Journal of Theoretical and Applied Information Technology
31st August 2017. Vol.95. No.16


ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

# HDFS CACHE PERFORMANCE USING SET ASSOCIATIVE CACHE MEMORY

[1]B.PURNACHANDRA RAO,[2]Dr.N.NAGAMALLESWARA RAO

[1]Research Scholor, Dept of Computer Science & Engg,ANU College of Engg & Technology,Guntur,India.

[2]Prof. Dept of Information Technology,R.V.R. & J.C. College of Engg& Technology,Guntur, India.

E-mail: [1]pcr.bobbepalli@gmail.com,[2]nnmr3654@gmail.com

**ABSTRACT**

Due to online activities and use of resources related to computing, data is being generated at an enormous rate. Distributed systems are the efficient mechanism to access and handle such huge data. One such mechanism is a Hadoop distributed file system (HDFS). An HDFS instant usually contains several nodes, each of which stores a small portion of its data. It creates multiple data blocks and store each of the block redundantly across the pool of servers to enable reliable, extreme rapid computation. HDFS supports common file system operations such as read and write files , create and delete directories. In this paper we are presenting a new paradigm for improving file accessing time in HDFS. It is known that accessing data from cache is much faster than disk access. The cache memory is used to store frequently accessed data & hence process it much more quickly. We have already observed the performance improvement using cache memory in the existing Hadoop environment. In this paper we will prove the performance further improvement by using set associative cache memory. Set associative cache mechanism is for managing the interaction between main memory and cache memory.

***Keywords***— *Hadoop Distributed File System (HDFS), MapReduce, Cache Memory , Set Associative Cache Memory, Average Memory Access Time AMAT, NameNode, DataNode, Second Level Cache, Victim Buffer, Prefetching.*

## 1 INTRODUCTION

Apache Hadoop [1] is a well known project that consists of open source implementation of a distributed file system and MapReduce. One of the significant designed features of the Hadoop system is high throughput which is extremely suitable for handling large scale data analysis and processing problems. HDFS [2] [3] is designed for write-once-readmany access model for files. In HDFS file reading may contain several interactions of connecting NameNode and DataNodes, which considerably decrease the access performance when the system is under a heavy workload. Hadoop [1], MapReduce [5], Dryad [10] and HPCC (High-Performance Computing Cluster) [12] frameworks are Data-intensive and they rely on disk based file systems to meet their exponential storage demands.The system having the namenode acts as the master server it manages the file system namespace. Regulates client's access to files. HDFS supports common file system operations such as read and write files and create and delete directories. The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. Cluster is having number of datanodes. These nodes manage the data storage of their system.As per the instructions from the client datanode will perform operations on the file system.As per the instructions from the namenode blocks will be created , deleted by the datanode. Generally the user data is stored in the files of HDFS. HDFS stores data in HDFS files, each of which consists of a number of blocks (default size is 128MB).In other words, the minimum amount of data that HDFS can read or write is called a block. The default block size is customizable ,i.e we can configure it using the HDFS configuration.

Hadoop distributed file system (HDFS) [6] has the capability to store huge amounts of data. There will be some time factor associated with retrieving or keeping the data in datanode. There are various mechanisms to minimize disk access latencies such as jobs are scheduled on the same node that hosts the associated data, in addition, data is replicated to different nodes in numerous ways to improve throughput and job completion time. When client applications need to write data to HDFS, they perform an initial write to a local file on the client machine, in a temporary file. When the client finishes the write and closes it, or when the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

temporary file's size crosses a block boundary, Hadoop will create a file and assign data blocks to the file. The temporary file's contents are then written to the new HDFS file, block by block. After the first block is written, two other replicas (based on the default replication factor three) are written to two other DataNodes in the cluster, one after the other. The write operation will succeed only if Hadoop successfully places all data block replicas in all the target nodes. Accessing data from cache is faster than accessing data from memory. The cache memory is used to store frequently access data and hence process it much more quickly. By providing a cache system to HDFS, we can avoid unnecessary trips to hard disk to fetch data and thus avoid delay. Accessing the data without cache will take longer time (milliseconds) compared to accessing the data with cache. We have already observed that the performance improvement using cache memory[13]. In this paper we will prove that the performance will be further improved by using set associative cache memory. The paradigm shift is to use set associative cache mechanism to manage the interaction between main memory and cache memory. In this paper we will prove the performance improvement by taking different associativity levels by varying the cache size and the block size i.e, for block size 16Bytes with different cache sizes for each associativity level (2-way , 4-way and 8-way) we will find out the memory access time. Based on the values we will prove that by using the set associativity cache memory we will improve the performance of the memory access , and by increasing the associativity level (2-way to 4-way, 4-way to 8-way) as well we can further improve the performance.

## 2 LITERATURE REVIEW

### 2.1 Hdfs With Cache System

Hadoop Distributed File System organizes its file system differently from the underlying file system such as the Linux ext3 or ext4 file system. HDFS employs a block-based file system, wherein files are broken up into blocks. A file and server in a cluster doesn't have a one-to-one relationship. This means that a file can consists of multiple blocks, all which most likely won't be stored on the same machine. A files blocks are spread throughout the cluster on a random basis. This lets Hadoop support files that are larger than the size of a single disk drive. Since Hadoop is designed to work with massive amounts of data, HDFS block sizes are much larger than those used by a typical relational database. Hadoop uses a minimum block size of 64MB , and its common to use a block size of 128MB or 256MB. The benefits with larger block size is the filesystem metadata will be smaller, large chunks of data can be read sequentially fast streaming reads of data are easier to perform. Based on the client request the data will be copied from main memory to datanode. If we use the HDFSCache system, the frequent access data will be copied to datanode from main memory so that the time required to access the data is less compared to access time without cache. In the existing HDFSCache system the interaction between main memory and cache memory is implemented using direct mapping technique, i.e main memory locations can only be copied into one location in the cache, we can get this configuration by dividing main memory into pages that correspond in size with the cache.Once all the blocks in cache filled with data, then we cannot write down the data to cache block and need to remove the data from the cache to accommodate the new data i.e, swapping the data word from cache to main memory using the replacement algorithm to decide which block in the cache gets replaced by new data which causes more number of swappings, increases the read operation time, which will reduce the performance of the HDFS cluster. HDFSCachesystem is implemented in Linux. HDFS system without cache and with cache We setup a test-bed consisting of five servers running Linux 12.04 64bit OS 15GiB memory to compare the time with HDFS without cache and HDFS with cache[13]. On every computer Hadoop 2.7 (stable version) is installed with the block size 128 MB. Four of these computers are configured to be DataNode servers and the remaining one is configured to be NameNode server. Every DataNode is having Cache Memory. Once all blocks in cache filled with data, swapping is required to copy the new word into cache which causes more number of swappings from main memory to cache. To test the existing environment 10 files have been copied to HDFS having different size. Files size varies from 1000 KB to 10000KMB and the same files have been read with cache and without cache mechanism using scala programming language with spark libraries to interact with Hadoop Distributed File System. Please refer figure 1 for assigning data to variable and reading/printing the data from the variable using scala language with spark APIs without cache. Here we are just showing only top 30 rows. Please refer figure 2 for assigning the data to variable and reading/printing the data from the variable using scala language with spark APIs with cache mechanism. Here we are just showing only top 30 records.

