Hadoop-Collaborative Caching in Real Time HDFS

Meenakshi Shrivastava 1, Dr. Hans-Peter Bischof 2
Computer Science, Rochester Institute of Technology, Rochester, NY, USA

Abstract—Massive amounts of data is being produced in everyday activities hence it is necessary to store and analyze such data. Hadoop is a popular distributed system used to store this data and MapReduce is used for performing analysis on it. Detail study and experiments led to conclusion that MapReduce job’s execution times can be lowered. A cost-effective mechanism known as collaborative caching has been proposed for efficient use of resources and system. This mechanism helps in improving the performance, reducing access latency and increasing the throughput. A new architecture called Hadoop-Collaborative Caching is proposed in order to lower the execution times. It incorporates collaborative caching, reference caching and Modified-ARC algorithm. Each of the DataNodes have their own dedicated Cache Manager that manages caching, replacement, collaborative caching and eviction. Cache is organized in to recent, frequent, recent history and frequent history. To evaluate, results obtained were compared with default configuration of Hadoop.

Keywords: Hadoop Collaborative Caching, Distributed Computing, MapReduce

1. Introduction

Data is being generated at an enormous rate, due to online activities and use of resources related to computing. To access and handle such enormous amount of data spread, distributed systems is an efficient mechanism. Hadoop is a widely used distributed system, follows clustered approach, highly scalable and it allows massive amounts of data to be stored. Hadoop follows the master/slave architecture decoupling system metadata and application data where metadata is stored on dedicated server NameNode and application data on DataNodes. Over the years, Hadoop has gained importance because of its scalability, reliability, high throughput and performing analysis and large computations on these massive amounts of data. Currently, Hadoop is being used by all the leading industries like the Amazon, Google, Facebook, Yahoo etc. Hadoop’s filesystem architecture and data computational paradigm has been inspired by Google File System and Google’s MapReduce[3]. At Yahoo, there is a of span 25,000 servers, and stores 25 petabytes of application data, with the largest cluster being 3500 servers[11]. In a paper presented at Sigmod [2], describes how Facebook is using Hadoop in real time, with only few modifications made to it, it provides high throughput and low latency.

1.1 MapReduce

Hadoop uses MapReduce paradigm [20] to perform analysis, transformations and parallel computations on the data stored. MapReduce is a parallel processing framework which divides the input into smaller inputs and execute tasks on it simultaneously hence achieving higher performance[10]. But according to PACMan [7], the initial phase which is map phase in MapReduce involves reading raw data from the disk and this task is I/O intensive. According to recent summit by Cloudera in June 2012, Optimizing MapReduce Job Performance [4], explains that certain optimizations can be made in order to improve the processing of MapReduce task. Also in paper "Optimizing Hadoop for the cluster" by Christer [5], mentions that the default configuration is slow and optimizations are needed. In a paper, "The Performance of MapReduce: An In-depth Study" [6] clearly mentions that MapReduce is slow, the processing execution time can be improved by adding more nodes to the cluster, but that is not a cost effective solution. There were two observed facts after the initial analysis of MapReduce experiments; the data during the initial phase was being read from disk and there were more Rack-Local tasks scheduled.

1.2 New Approach

It is known that accessing data from cache is much faster as compared to disk access. Hence to lower the job execution time of MapReduce jobs and improve the overall cluster efficiency of Hadoop system, improvements and architectural changes were incorporated in Hadoop Distributed filesystem which led to new system, Hadoop-Collaborative Caching. The new approach followed in order to lower job execution times was collaborative caching on DataNode. Collaborative caching is one such mechanism in which the cache distributed over the clients or dedicated servers or storage devices form a single cache to serve the requests. This technique led to more data-local jobs. Not only local data was cached on DataNodes and served as an input to MapReduce jobs but information about data cached on remote caches was stored on DataNodes, introducing a new layer to hierarchy resulting in NameNode cache, DataNode’s cache, Remote DataNode’s cache and the disk. Caches of all the participating DataNode’s machines taken together formed a single cache or global cache. NameNode is the central co-ordinator of this global cache, but allowing the decisions of remote caching to be taken by Cache Manager on DataNodes. New DataNode protocols have been introduced. New Approach allows efficient use of resources.
where instead of increasing the number of nodes, more slots can be added in order to improve performance.

