh to one million.

From Figures 9 and 10, we observe that the ARBP algorithm outperforms the AFRBP method, which in turn outperforms the FRD and ARRBP algorithms as well. The ARBP, AFRBP and FRD algorithms achieved the highest performance compared with the AFPMC algorithm which indicates that a file block that has the highest frequency rank value, which has recently been accessed, has the highest chances of access requests in the near future. We can also notice that similar trend is obtained while serving the requests from one lakh to one million.

In Figure 11, the hit ratio of the LCC of the DNDs, which are obtained by applying augmented prefetching algorithms, are presented. The LCC sizes ranged from 100 - 500 fbs in the DNDs, whereas their SCC sizes ranged from 1000 - 5000 fbs. The sizes of the LCC and SCC in the CCN are fixed at 2500 and 5000 and 25000 and 50000 fbs respectively. Here, the LCC hit ratio for the ARBP algorithm is higher than that of the AFRBP algorithm, and the AFRBP algorithm achieved a higher hit ratio than the FRD algorithm which in turn has higher hit ratio than and ARRBP algorithm. Note that the, hit ratios for the ARBP, AFRBP, and FRD algorithms are higher than those of the AFPMC algorithm.

In Figure 12, the SCC hit ratio of the DNDs is observed by applying the augmented algorithms and the existing FRD algorithm. The SCC size in the DNDs varied from 1000 - 5000 fbs, and the LCC size of the DNDs varied from 100 - 500 fbs. The size of the SCC in the CCN is fixed at 25000 and 50000 fbs, whereas the size of LCC in the CCN is fixed at 2500 and 5000 fbs.
We found that the SCC hit ratio for the ARBP algorithm is higher than that of the AFRBP algorithm, which had a higher hit ratio than the FRD algorithm which outperformed ARRBP algorithm. We also noticed that the ARBP, AFRBP and FRD algorithms had a higher hit ratio than the AFPMC algorithm.

The CCN hit ratios using the ARBP, AFRBP, FRD and ARRBP algorithms are shown in Figures 13, and 14.

**FIGURE 10.** ARAT in milliseconds when the LCC and SCC sizes of CCN are 5000 and 50000 fbs.

**FIGURE 11.** Figures (a), (b) represents hit ratio of LCC in DNDs with LCC and SCC sizes of CCN as 2500, 5000 and 25000, 50000 fbs for one million requests.
**FIGURE 12.** Figures (a), (b) represents hit ratio of SCC in DNDs with LCC and SCC sizes of CCN as 2500, 5000 and 25000, 50000 fbs for one million requests.

**FIGURE 13.** Figures (a), (b) represents hit ratio of LCC in CCN with the LCC size as 2500, 5000 fbs and SCC size as 25000, 50000 fbs for one million requests.

**FIGURE 14.** Figures (a), (b) represents hit ratio of SCC in CCN with the LCC size as 2500, 5000 fbs and SCC size as 25000, 50000 fbs for one million requests.
sizes of the LCCs and SCCs of the DNDs varied from 100 to 500 fbs and from 1000 to 5000 fbs, respectively. The sizes of LCC and SCC of the CCN are fixed at 2500, 5000 fbs and 25000, 50000 fbs respectively.

When the cache size increased, we observed an increase in the LCC and SCC hit ratios. The ARBP algorithm outperforms the AFRBP algorithm in terms of hit ratio, which is even higher than that of the FRD algorithm. The FRD algorithm also has higher hit ratio than the ARRB algorithm. It should be noted that the ARBP, AFRBP and FRD algorithms have higher LCC and SCC hit ratio compared to the AFPMC algorithm.

In addition, we also observed the performance of the DFS using all combinations of AF_rank (p) and AR_rank(q) percentages in addition to the above mentioned combinations by considering the LCC and SCC sizes of DNDs as 500 and 5000 fbs and the sizes of LCC and SCC in CCN as 5000 and 50000 file blocks respectively. The Figure 15 shows performance of the DFS in terms of average read access time.

From Figure 15, we can say that the file blocks with the rank value computed by taking 50% of AF_rank value and 50% of AR_rank value require less read access time which is the proposed ARBP algorithm. We have also found that the file block which has the highest support value and which is accessed recently, is likely to be requested in the near future.