Journal of Theoretical and Applied Information Technology
31st August 2017. Vol.95. No.16

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

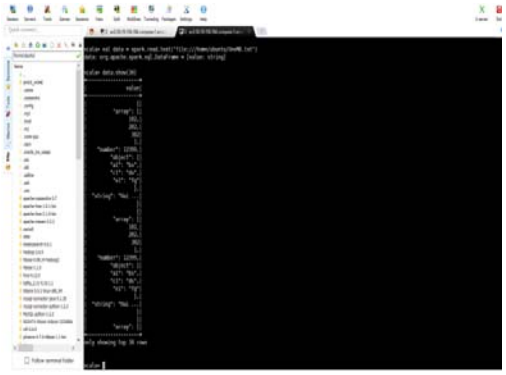


*Figure. 1. Memory access without cache.*

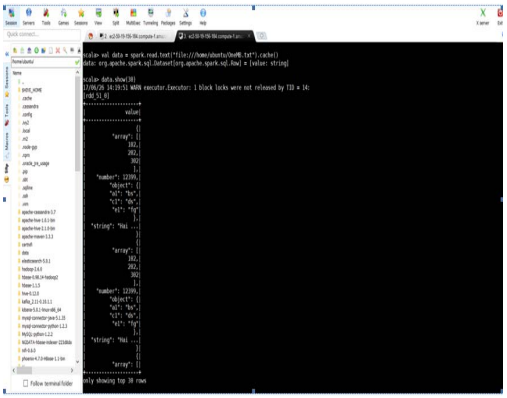


*Figure. 2. Memory access with cache.*

Please refer figure 3 and figure 4 for memory access time with and without cache using the naming convention based on the size of the file i.e; OneMB.txt is having the info about OneMB size text file memory access time, OneMB.txt is showing that 500ms without cache and 97ms with cache where as TwoMB.txt is showing that TwoMB file memory access time is 16ms and 25ms with Cache and without cache respectively. OneMB file is having huge values either with cache(97ms) or without cache(500ms) compared to any other file size from 2MB till 10MB, since that is the first time it is accessing the data. The data between TwoMB and ThreeMB is having little difference so While the file size is getting increased the time is getting down little bit since the data is almost common between OneMB and TwoMB , TwoMB and ThreeMB and so on. Please refer the table 1 for the values, from table it is seen that for the file size of 1000 KB, time required by HDFS is 500 milliseconds and that for HDFSCache is 97 milliseconds. We can observe that the values are getting down from top to bottom while the file size is getting increased , the reason is if we use the distinct file data then the time will goes up as we increase the file size . Here the files are having almost same data with little bit difference. So the time is getting down. In this architecture Direct cache memory technique has been used ,i.e the memory blocks are directly placed to cache memory. If the cache memory is full we need to use algorithms like Least Recently used , Adaptive Replacement Cache and Most recently used replace the cache word with memory word. We can further improve the memory access performance using set associative cache memory technique where we can store set of words in the same line, so that we can allocate most number of memory words at cache as compared to Direct Mapping Cache. Please refer graph1 for the time reduction while using HDFS Cache.

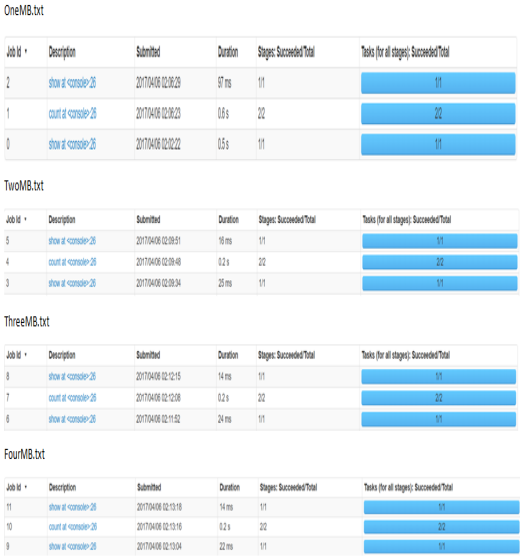
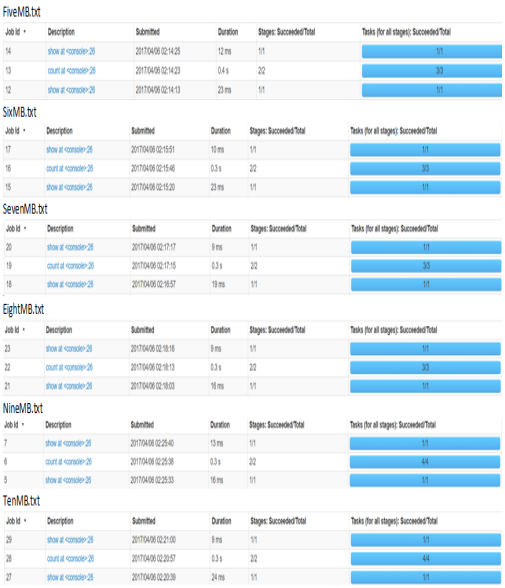



*Figure. 3. Memory Access Time with and without cache.*

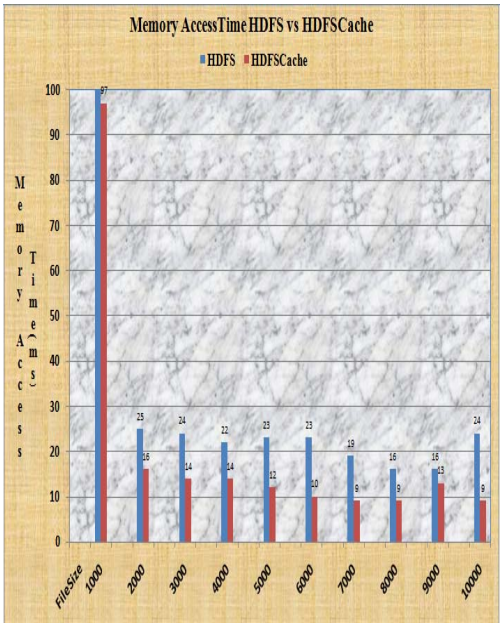


ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

*Figure. 4. Memory Access Time with and without cache.*

*Table 1: Memory AccessTime for HDFS without and with cache*

| FileSize(KBytes) | HDFS(ms) | HDFSCache(ms) |
|---|---|---|
| 1000 | 500 | 97 |
| 2000 | 25 | 16 |
| 3000 | 24 | 14 |
| 4000 | 22 | 14 |
| 5000 | 23 | 12 |
| 6000 | 23 | 10 |
| 7000 | 19 | 9 |
| 8000 | 16 | 9 |
| 9000 | 16 | 13 |
| 10000 | 24 | 9 |




*Graph .1.Memory Access Time for HDFS without and with cache*

### 2.1.1 Namenode

HDFS stores metadata on Namenode, and the application data is stored on the datanodes. The namenode detects failed DataNodes , unavailable replicas and other causes of data corruption. When we startup the Namenode it will do the these three things: The NameNode reads into memory the contents of the fsimage file it has, thus obtaining the HDFS file system state. The NameNode loads the edit log and replays the edit log to update the metadata it loaded into memory in the previous step.The NameNode also updates the fsimage file with the updated HDFS state information[16]. The NameNode starts running with a fresh , empty edits file. The DataNode daemon connects to NameNode and send it block reports that list all data blocks stored by a DataNode.Using Inodes files and directories will be represented on the NameNode. Inodemaintains attributes like permissions modification and access time, namespace and disk space quotas. Blocks on the datanode contains the file data and the replication factor is depends on the configuration parameter used in the HDFS configuration.Namespace in the namenode is having information related to blocks and datanode info information of the file. An HDFS client waiting to read a file first contact the NameNode for the locations of data blocks comprising the file and then reads block content from the DataNode closest to the client. In write operation the client requests the NameNode to nominate a set of DataNodes to write the data in block replicas. Once client receives set of datanodes the data will be written to datanodes in pipeline fashion [7,8].