Caching of data was made faster and easier by the reference caching technique. A HDFS block is composed of two files which is meta file and the block file. Meta file refers to checksum value of the data and block file refers to actual data. If these file references are cached, it helps in locating the meta files faster for checksum checks, faster caching of data from disk to memory since not much time is spent in searching for files when data stored is about petabytes. This mechanism provides an additional method of reducing the overall time.

Modified-ARC cache replacement policy was used in order to maximize cache hit ratio and to improve efficiency.

The organization of the paper is as follows; Section 2 explains the Related Work, Section 3 New Architecture, Section 4 Modified-ARC, Section 5 Data Flow between the different components of the newly proposed system, Section 6 Factors leading to improvement in the system, Section 7 Evaluations and Section 8 Concludes.

2. Related Work

According to PACMan, when multiple jobs are run in parallel, job’s running time can be decreased only when all the inputs related to running a job are cached. So according to Dhruba[7] et.al, either cache all the inputs related to that particular job or do not cache the inputs at all. Caching only part of the inputs will not help in improving the performance. These massive distributed clustered systems have large memories and job execution performance can be improved if these memories can be utilized to the fullest. PACMan is a caching service that coordinates access to the distributed caches. This service aims at minimizing total execution time of job by evicting those items whose inputs are not completely cached. For evicting the inputs which have been minimally used, LFU-F algorithm, LIFE sticky policy have been proposed. They define a new parameter wave-width of the job which refers to total tasks which can be executed in parallel at a time. They have used MapReduce and Dyrad as examples to illustrate their algorithm and hypothesis. According to them reading the raw input from filesystem is IO intensive and forms 79% of the phase. They conducted experiments on Hadoop and observed improvement in job execution time. They emphasize on memory-locality tasks which is an important factor contributing to cluster efficiency.

Dhruba[7] et.al proposes an architecture called the PACMan which coordinates the caches globally and it takes care of two things which is support queries where block is cached and coordinating the cache replacement. Its architecture includes a coordinator service and PACManClients are located on nodes where data lies. Blocks are cached on these clients. The coordinator includes the information regarding this block belonging to which file and wave-width of the file. This overall structure of the coordinator is used for scheduling tasks which are memory local, in implementing sticky policy[7] and to check on the incomplete files. If the data is not cached then it accesses disk. Also it emphasizes to schedule a data local job. For replacement policies they go for global cache replacement policies which are LIFE and LFU-F[7]. The overall job execution time is reduced attempting to schedule jobs of smaller-wave widths. This system does not take into account the remote caching.

The main aim defined in paper Dynamic Caching [16] is caching mechanism allowing concurrent access to data and proposes algorithms relating to locality of data which focuses on the decrease in the overall job completion time. For implementing Dynamic caching they are using Hadoop and their caching mechanism is based on Memcached[16] which are a set servers storing the mapping of block-id to datanode-ids. These servers also serve the remote cache requests. The caching of blocks is carried out on DataNodes.

The paper mentions about two different design architectures, First architecture defines; to serve the request of DataNode, simultaneous requests are sent to NameNode and Memcached servers. DataNode receives reply from both of them, but it checks if true from Memcache then access block from Memcache else access the block from disk whose location as indicated by NameNode. In second design architecture, again simultaneous request is sent to Memcache and NameNode, but NameNode does not reply back until indicated by Memcache about unavailability of block where in such a case NameNode sends block locations to DataNode. The design includes prefetching where whenever request is seen by Memcache, neighbouring blocks are also looked up. If neighbouring blocks are missing then Memcache requests NameNode to look for replicas and if available, requests are sent to DataNode to cache blocks and Memcache updates it’s locations. The caching system designed is not distributed and for lookups it is always required to contact the single set of Memcache servers. Also there incurs an extra delay with respect to second architecture when Memcache does not contain the blocks in cache.

3. Hadoop-Collaborative Caching: Architecture

Following section explains in detail the Hadoop-Collaborative Caching architecture; added functionalities to already existing components and newly added components. Fig.1, shows diagram of Hadoop-Collaborative Caching system architecture.

3.1 Cache Manager

Each of the DataNodes have their dedicated CacheManagers who have responsibilities of managing caches, lookup in local as well as global cache image upon request, replacement policy for cache and eviction policy for cache when cache is fully utilized. A buffer is maintained to cache the
file references which are meta file and block file. Blocks in the cache will be replaced when the cache is fully utilized and eviction in either of the caches will take place in LRU manner. Cache is divided into recent, frequent, recent history and frequent history.