The following Table 7 presents the details about the percentage of file blocks moved from SCC of DNDs, LCC and SCC of CCN, and from the file system to the LCC of DNDs where the requests are initiated using proposed algorithms when the sizes of LCC and SCC of DND are set to 500 and 5000 fbs and the sizes of LCC and SCC of CCN are set to 5000 and 50000 fbs respectively.

Based on the details presented in Table 7, we can see that the proposed ARBP algorithm requires less percentage of file blocks to be read from the file system when compared with other algorithms. The AFRBP algorithm has got the next lowest percentage of file system hits when compared with the AFPMC, ARRB, and ARPMC algorithms. The AFPMC algorithm has achieved the next lowest percentage of file system hits. Among the AFRBP and ARPMC algorithms, the ARBP algorithm got less percentage of file system hits when compared with the ARPMC algorithm. Less percentage of file system hits indicates that most file blocks are being read from caches maintained in DNDs and CCN rather than from file system. Reading file blocks from file system takes more time when compared to reading them from local and remote caches. Hence, the average read access time will be less if less percentage of file blocks are being read from the file system.

Overall, we conclude from the simulation results that the ARBP algorithm performs better than the AFPMC, FRD, ARPMC, SMPC, HDFS, DCache, AFRBP, and ARRB algorithms do. In the ARBP algorithm, we combined 50% of the Access-Frequency and 50% of the Access-Recency rank values to calculate the rank of a file block. This combined rank value indicates that a file block which is accessed recently with a high support value has a higher chance of receiving access requests again in the near future.

Table 8 summarizes the performance improvement of the proposed algorithms in terms of average read access time when compared with existing algorithms. The results presented in Table 8 show that the highest performance was achieved by the ARBP algorithm, with an improvement of 77%, 63%, 57%, and 29% when compared with the HDFS, DCache, SMPC and FRD algorithms, respectively. The next highest performance was observed for the AFRBP algorithm. There was an improvement of 70%, 54%, 47%, and 17% when compared with the HDFS, DCache, SMPC and FRD algorithms, respectively. The AFPMC algorithm also shows good improvements of 65%, 45%, and 36% when compared with the HDFS, Dcache, and SMPC algorithms, respectively. The ARPB and ARPMC algorithms showed smaller improvements than the HDFS, DCache, and SMPC algorithms. The AFRBP algorithm shows improvements of 58%, 36%, and 26%, whereas the ARPMC algorithm shows improvements of 56%, 31%, and 21% when compared with the HDFS, Dcache, and SMPC algorithms, respectively. The AFPMC algorithm shows less performance than the FRD algorithm because the former one is based purely on frequency of file blocks with out considering the recency. The AFRBP, and ARPMC algorithms are inclined towards caching the file blocks by giving importance to recency rather than frequency. Due to this reason, both the algorithms attained less performance when compared with FRD algorithm.
TABLE 8. Improvement of average read access time in comparison with existing approaches.

| Proposed Algorithms | Existing Algorithms | HDFS | Dcache | SMPC | FRD |
|---------------------|---------------------|------|--------|------|-----|
| ARBP                |                     | 77%  | 54%    | 57%  | 29% |
| APRBP               |                     | 70%  | 54%    | 47%  | 17% |
| APFM                |                     | 65%  | 45%    | 36%  | -13%|
| ARBP                |                     | 58%  | 36%    | 26%  | -22%|
| ARPM                |                     | 56%  | 31%    | 21%  | -29%|

VII. CONCLUSION AND FUTURE WORK

In this study, we proposed Access-Frequency and Access-Recency based ranking algorithms for ranking and prefetching. We also proposed augmented algorithms by combining Access-Frequency and Access-Recency ranking algorithms. All the proposed algorithms prefetch file blocks based on different measures to improve the performance of read operations carried out on DFS. We also proposed a caching algorithm which caches prefetched data in multi-level memories of data nodes and common cache node. Our proposed algorithms were compared with four algorithms discussed in the literature: simple support based prefetching, Hadoop with distributed caching, and Hadoop without caching and Frequency-reuse distance based caching. Based on the simulation results, we observed that our proposed algorithms outperformed other conventional algorithms. Moreover, we also observed the proposed augmented rank based prefetching algorithm exhibited the best performance. In future, we plan to implement the proposed algorithms in a real-environment to demonstrate their efficiency.