### 2.1.2 Datanode

The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. The cluster is having number of datanodes. These nodes manage the data storage of their system.As per the client request datanode performs operations on the file .Datanode manages blocks as per the namenode instructions. Namenode will be hvaing handshaking mechanism with datanodes at the startup. During handshake, the namespace ID and software version of DataNode is verified with the NameNode. Based on the success of the match the datanode position will be continued with the namenode. In the failure case of the match the DataNode will automatically shuts down. Namespace ID is assigned to the file system instance when it is formatted.

Journal of Theoretical and Applied Information Technology
31st August 2017. Vol.95. No.16

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

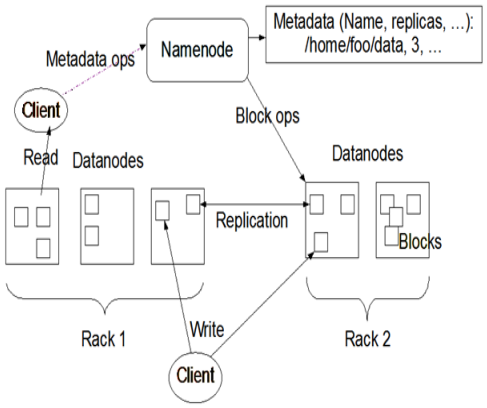



*Figure. 5. HDFS Architecture*

A newly initialized DataNode without any namespace ID can join the cluster and it will receive the cluster's namespace ID. Datanode will be recognized using its unique storage ID.If we restart the datanode with different ip address or port in this case aswell storage ID is usefull to recognize the datanode. When you startup a datanode , it connects to the NameNode and performs a handshake to verify the namespace ID and the Datanode's software version. Following the initial registration with the Namenode, all datanodessend two pieces of information to the Namenode periodic hearbeats that showsthe are alive and a block report that shows block information. Each DataNode send block report to the NameNode to identify the block replicas in its possession. First block report is send during DataNode registration and the subsequent block reports are sent at every hour. This helps the NameNode to keep an up-to-date view of where block replicas are located on the cluster. Each DataNode send heartbeat to NameNode to confirm that it is operating and its block replicas are available. Default heartbeat interval is 3 seconds and if no heartbeat signal is received at NameNode in 10 minutes, the NameNode will mark the DataNode as unavailable. To fulfill this datanode position NameNode schedules creation of new replica of those blocks on another DataNode. Refer figure.5 for HDFS architecture.

### 2.1.3 MapReduce

MapReduce is a programming model for processing and generating large datasets. It provides a series of transformations from a source to a result data set. In a simplest case, the input data is fed to the map function and the resultant temporary data to a reduce function. The developer only defines the data transformations. Hadoop MapReduce job manages the process of how to apply these transformations to the data across the cluster in parallel[15].User specifies a map function and a reduce function. The map function process a key/value pair to generate an intermediate key/value pair. The reduce function merges all intermediate values associated with same intermediate key.In general we can say that a MapReduce job consists of two steps: map and reduce, Map processes the original input file in a parallel fashion and transforms it into an intermediate output. Reduce is the summarization step processes all relevant records together. We need to configure MapReduce environment using mapred-env.sh file[16]. The programs written are inherently parallel and execute on a large cluster of commodity servers. The runtime system take care of all internal details like details of splitting the incoming data into number of parts, programs execution scheduling and handling machine failure.

The MapReduce library group together all intermediate values associated with same intermediate key and pass them to reduce function. Reduce function accept an intermediate key and a set of values for that key and merges together these values to form a smaller set of values just say like zero or one output value is produced per reduce invocation. Iterator function will be used to supply intermediate values to users reduce function. Refer figure 6 for MapReduce Architecture.

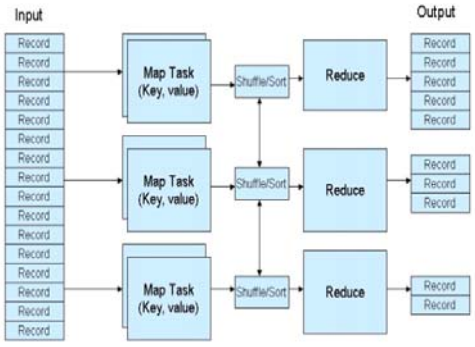



*Figure 6. MapReduce Architecture*

### 2.1.4 Data Distribution in HDFS

HDFS support operation to read, write and delete file as well as to create and delete directories. For reading a file, the HDFS client request the NameNode for the list of DataNodes that host the replicas of the data blocks of the file. Then it directly contacts the DataNode and request the transfer of desired blocks. During writes, the client request the NameNode to choose a list of DataNodes that can host the replicas of the first block of the file. After choosing the DataNodes ,client establishes a pipeline from node to node and

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

sends the data block. After storing the first block, namenode will send next set of blocks once it receives request from datanode for second set. New pipeline will be established between the new set of DataNodes and client sends the further bytes of the file [7].

HDFS provide APIs to retrieve the location of a file block in the cluster. APIs are useful to schedule the task to the node where data are located, thereby improving the read performance. This allows the application to set the replication factor of a file. Three is the defaultreplication factor. For frequently accessed or critical files , setting the replication factor improves their tolerance against faults and increases the read bandwidth. Refer figure 7 for HDFS write operation.

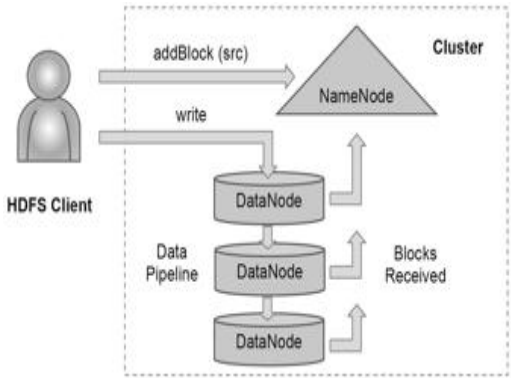



*Figure. 7. HDFS Write Operation*

#### 2.1.4.1 Images and Journal

The inodes and the list of blocks that define the metadata of the name system are called the image. Entire Namespace image is available in RAM. Checkpoint is the location in Namenode where persistent record of the image stored in the local native filesystem of the namenode. The operations or transactions on the HDFS will be recorded by Namonode in write ahead log called journal in its local native filesystem.The jouranl file flushed and synched before the acknowledgement after tranaction initiation by client.NameNode will not change the checkpoint file. During restart checkpoint file will be written when requested by the administrator or by the CheckpointNode.When ever administrator requested for new checkpoint file During startup the NameNode initializes the namespace image from the checkpoint, and then adopt the changes from the journal.