3.2 Global Cache Image

Global Cache Image is a mapping of block to DataNode, denoting that this block is cached on which DataNodes in the cluster. This mapping is maintained by NameNode and copy of it is sent to all the DataNodes as a response to cache block report. Each of the DataNodes maintain a copy of Global Cache Image as well. Global Cache Image lookup is done by Cache Manager upon local cache miss.

3.3 NameNode

NameNode is the central co-ordinator for maintaining the Global Cache Image. It builds its global cache image when it obtains the cached block report from the DataNodes. As soon as it obtains its report, it updates the mapping of cached block to DataNode. Upon updation of the Global Cache Image, as a response to cached block report it sends a copy of Global Cache Image to the DataNode via DNA_UPDATE_GCI command.

3.4 DataNode

DataNode provides a cached block report of its local cache to NameNode after periodic interval. As a response to this report NameNode commands DataNode to update global cache image.

3.5 DFSClient

DFSClient receives set of caching DataNodes along with non caching DataNodes from NameNode as response to its request to read a particular file.

3.6 New DataNodeProtocols

These OP Codes have been introduced as part of the collaborative caching mechanism and as DataNode transfer protocol. The receipt of these OP Codes help in determining the next set of steps to be taken by DFSClient or DataNode.

- **OP_READ_BLOCK_CACHED**: To indicate that DFSClient is attempting to read the data from cache.
- **OP_STATUS_BLOCK_CACHED_ELSEWHERE**: DataNode sends this to DFSClient to signify that this block is cached on a different node.
- **OP_STATUS_BLOCK_NOT_CACHED**: DataNode sends this to DFSClient to signify that this block is not cached at all.
- **DNA_UPDATE_GCI**: This is the instruction sent by NameNode to DataNode in response to the cached block report.

4. Modified-ARC algorithm

The following section explains the Modified-ARC algorithm in detail. Fig 2 shows the diagram of Modified-ARC. A variant of this algorithm is implemented. Basic idea is to divide the caches into two different sections namely cached objects and history objects. Cached section contains the actual data and History section contains the references of evicted items. Hence the cached section is further divided into Recent Cache and its Recent History and Frequent Cache and its Frequent History. The size of recent and frequent together is fixed. The idea derived from actual ARC algorithm[9].

- **Recent Cache**: A cache where the block seen for the first time is placed.
- **Frequent Cache**: A second reference to the same block will cause the block to be placed in this cache.

The basic idea behind is:

- Initially on a request for block, check for references in either of the history caches, if present then place their blocks in recent or frequent cache, else cache references and serve request from either of the history caches which helps in faster caching as well as locating the files for initial checks.
- If references are found in the recent history then it is used to cache the block and place it in recent cache. If block is found in recent cache, then place it in frequent
cache, hence hit in either of the history caches removes the references and places the corresponding block in either of the caches (recent or frequent). Caching of block involves caching metadata as well data.

- When either of the caches are fully utilized then block is evicted from recent or frequent cache but its reference is placed into its corresponding history. When either of the history caches are fully utilized causes the references to just drop out of the cache.

5. Data Flow

In the following section, detailed explanation is provided of interaction between the components DataNode, DFSClient and NameNode with respect to new functionalities implemented.

5.1 NameNode and DFSClient Data Flow

DFSClient requests NameNode for blocks locations depending on the file and offset. Upon the request, NameNode returns block’s id and set of locations where these blocks are located along with replicated blocks locations. If the blocks are cached, then it returns set of caching DataNodes as well as non caching DataNodes. DFSClient tries to connect to best possible node by sorting in the order such that caching DataNodes are first in the list and then Non Caching DataNodes.

5.2 DFSClient and DataNode Data Flow

![Diagram](image)

Fig. 3: DFSClient and DataNode Interaction Sequence Diagram

Fig3. shows the sequence diagram of data flow between DataNode and DFSClient. After obtaining locations of the blocks and their block ids from NameNode, DFSClient sorts them in the order such that caching DataNodes are first in the list. It checks for total cached read attempts, if maxed then connect directly to non caching DataNode. It checks if the DataNode it is trying to connect was previously declared dead. A dead DataNode is a node when the client tried to read earlier and after maxed attempts, the DFSClient failed to connect to DataNode. Similar to deadnodes are cached deadnodes which indicates that particular DataNode previously had it cached but cached block was removed and the block could not be obtained. It sorts and tries to connect to best DataNode possible.