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in Proc. 19th ACM Symp. Operating Syst. Prin. (SOSP), 2003, pp. 29–43.
[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop distributed file system,” in Proc. IEEE 26th Symp. Mass Storage Syst. Technol. (MSST), May 2010, pp. 1–10.
[3] M. Dawodli, M. H. Hedayati, J. A. Baktash, and A. L. Efran, “Facebook MySQL performance vs MySQL performance,” in Proc. IEEE 10th Annu. Inf. Technol., Electron. Mobile Commun. Conf. (IEMCON), Oct. 2019, pp. 0103–0109.
[4] T. Stein, E. Chen, and K. Mangla, “Facebook immune system,” in Proc. 4th Workshop Social Netw. Syst. (SNS), 2011, pp. 1–8.
[5] A. Wildani and I. F. Adams, “A case for rigorous workload classification,” in Proc. IEEE 23rd Intl. Symp. Model., Anal., Simul. Comput. Telecommun. Syst., Oct. 2015, pp. 146–149.
[6] J. Liao and S. Chen, “Optimization of reading data via classified block access patterns in file systems,” IEEE Access, vol. 4, pp. 9421–9427, 2016.
[7] S. Mittal, “A survey of recent prefetching techniques for processor caches,” ACM Comput. Surveys, vol. 49, no. 2, pp. 1–35, Jun. 2017.
[8] N. Oren, “A survey of prefetching techniques,” Comput. Rev. Categories: VOLUME 10, 2022

126251

[9] W. Shi, P. Cheng, C. Zhu, and Z. Chen, “An intelligent data placement and learning assisted data placement mechanism for hybrid storage systems,” in Proc. IEEE Trans. Parallel Distrib. Process., vol. 5, no. 1, pp. 92–120, Jan. 2020.
[10] R. Gopisetty, T. Raganathan, and C. S. Bindu, “Support-based prefetching technique for hierarchical collaborative caching algorithm to improve the performance of a distributed file system,” in Proc. 7th Int. Symp. Parallel Architectures, Algorithms Program. (PAAP), Dec. 2015, pp. 97–103.
[11] R. Gopisetty, T. Raganathan, and C. S. Bindu, “Improving performance of a distributed file system using hierarchical collaborative global caching algorithm with rank-based replacement technique,” Int. J. Commun. Netw. Distrib. Syst., vol. 26, no. 3, pp. 287–318, 2021.
[12] A. Nalajala, T. Raganathan, S. H. T. Rajendra, N. V. S. Nikhith, and R. Gopisetty, “Improving performance of distributed file system through frequent block access pattern-based prefetching algorithm,” in Proc. 10th Int. Conf. Comput., Commun. Netw. Technol. (ICCCTN), Jul. 2019, pp. 1–7.
[13] A. Nalajala, T. Raganathan, R. Gopisetty, and V. Gaurapally, “Rank-based prefetching and multi-level caching algorithms to improve the efficiency of read operations in distributed file systems,” in Proc. Int. Conf. Big Data Analytics, Cham, Switzerland: Springer, 2021, pp. 227–243.
[14] W. Shih, C. D. Brumgard, B. Xie, S. S. Vazhkudai, D. Ghoshal, S. Oral, and L. Ramakrishnan, “Data jockey: Automatic data management for HPC multi-tiered storage systems,” in Proc. IEEE Int. Parallel Distrib. Process. Syst. (IPDPS), May 2019, pp. 511–522.
[15] B. Wadhwia, S. Byna, and A. R. Butt, “Toward transparent data management in multi-layer storage hierarchy of HPC systems,” in Proc. IEEE Int. Conf. Cloud Eng. (ICCE), Apr. 2018, pp. 211–217.
[16] S. He, Y. Wang, Z. Li, X.-H. Sun, and C. Xu, “Cost-aware region-level data placement in multi-tiered parallel I/O systems,” IEEE Trans. Parallel Distrib. Process., vol. 28, no. 7, pp. 1853–1865, Jul. 2017.
[17] J. Ren, X. Chen, D. Liu, Y. Tan, M. Duan, R. Li, and L. Jiang, “A machine learning assisted data placement mechanism for hybrid storage systems,” J. Syst. Archit., vol. 120, Nov. 2021, Art. no. 102295.
[18] L. Thomas, S. Gougeaud, S. Rubini, P. Deriau, and J. Boukhobza, “Predicting file lifetimes for data placement in multi-tiered storage systems for HPC,” in Proc. Workshop Challenges Opportunities Efficient Perform. Storage Syst., Apr. 2021, pp. 1–9.
[19] W. Shi, P. Cheng, C. Zhu, and Z. Chen, “An intelligent data placement strategy for hierarchical storage systems,” in Proc. IEEE 6th Int. Conf. Comput. Commun. (ICCC), Dec. 2020, pp. 2023–2027.
[20] T. Wang, S. Byna, B. Dong, and H. Tang, “Univistor: Integrated hierarchical and distributed storage for HPC,” in Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER), Sep. 2018, pp. 134–144.
[21] P. Cheng, Y. Lu, Y. Du, and Z. Chen, “Accelerating scientific workflows with tiered data management system,” in Proc. IEEE 20th Int. Conf. High Perform. Comput. Commun., IEEE 16th Int. Conf. Smart City, IEEE 4th Int. Conf. Data Sci. Syst. (HPCP/SmartCity/DSS), Jun. 2018, pp. 75–82.
[22] S. Ahmadian, O. Mutlu, and H. Asadi, “ECI-cache: A high-endurance and cost-efficient I/O caching scheme for virtualized platforms,” in Proc. ACM Meas. Anal. Comput. Syst., vol. 2, no. 1, 2018, pp. 1–34.
[23] J. Jiang and X. Zhang, “ECI: An efficient low inter-reference set replacement policy to improve buffer cache performance,” SIGMETRICS Perform. Eval. Rev., vol. 30, no. 3, pp. 1–21, Jun. 2002.
[24] T. Cao, “Popularity-aware storage systems for big data applications,” Ph.D. dissertation, Auburn Univ., Auburn, AL, USA, Aug. 2022. [Online]. Available: https://etd.auburn.edu//handle/10415/8270
[25] M. Dessokey, S. M. Saif, H. Eldeeb, S. Salem, and E. Saad, “Importance of memory management layer in big data architecture,” Int. J. Adv. Comput. Sci. Appl., vol. 13, no. 5, pp. 1–8, 2022.
[36] T. Kim, S. Choi, J. No, and S.-S. Park, “HyperCache: A hypervisor-level virtualized I/O cache on KVM/QEMU,” in Proc. 10th Int. Conf. Ubiquitous Future Netw. (ICUFN), Jul. 2018, pp. 846–850.