A new checkpoint and an empty journal are written back to the storage directories before the NameNode starts serving clients.For improved durability, redundant copies of the checkpoint and journal are typically stored on multiple independent local volumes and at remote Network File System servers. Single volume failure will be saved by first choice where as failure of the entire node will be protected by second choice. If the NameNode encounters an error writing the journal to one of the storage directories it automatically excludes that directory from the list of storage directories. The NameNode automatically goes down if no storage directory is available. The NameNode is having a multithreaded capability and processes requests simultaneously from multiple clients. Saving a transaction to disk becomes a performance issue since all other threads need to wait until the synchronous flush-and-sync procedure initiated by one of them is complete. In order to improve the performance in this process, the NameNode groups couple of transactions. When one of the NameNode's threads starts a flush-and-sync operation, all the transactions grouped at that time are committed together. Remaining threads only need to check the status that their transactions have been saved and do not need to initiate a flush-and-sync operation.The Namespace image is the file system metadata that describes the organization of application data as directories and files. A persistent record of the image saved to disk is called a checkpoint. For each client-started transaction, the change is saved in the journal, and the journal file is flushed and manages the synch before the change is committed to the HDFS client. The NameNode is a multithreaded system and process request simultaneously from multiple clients. To optimize the saving of transaction to disk, the NameNode batches multiple transactions initiated by different clients. Remaining threads only need to check their transactions have been saved and do not need to initiate a flush-and-sync operation [8].

#### 2.1.4.2 CheckpointNode

Checkpointing is the process that creates a new fsimage and the edit log . Once the edit log reaches a specified threshold or when a certain period of time elapses, the new entries in the edit log are committed to the fsimage file.While the edit log segments are quite small in comparison with the fsimagefile , if you dontreguraly update the fsimage file with the edit log transactions, the edit log could get pretty large itslef, this will delay the start of the Namenode. Checkpointing periodically merges the latest fsimage file with the edit log, creating a brand new up to date fsimage. This helps the Namoenode load its final in memory state directly from the fsimage file instaed of having to reoplay a vast number of files from the edit log. When a NameNode starts up, it merges the fsimage and edits journal to provide an up-to-date view of the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

file system metadata. The checkpoint node and Namenoderequires same storage capacity. So these two runs on different machines.

We can configure checkpoint node and web interface using dfs.namenode.backup.address and dfs.namenode.backup.http-address variables.There are two checkpoint configuration parameters one for maximum delay between two consecutive checkpoints(dfs.namenode.checkpoint.period, set to 1 hour by default), and the number of uncheckpointed transactions on the NameNode which will force an immediate checkpoint, even if the checkpoint period has not been reached(dfs.namenode.checkpoint.txns, set to 1 million by default). Latest checkpoint will be always available at the Namenodes directory. So it is for the namenode to read the checkpoint if necessary. Multiple checkpoint nodes may be specified in the cluster configuration file.The CheckpointNode periodically combines the existing checkpoint and journal to create a new checkpoint and an empty journal. The system can start from the most recent checkpoint if all other persistent copies of the namespace images or journal are unavailable.

#### 2.1.4.3 BackupNode

The Backup node provides the same checkpointing functionality as the Checkpoint node, as well as having an in-memory, latest copy of the file system's namespace that is always synchronized with the active NameNode state[1]. Along with accepting a journal stream of file system edits from the NameNode and persisting this to disk. The same copy of the edits will be saved by Backup node namespace in memory, which is the backup of the namespace. The Backup node does not need to download fsimage and edits files from the active NameNode in order to create a checkpoint, as it is happening with a Checkpoint node or Secondary NameNode, since it is having an up-to-date state of the namespace state in memory. The Backup node checkpoint process is better than checkpoint process as it only needs to save the namespace into the local fsimage file and reset edits.As the Backup node maintains a copy of the namespace in memory, and the RAM requirements are the same as the NameNode. The NameNode supports one Backup node at a time. No Checkpoint nodes may be required if a Backup node is in use.

Usage of multiple Backup nodes concurrently will be supported in the future. The Backup node is configured in the same manner as the Checkpoint node. The command bin/hdfsnamenode -backup is used for starting the BackupNode. Two configuration parameters used in the backup/checkpoint node are dfs.namenode.backup.address, dfs.namenode.backup.http-address configuration variables.Use of a Backup node provides the option of running the NameNode with less responsibilities like no persistent storage, assigning all responsibility for persisting the state of the namespace to the Backup node. Using -importCheckpoint option we can start the NameNode, along with specifying no persistent storage directories dfs.namenode.edits.dir for the NameNode configuration.

It creates periodic checkpoints and in addition it maintains an in-memory, latest image of the file system namespace that is always synchronized with the state of the NameNode. The BackupNodeallows the journal stream of namespace transactions from the active NameNode, and saves them to its own storage directories. The same transactions will be applied to its own namespace image in memory. If the NameNode fails the BackupNode's image in memory and the checkpoint on disk is a record of the latest namespace state. It can perform all operation of the regular NameNode that do not involve modification of the namespace or knowledge of block locations.

#### 2.1.4.4 FileSystem Snapshots

HDFS Snapshots are read-only point-in-time copies of the file system. Subtree of the file system or the entire file system can be taken as Snapshots. Data backup, protection against user errors and disaster recovery are the some common use cases of snapshots. Snapshots can be taken on any directory once the directory has been set as snapshottable. Simultaneously 65,536 snapshots will be accommodated by snapshottable directory. There is no limit on the number of snapshottable directories. Administrators can set any directory to be snapshottable. We need to delete all snapshots inside any snapshottable directory before deleting the snapshottable directory. Nested snapshottable directories are currently not allowed. A directory cannot be set to snapshottable if one of its ancestors/descendants is a snapshottable directory.File system snapshot helps to persistently save the current state of the file system. It helps to rollback in case of failure during upgrade. This helps HDFS to return to the namespace and storage state as they were at the time of snapshot. Each DataNode creates a copy of storage directory and hard links existing block files into it.

When a data block is removed, it removes only the hard link and block modification during append use copy-on-write technique. Thus old block replica remains untouched in their old directory [4]. HDFS implements a single write, multiple-reader model. The client that opens a file for writing is granted a

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

lease for that file and no other client can perform write to the file. The lease is revoked when the file is closed. After writing data to the file, the user application explicitly calls hflush operation. Current packet is pushed to the pipeline and hflush operation waits until all DataNodes in the pipeline acknowledges the successful transmission of the packet. This makes all data written before hflush operation visible to readers [9].

## 3 PROPOSED METHOD

### 3.1 Problem Statement

Improving the cache performance in DataNode: Once all the blocks in cache filled with data, then we cannot write down the data to cache block and need to remove the data from the cache to accommodate the new data i.e, swapping the data word from cache to main memory is required to copy the new word into cache which causes more number of swappings, increases the read operation time, which will reduce the performance of the HDFS cluster.This is the problem in the existing architecture.

### 3.2 Proposal

We can use Set Associative Cache Memory where it will store more number of data words since it is having high number of blocks compared to Directcache mapping technique. 2-way set associative cache memory is having two words of data in each line , 4-way set associative cache memory is having 4 words of data in each line and n-way set associative cache memory is having n words of data in each line. The hit ratio will improve as the set size increases, because more words with the same index but different tags can reside in cache. Compared to DirectMappingCache technique this will take extra logic cost in comparators and cache miss penalty as well. Once the data is available in Cache memory in this process this will take less time compared to DitrectMappingCache. A second-level cache helps improve performance with less database calls, keeping the entity data local scope to the application. The application interacts with normal entity manager without knowing about the cache. The second level cache is responsible for caching objects across sessions. When this is active, objects will first be searched in the cache and if they are not found, a database query will be fired. In a CPU cache, a write buffer can be used to hold data being written from the cache to main memory or to the next cache in the memory hierarchy. A victim buffer is a type of write buffer that stores dirty evicted lines in write-back caches so that they get written back to main memory. Cache prefetching is a technique used by computer processors to boost execution performance by fetching instructions or data from their original storage in slower memory to a faster local memory before it is actually needed. In the proposed implementation we are not considering these hardware optimization techniques.