A connection is established between the first chosen DataNode in the list and DFSClient. If DFSClient is reading the cached block then it writes opcode OP_READ_BLOCK_CACHED to the DataNode. DataNode in turn instantiates DataXceiver and forwards the opcode sent by DFSClient. If cached block is found then opcode OP_SUCCESS is sent followed by data using BlockSender and is forwarded to DFSClient so that it can start reading data.

If the cached block is not found then send OP_STATUS_BLOCK_NOT_CACHED opcode or OP_STATUS_BLOCK_CACHED_ELSEWHERE to DFSClient. Opcode OP_STATUS_BLOCK_NOT_CACHED indicates no cached block can be found and opcode OP_STATUS_BLOCK_CACHED_ELSEWHERE indicates this block can be obtained in remote cache. This opcode is followed by the socket address of the DataNode to connect to. This is where collaborative caching helps, attempting to read from neighboring cache.

After receiving the opcode OP_STATUS_BLOCK_CACHED_ELSEWHERE, it again checks for cache read attempts, if not maxed out then try reading from cache specified by other DataNode, if caching DataNode is available. If opcode OP_READ, then read the data from non caching DataNode. If the read is for the first time, DataNode indicates CacheManager to cache the block’s metadata as well data.

5.3 NameNode and DataNode Data Flow

DataNode sends NameNode cached block report about its locally cached blocks after a certain configured interval. This is sent in the form of CachedBlocksCommand. Accordingly Global Cache Image in the namesystem is updated. As a response to this command NameNode sends DataNode instruction to updates its Global Cache Image with the help of DNA_UPDATE_GCI command.

6. Improvements in Hadoop

The proposal to implement collaborative caching and integrating with Hadoop works. It proved to be successful resulting in a considerable lower job execution times which led to improvement and enhancement in the overall system.

Reading data from cache is always faster as compared to reading data from the disk. On the DataNode side, the data was being streamed from disk hence the overall performance
of a MapReduce job was considerably slow and had an overall high I/O rate. An attempt has been made to cache this data and stream from cache. Moreover, collaborative caching allows us to stream the data from remote caches which proves to be an added advantage. In Hadoop, it was also observed that caching references added to the overall improvement of the system performance. There are three main reasons that contribute to improvement in the system.

a) Remote Memory Caching: Caching of input data at the DataNode level helps in improving job execution time. A distributed cache structure is followed where each of the DataNodes have their own caches maintained by Cache Managers. We are utilizing the memories of all the participating DataNodes thus reducing the no. of disk accesses. If the data is not found in requested node’s cache but found in other node’s cache, so instead of serving from disk we are serving from cache and which saves us execution time. This approach not only reduces job execution time, but also helps us to utilize the resources efficiently. To improve the performance, instead of adding more nodes, we can focus on adding more slots causing more map and reduce tasks to be scheduled at once parallely and with the technique of collaborative caching, data is available in cache causing in overall lowering of the job execution timing.

b) More data-local Tasks: The second reason is, whenever JobClient submits the Job, JobTracker tries to schedule the job on the same node as the input. More the data-local tasks, better the execution time. So as soon as TaskTracker contacts JobTracker either with slot available or no, if the slot is available then schedule a task. If the input required for the task resides on a different node, then it is rack-local and in such a case the data is streamed from other node to the node where the task is scheduled and then the task is carried out. This results in added extra time to complete the task resulting in increasing the overall job execution time. But with caching, the tasks scheduled get completed earlier and slots become available faster which provides room for tasks to be scheduled as data-local, hence improving the overall execution time. Also in terms of collaborative caching, the data is streamed from neighbouring node’s cache, although there is n/w I/O involved, but minimal and moreover, it is from cache hence it reduces the overall time.

c) Reference Caching: The third reason was, initial request served by the references which were cached during the request contributed to the improvement as well. This is because, these references help the system to obtain the data into cache faster since disk lookup with large number of files stored incurs extra time and with reference caching, this lookup is saved. Hence, the effect of caching was visible the first time itself. When first time a request is encountered and a cache check is made for the request, if data is not found in cache causes the data to be cached with the help of reference caching. The main advantage here is that with the initial request, only certain number of bytes are read and not the actual data. The same request for the same block comes again and this time it is already cached, so the data is streamed from cache. But with first time requests, there incurs extra time to cache the block although difference in timings due to reference caching can be observed during initial execution. The second time, a request for the same block and with data available and streamed from cache, a considerable decrease in the timing can be observed.