[37] G. Eíñiger, R. Friedman, and B. Manes, “Tinyfu: A highly efficient cache admission policy,” ACM Trans. Storage, vol. 13, no. 4, pp. 1–31, 2017.

[38] A. Blankstein, S. Sen, and M. J. Freedman, “Hyperbolic caching: Flexible caching for web applications,” in Proc. USENIX Annu. Tech. Conf. (USENIX ATC), 2017, pp. 499–511.

[39] N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache,” in Proc. 2nd USENIX Conf. File Storage Technol. (FAST), 2003, pp. 1–17.

[40] S. Park and C. Park, “FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance,” in Proc. 33rd IEEE Int. Conf. Mass Storage Syst. Technol. (MSST), May 2017, pp. 1–26.

[41] X. Zhang, B. Liu, Z. Gou, J. Shi, and X. Zhao, “DCache: A distributed cache mechanism for HDF5 based on RDMA,” in Proc. IEEE 22nd Int. Conf. High Perform. Comput. Commun., IEEE 18th Int. Conf. Smart City, IEEE 6th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS), Dec. 2020, pp. 283–291.

[42] D. A. Bengar, A. Ebrahimnejad, H. Motameni, and M. Golosorkhtabaramiri, “A page replacement algorithm based on a fuzzy approach to improve cache memory performance,” Soft Comput., vol. 24, no. 2, pp. 955–963, Jan. 2020.

[43] M. Sabeghi and M. H. Yaghmaee, “Using fuzzy logic to improve cache replacement decisions,” Int. J. Comput. Sci. Netw. Secur., vol. 6, no. 3A, pp. 182–188, 2006.