Refer figure 8 for proposed architecture. Map-Reduce framework will provide cache facility to cache files needed by applications. Hadoop framework will ensure the cached data availability on each and every data nodes DN1,DN2 and DN3(in file system, not in memory) as shown in the figure 8 where your map/reduce tasks are running. Datanodes are having Set associative cache memory configured inside each datanode. You can access the cache file as local file in your Mapper Or Reducer job. Cache memory is having two sets of data along with tag info. If the address is 00000 this is the tag followed by index[11] as shown in the figure 8 this will match for tag using 00 address followed by the index 000 , now the data is 5670. In this set associative cache memory we can store two sets of data for the same tag value like 01 000 and 00 000. Cache system checks whether requested file is available in cache local memory or not. If requested file is available then request is fulfilled by cache. Hence here we can avoid disk access to fetch a file. Else client communicates with DataNodes to check whether requested file is present in their local memory. If file is available then request will be processed. Else if file is not available in cache local memory then file is fetched from disk by using HDFS API. The same will be copied to local cache memory for future reference.

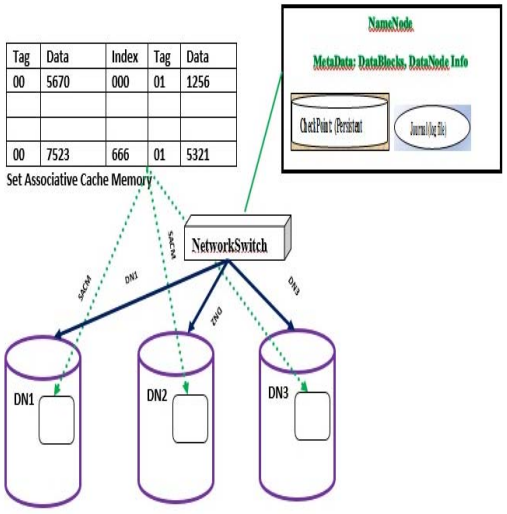



*Figure. 8. Hadoop Distributed File System Architecture with SetAssociativeCacheMemory*

Journal of Theoretical and Applied Information Technology
31st August 2017. Vol.95. No.16

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

## 4 IMPLEMENTATION

Refer figure 9 for the Implementation architecture using Set Associative Cache Memory. NameNode is connected withDatNodes and DataNodes are having internal CacheMemory with Set Associative Cache Memory(SACM) architecture. Whenever client sends request for read or write data file this will be searched in cache memory inside DataNode. If the data is available in the cache memory that will be used for read or write purpose. If the file is not available then that will be accessed from disk using HDFS API. In Set associative cache memory, each word of cache can store two or more words of memory under the same index address. Each index address refers to two data words and their associated tags. The hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache. Compared to DirectMappingCache technique this will take extra logic cost in comparators and cache miss penalty as well. Once the data is available in Cache memory in this process this will take less time compared to Direct Mapping Cache since number of data blocks are less and swapping is high in Direct Mapping Cache.

We can explain our experimental setup with one example. Consider 5 datanodes having local cache memory setup using set associative cache memory technique. Since the space issue here we can consider only two datanodes . The files are scatterred across the datanodes. The Namespace is maintaining datablock and datanode info. When HDFS Cleint sends request for file info this will be verified in datanode local cache. If the data is available this will be fetched from local cache, if not that will be accessed from disk using HDFS API. 2-way set associative cache memory is having two words of data in each line , 4-way set associative cache memory is having 4 words of data in each line and n-way set associative cache memory is having n words of data in each line. Here the given example/figure 9 is showing 2 -way set associative so it is having two sets of data along with tag info. If the address is 00000 this is tag followed by index as shown in the figure 9 this will match for tag using 00 address followed by the index 000 , now the data is 5670. In this set associative cache memory we can store two sets of data for the same tag value like 01 000 and 00 000. In Direct Mapping Cache technique the words available at cache is always less than the words available at set associative cache memory technique. At any instant of time the availability is high so we can reduce the trips to visit memory, i.e, we can have less average memory access time in set associative cache memory. This is how we can validate the performance of set associative cache memory is better than the direct mapping cache.

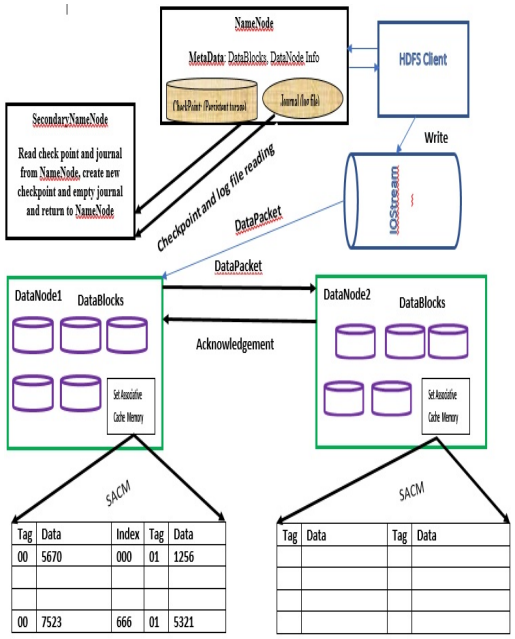



*Figure. 9. DataNode Architecture with SetAssociativeCacheMemory*

## 5 EVALUATION

The results are here for Average Memory Access Time (AMAT) using the cache time analyzer [14] for set associative mapping technique. Refer figure 10 for Cache Time Analyzer [14]. This Cache time analyzer demonstrates Average Memory Access Time analysis for the cache parameters we specify.


*Figure.10. Cache Time Analyzer*

Journal of Theoretical and Applied Information Technology
31st August 2017. Vol.95. No.16

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Write-through cache is the technique where every write operation to the cache is accompanied by a write of the same data to main memory. When this is implemented, then the input/output processor need not consult the cache directory when it reads memory, since the state of main memory is an accurate reflection of the state of the cache as updated by the central processor. Although this scheme simplifies the accesses for the input/output processor, it results in fairly high traffic between central processor and memory, and the high traffic tends to degrade input/output performance.The block is modified in the main memory and not loaded into cache is what we called "No write allocate" and the block is loaded on a write miss, followed by the write-hit action is what we called "Write allocation on Miss".Write-back cache is the technique the central processor updates the cache during a write,but actual updating of the memory is deferred until the line that has been changed is discarded from the cache. At that point, the changed data are written back to main memory. Write-back caching yields somewhat better performance than write-through caching because it reduces the number of write operations to main memory. The information is written only to block in the cache. The cache block once it is modified will be written to main memory. To reduce the frequency of writing back blocks on replacement, a dirty bit will be used. This status bit indicates whether the block is dirty (modified while in the cache) or clean (not modified). If it is clean the block will not be written on a miss. Dirty data means the data in the cache if it is modified but not modified in main memory. Whereas, dirty bit (modify bit) is a cache line condition(status) identifier,its purpose is to indicate whether contents of a particular cache line are different to what is stored in operating memory.If we try to write to an address that is not already contained in the cache this is called a write miss. Percentage of memory accesses that are reads or writes (we are treating them as %reads and %writes), data found in cache is cache hit and data not found in cache is cache miss. If the data is not available in cache, then processor loads the data from memory to cache, which results an extra delay is called miss penalty. Using the following formula[14] we can calculate the Average Memory Access Time(AMAT) with 22% writes, 10% dirty data, 40 Miss Penalty (cycles), 1 Hit Time(cycles), 6 Memory hit (cycles)and block size is 16Bytes. Clocks MemWrite is the write time in clock cycles for a single write.