With the Modified-ARC cache replacement algorithm, better cache hit ratios were observed, a factor leading to overall improvement.

d) Collaborative Caching Works: The overall improvement is also result of the caching algorithm. DFSClient always attempts to sort so as to have caching DataNode first in the list indicates that attempt is made to always stream from cache. Moreover, if DFSClient cannot find the block in the local cache of this DataNode, it can connect to next location provided by DataNode for streaming from cache.

7. Evaluation

This is a preliminary work. The experiments were conducted with limited number of nodes and hardware. Experiments were performed on default configuration of Hadoop and compared with results obtained on execution of jobs on Hadoop-Collaborative Caching.

The following section explains the configurable parameters used while conducting experiments.

7.1 Configurable Parameters

Following are the list configurable parameters used while conducting experiments:

- Local Cache Size: To calculate the cache hit and miss rates, recent cache size and frequent cache capacity, parameters were configured from 6 - 24 blocks depending on the block size.
- Block Size: To perform experiments on cluster, the block sizes used 32 MB, 64 MB, 128 MB.
- Minimum Block Size For Caching: Block size >1 MB
- History Cache Size: This size was configured to 100.
- Max Read Cached Attempts: This number is set to 2.
- Caching Enable: To enable or disable caching.
- Cache Block Report Interval: Configured as 10 secs.

7.2 Metrics

Following section explains the metrics used while conducting the experiments:

- Average Block Access Time: Hadoop has large block size (default 64MB), it does not read all the data at once, instead it streams the data in the form of packets.
and then sends over the network to recipient. Hence for Hadoop the total average block access time is
\[ \text{Total Average Block Access Time} = \text{Time to read from disk or cache} + \text{Time to transmit the data over the network.} \]

- **Cache Hit / Miss Rate:** It is used to measure how many times a block was read from the cache. The higher the number, the better it is. Cache hit rate is calculated:
  \[ \text{Hit Rate} (\%) : \frac{\text{Total no. hits}}{\text{Total number of requests for blocks}}. \]
  \[ \text{Total Hits} (\%) : \frac{\text{Total no. hits in local cache (recent + frequent) and global cache}}{\text{Number of requests}}. \]
- **Local Cache Size:** Local Cache size is a combination of Recent cache size and Frequent cache size.
  \[ \text{Local Cache Size} : \text{Recent Cache Size} + \text{Frequent Cache Size}. \]
- **Max Read Cached Attempts:** This number indicates the maximum number of times we connect to DataNode to read from cache before we finally read from the disk.
- **Cache Block Report:** It is a report sent by all the DataNodes regarding information about their local caches to NameNode. This helps NameNode construct the Global Cache Image.

### 7.3 Experiments and Results

The experiments described below use WordCount application, in order to test the new improvements incorporated. The WordCount application counts the frequency of words for a given input. Inbuilt `time` command of Linux was used to compare the results. The cluster was restarted when files of greater data size was run due to memory limitations.

e) **MapReduce:** Following explains in detail of how the MapReduce experiment was conducted, results obtained and explanation about the observations made.

**Observations and Results:** It can be observed that Hadoop-Collaborative Caching results obtained showed a considerable improvement. From Fig. 4, it can be seen that with increase in file size, the map reduce timings have decreased. It can also be observed that there are drops in timings when file size is 750 MB and 950 MB. Also it is observed that with increase in file size, difference in the job execution time of default Hadoop and Hadoop-Collaborative Caching increases. Hence collaborative caching shows a significant improvement in MapReduce job execution times.