[44] P. Aimonthagum, C. So-In, and S. Sanguanpong, “A novel web caching scheme using hybrid least frequently used and support vector machine,” in Proc. 13th Int. Joint Conf. Comput. Sci. Softw. Eng. (JCSSE), Jul. 2016, pp. 1–6.

[45] Y. Wang, Y. Yang, C. Han, L. Ye, Y. Ke, and Q. Wang, “LR-LRU: A PACS-oriented intelligent cache replacement policy,” IEEE Access, vol. 7, pp. 58073–58084, 2019.

[46] M. Tinghuai, J. Qu, W. Shen, Y. Tian, A. Al-Dhelaan, and M. Al-Rodhaan, “Weighted greedy dual size frequency based caching replacement algorithm,” IEEE Access, vol. 6, pp. 7214–7223, 2018.

[47] W. Chao, “Web cache intelligent replacement strategy combined with GDSF and SVM network re-accessed probability prediction,” J. Ambient Intell. Humanized Comput., vol. 11, no. 2, pp. 581–587, Feb. 2020.

[48] W. Xiong, S. Katzenbeisser, and J. Szefer, “A novel web caching scheme using hybrid least frequently used and support vector machine,” IEEE Access, vol. 7, pp. 58073–58084, 2019.

[49] Corsair Vengeance. (2019). Corsair Vengeance LPX DDR4 3000 C15 2X16Gb Cmkm32Gvs4M2B3000C15. [Online]. Available: https://ram.userbenchmark.com/Compare/CorsairVengeance-LPX-DDR4-3000-C15-2X16GB-vs-Group-m0f0s54v10

[50] Intel. (2019). List of Intel SSDs. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_Intel_SSDs&oldid=898338259

[51] Seagate. (2015). Storage Reviews. [Online]. Available: https://www.storagereview.com/seagate_enterprise_performance_10k_hdd_review

[52] Cisco Nexus 5020. (2019). Switch Performance in Market-Data and Back-Office Data Delivery Environments. [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/swiches/nexus-5000-series-switches/white_paper_c11-492751.html

[53] W. Tang, Y. Fu, L. Cherkasova, and A. Vahdat, “MediSyn: A synthetic streaming media service workload generator,” in Proc. 13th Int. Workshop Netw. Operating Syst. Support Digit. Audio Video (NOSSDAV), 2003, pp. 12–21.

[54] S. Nagaraj, “Zipf’s law and its role in web caching,” in Web Caching and Its Applications. Boston, MA: USA: Springer, 2004, pp. 165–167.

[55] Intel. (2019). Resource and Design Center for Development With Intel. [Online]. Available: https://www.intel.com/content/www/us/en/design/resource/design-center.html

ANUSHA NALAJALA received the M.Tech. degree in computer science and engineering from JNTU Kakinada, Andhra Pradesh, India, in 2014. She is currently pursuing the Ph.D. degree in computer science and engineering with SRM University AP, Andhra Pradesh. She has four years of teaching experience. Her research interests include distributed file systems, big data analytics, cloud, and fog computing.

THIRUMALISAIMY RAGUNATHAN (Member, IEEE) received the Ph.D. degree in computer science and engineering from IIIT Hyderabad, Hyderabad. He has 27 years of teaching experience and 17 years of research experience. He is currently working as an Associate Dean (Engineering) at SRM University AP, Andhra Pradesh. He conducts research activities with regard to the development of a fast distributed file system for the efficient storage and retrieval of large data. He also conducts research on the task scheduling and load balancing issues in cloud computing systems and developing a health information system for disease identification and treatment plan generation.

RANESH NAHA (Member, IEEE) is currently a research fellow at the University of Adelaide. He received his Ph.D. degree from the University of Tasmania, Australia. He was awarded a Tasmania Graduate Research Scholarship (TGRS) to support Ph.D. studies. He received his M.Sc. degree from Universiti Putra Malaysia, in 2015. He was awarded a prestigious Commonwealth Scholarship provided by the Ministry of Higher Education, Malaysia. His research interests include software-defined networking (SDN), distributed computing (Fog/Edge/Cloud), network security, Blockchain and Internet of Things (IoT). He authored more than 30 scientific papers.