ReadHitContribution :%Reads * Hit_rate * HitTime

ReadMissContribution:%Reads*MissRate*((MissPenalty+HitTime)+(%Dirty*MissPenalty))

Write Hit Contribution: %Writes * HitRate * HitTime

WriteMissContribution: %Writes*MissRate* MissPenalty. Average Memory access Time has been collected for different input factors like Write Back policy and Write Through Policy using No-write Allocate on miss and Allocate on Miss methods on each policy with different units on CacheSize, Associativity and BlockSize[14].

Refer figure 11 for Write Back-No Write Allocate cache write policy. Table 2 shows the results along with the associativity for the block size 16. Average Memory Access Time (AMAT) is decreasing once we increase the associativity from 2 till 8. For the cache size 2 , the top row shows that the time is decreasing once we increase the associativity level i.e for 1-way associativity is taking 6.14ms whereas 2-Way associativity is taking 4.9897 ms , 4-Way is taking 4.35 ms and 8-Way is taking 3.83ms.The time is going down while we are increasing the associativity level. We can observe the same in table as well as in graph. 2

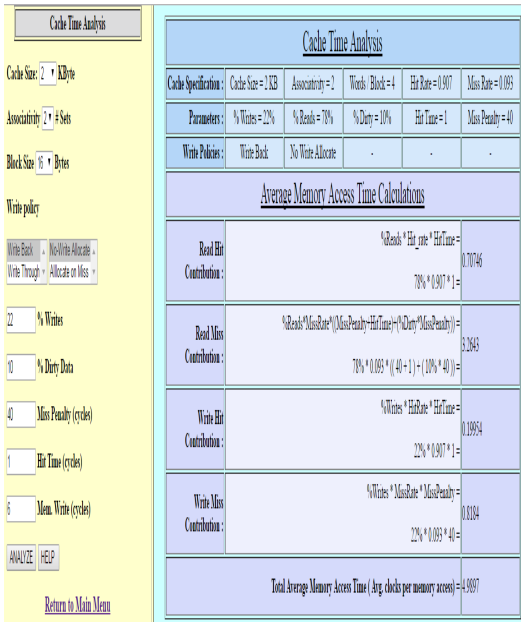


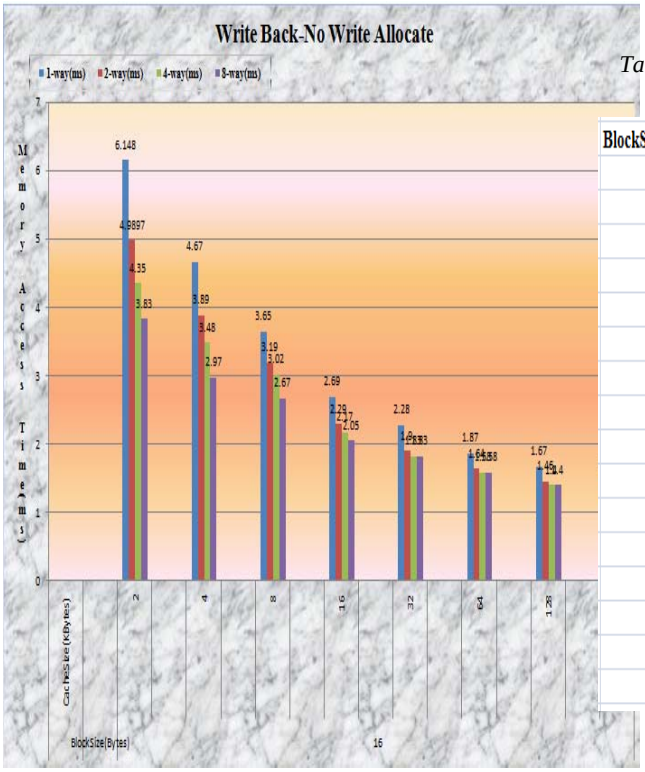

*Figure. 11. Write Back -No Write Allocate*

Journal of Theoretical and Applied Information Technology
31st August 2017. Vol.95. No.16

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

*Table 2: AMAT with different associativity levels: Write Back -No Write Allocate*

| BlockSize(Bytes) | CacheSize(KBytes) | 1-way(ms) | 2-way(ms) | 4-way(ms) | 8-way(ms) |
|---|---|---|---|---|---|
| 16 | 2 | 6.148 | 4.9897 | 4.35 | 3.83 |
| 16 | 4 | 4.67 | 3.89 | 3.48 | 2.97 |
| 16 | 8 | 3.65 | 3.19 | 3.02 | 2.67 |
| 16 | 16 | 2.69 | 2.29 | 2.17 | 2.05 |
| 16 | 32 | 2.28 | 1.9 | 1.83 | 1.83 |
| 16 | 64 | 1.87 | 1.64 | 1.58 | 1.58 |
| 16 | 128 | 1.67 | 1.46 | 1.4 | 1.4 |
| 16 | 256 | 1.46 | 1.33 | 1.26 | 1.26 |




*Graph.2. Write Back-No Write Allocate with different Associativity levels:Block Size 16Bytes*

Refer figure 12 for Write Back - Allocate on Miss cache write policy. Table 3 shows the results along with the associativity for the block size 16. Average Memory Access Time (AMAT) is decreasing once we increase the associativity from 2 till 8. For the cache size 2 , the top row shows that the time is decreasing once we increase the associativity level . We can observe the same in table3 as well as in graph. 3.

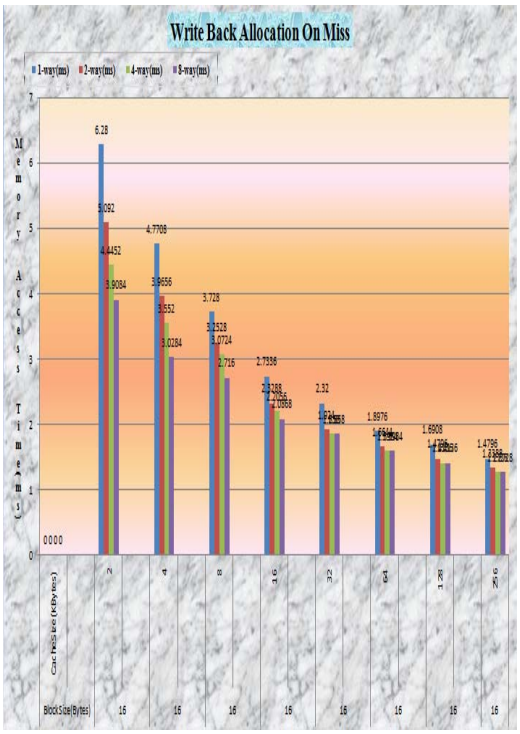

*Figure. 12. Write Back - Allocate On Miss*

*Table 3. AMAT with different associativity levels: Write Back - AllocateOn Miss*

| BlockSize(Bytes) | CacheSize(KBytes) | 1-way(ms) | 2-way(ms) | 4-way(ms) | 8-way(ms) |
|---|---|---|---|---|---|
| 16 | 2 | 6.28 | 5.092 | 4.4452 | 3.9084 |
| 16 | 4 | 4.7708 | 3.9656 | 3.552 | 3.0284 |
| 16 | 8 | 3.728 | 3.2528 | 3.0724 | 2.716 |
| 16 | 16 | 2.7336 | 2.3288 | 2.2056 | 2.0868 |
| 16 | 32 | 2.32 | 1.924 | 1.858 | 1.858 |
| 16 | 64 | 1.8976 | 1.6644 | 1.5984 | 1.5984 |
| 16 | 128 | 1.6908 | 1.4796 | 1.4136 | 1.4136 |
| 16 | 256 | 1.4796 | 1.3388 | 1.2728 | 1.2728 |

Journal of Theoretical and Applied Information Technology
31st August 2017. Vol.95. No.16

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195




*Graph.3. Write Back- Allocation On Miss with different Associativity levels:Block Size 16Bytes*

Refer figure 13 for Write Through - No Write Allocate cache write policy. Table 4 shows the results along with the associativity for the block size 16. Average Memory Access Time (AMAT) is decreasing once we increase the associativity from 2 till 8. For the cache size 2, it shows that the time is decreasing once we increase the associativity level. We can observe the same in table 4 as well as in graph. 4.