**Explanation:** This is due to the fact that streaming from cache is faster as compared to from disk which is combined with serving requests from references. With increase in file size, no. of blocks that make up the block increases and if the blocks are found in cache (local or remote) instead of accessing from disk, the overall execution time decreases resulting in a larger gap between Hadoop and Hadoop-Collaborative Caching. The other major factor leading to decrease in overall job execution time is job being executed as data-local tasks. Job can be executed as data-local and rack-local. When jobs are executed on the same node as the input, it is data-local task whereas when the job is executed on node other than the input it is rack-local job. Execution of data-local jobs are faster as compared to rack-local. In Rack-Local jobs, when job is executed data is streamed from other node hence increasing the overall timing. So more the rack-local jobs, more time required to complete the job. The reason for execution of rack-local jobs is; as number of slots are fixed, for slots are being used by TaskTrackers, if JobTracker receives a request from TaskTracker and it has an empty slot, an attempt is made to schedule a data-local task. But if that is not possible then rack-local task is scheduled. In case of collaborative caching because of the raw data in memory, jobs are executed and completed earlier and hence slots become available earlier leading to more of data-local tasks. The next factor which acts as an advantage is during the first run with Hadoop-Collaborative Caching implemented, the caching effect is seen. This is because Hadoop initially reads 516 bytes and then the whole data. When the initial bytes are read, with the help of reference caching block is cached and when the data is to be read, it is read from cache. Although there incurs extra time when the block is cached for the first time and successive requests for the same blocks results in higher decrease in timings. A drop in timings from file size 900 MB to 950 MB, which is due to fact the way jobs are scheduled and the data distribution of the blocks when the input files are loaded into HDFS. The overall decrease in timings was also due to reference caching and Modified-ARC algorithm.

**Fig. 4:** MapReduce Job Execution (Block Size 64 MB)

**Experiment Conducted:** For MapReduce experiments, MapReduce Job was run on datasets ranging from 500 MB - 1 GB on default as well Hadoop-Collaborative Caching in cluster mode. It was conducted for block sizes 32MB, 64MB and 128 MB. Cluster was restarted for Hadoop-Collaborative Caching when MapReduce Job was run for data files greater than 650 MB, due to memory limitations. Fig 4 depicts graph for MapReduce Job Execution for block size 64 MB.

The results varies depending on the distribution of blocks across the cluster and the way the jobs are scheduled across the cluster.
There seemed to be overall high n/w I/O so caching results can be improved to a larger extent with higher network speed of 1 Gbps. With increase in file size, the MapReduce job execution time decreases. More data-local tasks lead to better execution time.

f) Multiple Clients Execution: Following explains in detail of how the Multiple Client Execution experiment was conducted, results obtained and explanation about the observed results.

![MapReduce Job Execution](image-url)

Fig. 5: Multiple Job Execution (File Size 500 MB)

**Experiment:** This experiment was conducted to run jobs parallelly such that each of them are run in background. The maximum jobs which could be run was 6 due to memory limitations. Fig 5 shows graph for Multiple Job Execution time for multiple clients. This experiment was carried out for file size 500 MB, where multiple clients are trying to run MapReduce task for file size 500 MB.

**Observations and Results:** Above graph shows simultaneous clients submitting the job. It can be clearly observed that Hadoop-Collaborative Caching shows better results as compared to default Hadoop configuration. It can also be observed that execution time of Hadoop-Collaborative Caching is consistently low.

**Explanation:** The reason for only having 6 max jobs in parallel is due to infrastructure and memory limitations. The machines could not launch more than 6 JVMs limiting the experiment to run only 6 jobs at a time. The execution time is low for Hadoop-Collaborative Caching due to the reason that data is available in cache causes more data-local tasks which causes overall decrease in the job execution time.

8. Conclusions

Our next steps would be to expand this preliminary set of experiments to larger cluster. A new architecture has been proposed named as Hadoop-Collaborative Caching. The architecture aimed at improving the overall MapReduce job execution time by lowering it and increasing the efficiency of the system. This was done through the mechanism of collaborative caching where data is served from local caches as well as remote caching. This was combined with caching references and Modified-ARC algorithm. NameNode’s response to client was modified to send cached locations as well to the non cached locations of DataNode. NameNode maintains the Global Cache Image which is image of location of all the cached blocks on the cluster. This is mapping of cached block to DataNodes indicating this block is cached on these DataNodes. On the DataNode side, each of them have their dedicated Cache Managers which have responsibilities of caching data, replacement policy which is Modified-ARC and caching of references. Caching of the references also improved the overall system execution. The cache was divided into four sections recent, recent history, frequent and frequent history instead of maintaining just a single cache. This mechanism helped in better caching replacement. Hence overall, the job execution time decreased by a considerable amount, efficiency of the system increased and there were more data-local jobs scheduled.

References