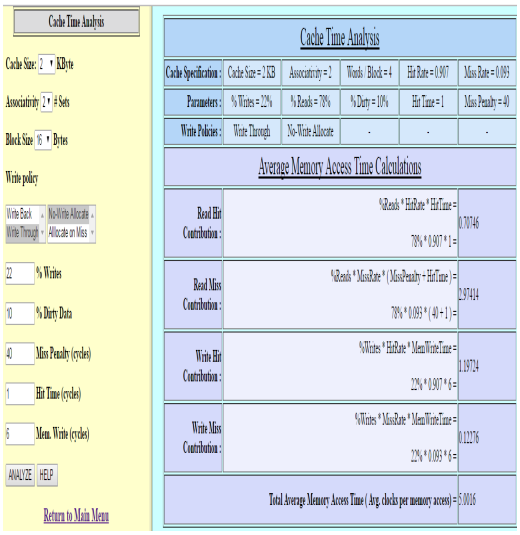
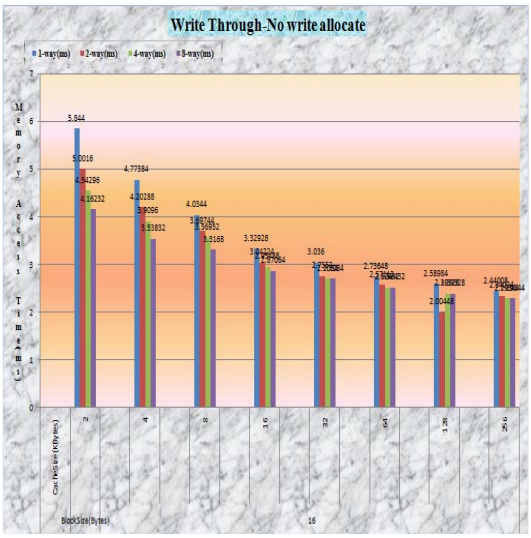



*Figure. 13. Write Through - No Write Allocate*

Table 4. *AMAT with different associativity levels: Write Through -No Write Allocate*

| BlockSize(Bytes) | CacheSize(KBytes) | 1-way(ms) | 2-way(ms) | 4-way(ms) | 8-way(ms) |
|---|---|---|---|---|---|
| 16 | 2 | 5.844 | 5.0016 | 4.54296 | 4.16232 |
| 16 | 4 | 4.77384 | 4.20288 | 3.9096 | 3.53832 |
| 16 | 8 | 4.0344 | 3.69744 | 3.56952 | 3.3168 |
| 16 | 16 | 3.32928 | 3.04224 | 2.95488 | 2.87064 |
| 16 | 32 | 3.036 | 2.7552 | 2.7084 | 2.7084 |
| 16 | 64 | 2.73648 | 2.57112 | 2.52432 | 2.52432 |
| 16 | 128 | 2.58984 | 2.00448 | 2.39328 | 2.39328 |
| 16 | 256 | 2.44008 | 2.34024 | 2.29344 | 2.29344 |




*Graph.4. Write Through- No Write Allocate with different Associativity levels: Block Size 16Bytes*

Refer figure 14 for Write Through - AllocateOn Miss cache write policy. Table 5 shows the results along with the associativity for the block size 16. Average Memory Access Time (AMAT) is decreasing once we increase the associativity from 2 till 8. For the cache size 2, the top row shows that the time is decreasing once we increase the associativity level. We can observe the same in table 5 as well as in graph. 5

Journal of Theoretical and Applied Information Technology
31st August 2017. Vol.95. No.16

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

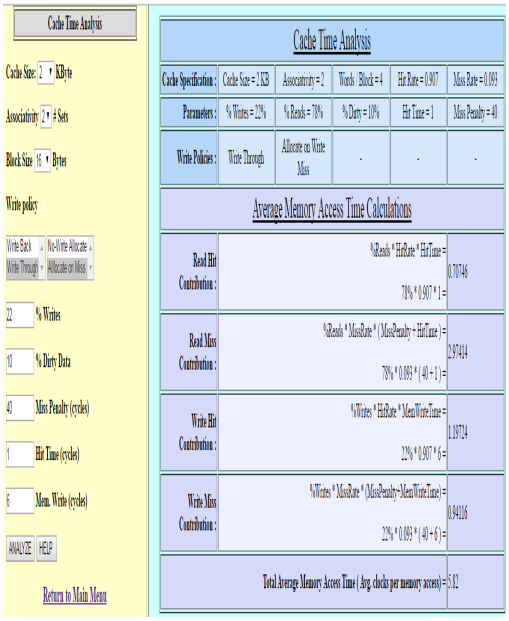
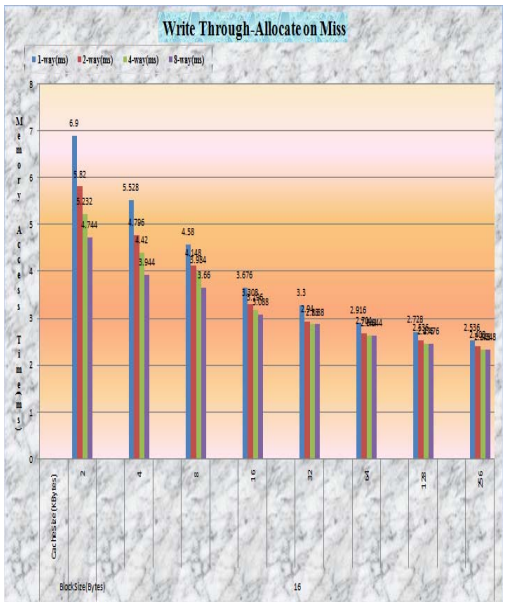



*Figure 14. Write Through - Allocate On Miss*

*Table 5: AMAT with different associativity levels: Write Through - AllocateOn Miss*

| BlockSize(Bytes) | CacheSize(KBytes) | 1-way(ms) | 2-way(ms) | 4-way(ms) | 8-way(ms) |
|---|---|---|---|---|---|
| 16 | 2 | 6.9 | 5.82 | 5.232 | 4.744 |
| 16 | 4 | 5.528 | 4.796 | 4.42 | 3.944 |
| 16 | 8 | 4.58 | 4.148 | 3.984 | 3.66 |
| 16 | 16 | 3.676 | 3.308 | 3.196 | 3.088 |
| 16 | 32 | 3.3 | 2.94 | 2.88 | 2.88 |
| 16 | 64 | 2.916 | 2.704 | 2.644 | 2.644 |
| 16 | 128 | 2.728 | 2.536 | 2.476 | 2.476 |
| 16 | 256 | 2.536 | 2.408 | 2.348 | 2.348 |

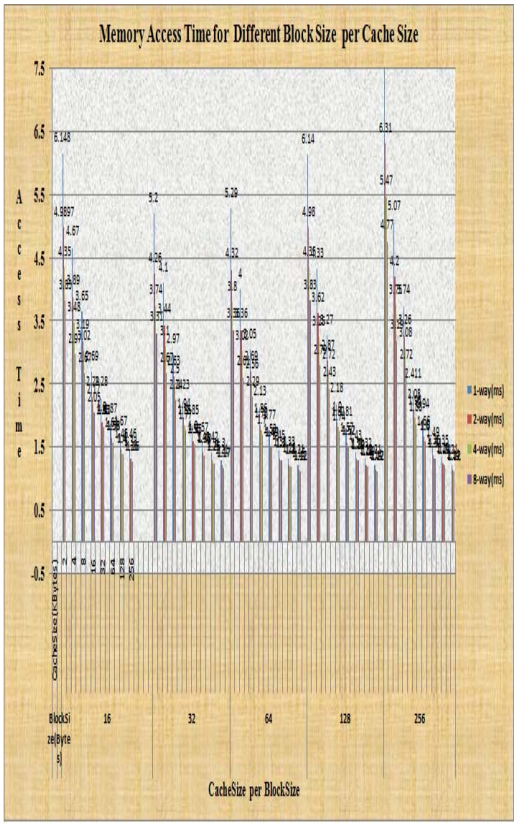



*Graph.5. Write Through- Allocate On Miss with different Associativity levels:Block Size 16Bytes*

Average Memory AaccessTime (AMAT) varies for different Associativity level and it is decreasing as shown the graph.6. As shown in the table 2,3,4, and 5for each row the time is getting decreased as we move on from lower to higher associativity level, table 2 shows that for block size 16 and cache size 16 the average memory access timings are 2.69, 2,29, 2.17 and 2.05 for associativity levels 1,2, 4 and 8 respectively using write back cache and no write allocate. As in table 3 for block size 16 and cache size 8 the average memory access timings are 3.728, 3.2528, 3.0724 and 2.716 for associativity levels 1, 2, 4 and 8 respectively using write back cache and allocate on Miss policy.Table 4 shows that for block size 16 and cache size 4 the average memory access timings are 4.77384, 4.20288,3.9096 and 3.53832 for associativity levels 1,2,4 and 8 respectively using write through cache and no allocate write policy. Table 5 shows that for block size 16Bytes and cache size 4KBytes the average memory access timings are 6.9, 5.82, 5.232, and 4.744 for the associativity levels 1,2,4 and 8 respectively using write through cache and allocate on miss policy. The average access time is even getting down if we increase the associativity level, i.e, average access time with associativity level 8 is lower than associativity level 2 or 4. As shown in Table 5the average access time is 3.088 milliseconds for associativity level 8, where as it is 3.66, 3.944 and 4.744 for associativity levels 4,2 and 1 respectively. Graph 6 is showing that AMAT is going down while increasing associativity level for each block size 16, 32, 64, 128 and 256. Using

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

the stats which we have had in the tables and graphs, we can conclude that we can decrease the average memory access time which includes read and write operations by using the set associativity cache levels as compared to Direct Mapping cache technique. Existing architecture in the Hadoop is having Direct Mapping cache Technique, i.e what we have showed in table 2, 3, 4 and 5with associativity level 1. As per the analysis based on the stats what we had the average memory access time for Direct Mapping cache technique (associativity level 1) is always higher than any associativity level average memory access time. So by implementing the set associative cache memory mapping technique in the Hadoop architecture, we can reduce the memory access time, i.e, we can accommodate number of data words at datanode itself so that map reduce client can get the data word from datanode itslef, instead of getting from memory, so reducing the average memory access time.




*Graph.6. AMAT with different Associativity levels*

Write Back - No Write Allocate

Block Size: 16, 32, 64, 128, 256Bytes

## 6 CONCLUSION

The cache memory is used to store frequently accessed data & hence process it much more quickly.We have already observed the performance improvement using cache memory in the existing Hadoop environment[13]. In this paper we have proved that the performance further improvement by analysis results using Set Associative Cache Memory. Set associative cache mechanism is for managing the interaction between main memory and cache memory.Based on the analysis of values for same cache and block size with different associativity levels we can say that there is improvement in average memory access time by using the set associative cache memory technique for mapping cache memory to main memory. The memory access time is even lower than the existing HDFS Cache architecture memory access time. So we can conclude that HDFS with cache is better than without cache[13], and HDFS Cache using associativity levels is even better than HDFS with Cache. The second level cache is responsible for caching objects across sessions. A victim buffer is a type of write buffer that stores dirty evicted lines in write-back caches so that they get written back to main memory. Cache prefetching is a technique used by computer processors to boost execution performance by fetching instructions or data from their original storage in slower memory to a faster local memory before it is actually needed. Hardware based prefetching is typically accomplished by having a dedicated hardware mechanism in the processor that watches the stream of instructions or data being requested by the executing program, identifies the next few elements that the program might need based on this stream and prefetches into the processor's cache.In the proposed architecture, these techniques have not been considered. The future work includesreducing the average memory access time even further by considering hardware optimization techniques such as second level cache, victim buffer and prefetching while implementing the set associative cache memory architecture in Hadoop Distributed File System.

## REFERENCES

[1] Apache Hadoop. Available at Hadoop Apache.

[2] Apache Hadoop Distributed File System. Available at Hadoop Distributed File System Apache.

[3] Scalability of Hadoop Distributed File System.

[4] George Porter. Decoupling storage and computation in Hadoop with SuperDataNodes,

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ACM SIGOPS Operating System Review, 44, 2010.

[5] Hadoop Distributed File System with Cache technology by Archana Kakade and Dr. Suhas Raut, Industrial Science Vol.1,Issue.6/Aug. 2014 ISSN : 2347-5420

[6] J. Dean and S. Ghemawat (2004), "Mapreduce: Simplified Data Processing on Large Clusters". In Proceeding of the 6th Conference on Symposium on operating Systems Design and Implementation (OSDI'04), Berkeley, CA, USA, 2004, pp.137-150.

[7] Shafer J, Rixner S, Cox AL. The Hadoop Distributed Filesystem: Balancing Portability and Performance, in Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2010), White Plains, NY, 2010.

[8] Feng Wang et al. Hadoop High Availability through Metadata Replication, IBM China Research Laboratory, ACM, 2009.

[9] Derek Tankel. Scalability of Hadoop Distributed File System, Yahoo developer work, 2010.

[10] "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM" Department of ComputerScience Stanford University.

[11] Computer System Architecture, Third Edition, M.Morris Mano

[12] J. Shafer and S Rixner (2010), "The Hadoop distributed file system: balancing portability and performance", In 2010 IEEE International Symposium on Performance Analysis of System and Software (ISPASS2010), White Plains, NY, March 2010. Pp.122-133.

[13] Ms. Archana Kakade, Dr. Suhas Raut, "HDFS with cache system – a paradigm for performance improvement"

[14] William Stallings(2013), "Computer Organization and Architecture: Desigining for performance", Ninth Edition .

[15] Garry Turkington(2013), HadoopBeginner's Guide, Learn how to crunch big data to extract meaning from the data avalanche

[16] SAM R. ALAPATI , Expert Hadoop Administration, Managing, Tuning and Securing Spark, YARN and HDFS, Addison wesley data & analytics series, 2017.