Mitigating Hotspots in Distributed File Systems by Using System-State Metrics

Prashant Raghav
University of Waterloo
praghav@uwaterloo.ca

Harshdeep Singh Saluja
University of Waterloo
hssaluja@uwaterloo.ca

Jayanth Kottapalli
University of Waterloo
jkottapa@uwaterloo.ca

Rahul Rawat
University of Waterloo
r3rawat@uwaterloo.ca

***Abstract*—A Distributed File System (DFS) is scalable, runs on inexpensive hardware and most importantly, uses file replication to achieve fault tolerance. Multiple instances of a file mean that a data access request must be directed to one of the storage nodes by a central server. In DFS, hotspots arise when a single storage node is bogged down by requests from many clients which leads to low throughput and high latency for data access. In this work we propose a mechanism to avoid the formation of hotspots by making the central server more aware about the system state of each storage node. We piggyback system state metrics onto heartbeat messages to achieve this. We also benchmark our implementation and compare its performance with latest version of HDFS, a popular open source distributed file system.**

Keywords*—*Hadoop Distributed File-System; HDFS; Namenode; Datanode; Hotspots; System-state Metrics; Hadoop Benchmarking;

I. Introduction

The phrase digital universe is used to refer to all the digital data that has been created, consumed, replicated and stored. This digital universe includes images and videos created using various devices, data being offered by streaming services such as Youtube, Netflix etc., data related to financial institutions such as banks, footage from security cameras, vast amounts of scientific data collected from experiments and data from a multitude of other sources. A recently concluded study [1] by EMC and IDC points out that the digital universe is growing at a rate of 40% a year and is expanding to include things like smart devices (e.g. the NEST thermostat) for homes. Currently, the digital universe is comprised of 4.4 trillion GBs of data and the 40% growth rate suggests that this number will reach 44 trillion GBs by 2020 [1]. This rapid growth of unstructured data presents us with both opportunities and challenges. Innovation at a large number of organizations depends upon the capability to collect and analyze the vast amount of digital data collected. The analysis of the collected data forms the innermost loop of the product improvement at these organizations. The search logs which are collected for several users can be analyzed to exploit identifiable trends for use by designers of the search engine ranking algorithms. Amazon.com, Inc. analyzes the click stream and the historical data of a user to recommend products. Online advertisements that are generated by analyzing user search logs and are targeted at particular groups of users, generated close to $113 billion in 2013 [13].

A conventional file system like the widely used NTFS is incapable of storing data at a large scale and there is a need for scalable distributed file systems in order to meet the storage requirements of large distributed data-intensive applications, both in the industry (e.g. web data analysis, click stream analysis, network- monitoring log analysis) and in the sciences (e.g., DNA sequence analysis, analysis of data produced by high resolution scans, physics experiments, massive scale simulations and new high throughput sensors and devices). In addition, as enterprises are moving towards using distributed systems to handle their business activities, the number of clients has only continued to increase [1].

In the past few years, this problem has gained a lot of attention and multiple distributed file systems have been proposed, in particular, Google File System (GFS) [2], Hadoop Distributed File System (HDFS) [4], GPFS [3], Ceph [6], GlusterFS [10], GFarm [7] among others. One of the many ways this is accomplished is by replicating the data across the network in multiple locations rather than using a dedicated storage cluster and high bandwidth network between compute nodes and the storage nodes to improve storage access performance. Distributed File Systems such as GFS and HDFS use this approach of replicating files on different storage nodes and are very efficient in storing large amounts of data. A single file is broken down into blocks of equal size and each block is stored on multiple servers to ensure that a catastrophic failure of a single server does not render an entire file unavailable. When a data access request is made by a client, the request is directed to a central server which then returns the location of the server which holds the requested file block. Hotspots arise when a single storage server in a distributed file system is bogged down by data access requests by many clients.

In this work, we present a way to avoid the formation of hotspots by altering the decision making process of the central server, by which it decides where to direct the data access request of a client, in case a storage node is under load. HDFS, a widely used DFS, uses the topological distance between a client and the storage server as the only metric while routing data access requests to appropriate servers. We propose the system state of a storage node must also be taken into consideration in this decision making process. We chose the latest stable version of Apache Hadoop, 2.4.1, to make the necessary changes for the implementation of the proposed mechanism for preventing the formation of hotspots. The rest of the document is structured as follows: Section 2 provides the necessary background information which includes an overview of why we need a DFS as well as how hotspots can form in a distributed file system. Section 3 contains the technical details with regards to how the current HDFS is implemented and what modifications we have made. Section 4 contains the

performance evaluation of our implementation which includes benchmarking and experimental results. Section 5 provides our conclusion statements along with the technical challenges we faced as well as future work.

II. BACKGROUND

A file system is an essential component for an operating system to organize data in a particular manner in its persistent storage. There are many different types of file systems that are used by operating systems. The most well known file systems are that of NTFS and FAT used by various versions of Windows operating systems, and the inode file system used by Unix-based operating systems of Mac OS and Linux. To satisfy the need for being able to share data across a network, a distributed file system called Network File System (NFS) was proposed to share data over the network. NFS provides a transparency model and allows remote access to a single logical volume (a logical drive) on a single machine that can make a portion of the local file system visible to other users in the network [8]. By using NFS, users can mount this logical volume onto their own local file system and interact with it as it were a part of their local drive. However, NFS is very limited in its performance as a distributed file system. Although the files appear to be existing locally, they are still physically located on a remote machine. When a user wants to perform a read or write operation on a particular file that exists on a NFS volume, that user has to copy the entire file on their local machines before they can operate on it. In addition, as information can only be stored on one machine, this does not have any guarantee of reliability or availability. If the machine the NFS volume is physically located has any sort of failure, the data might no longer be available to other users. This is a result of NFS not providing any replication of the files on other machines.

There is an obvious need for a distributed file system that has a significantly greater performance while being reliable and highly available regardless of catastrophic failures occurring. Subsequently, numerous distributed file systems have been proposed to meet these requirements. GFS is one of the more popular file systems that has been proposed by Google Inc. to meet the distributed computing requirement. HDFS is an Apache Software Foundations open-sourced distributed file system that is based on GFS [15] and is currently one of the most extensively used open-sourced distributed file system where various data intensive applications are allowed to run on commodity hardware. IBMs GPFS is a proprietary software for computer clusters that uses RAID controllers which are not supported by HDFS, to prevent data loss [3].

In a distributed file system that is based on inexpensive commodity hardware, component failures are a norm rather than an exception [2]. Consequently, failure of a single node should not cause unavailability of an entire file. This problem is solved by replicating the files across multiple nodes in the distributed environment. Apache Hadoop uses a Hadoop Distributed File System (HDFS) [4] which is a block structured file system where a single file is divided into block of equal fixed sizes. In addition, not all the blocks of a file are necessarily stored on a single node. This implies that access to a single file will require coordination between multiple data nodes.

When a client makes a request for data using HDFS, this request is for specific blocks and considering a block exists in multiple nodes, the data request can be serviced by the closest data node in the network topology. The selection of the datanode that services the data request is determined by a master node (namenode in HDFS) that maintains metadata regarding the file blocks, datanodes and replication factor of a particular file, etc. Client data access pattern affects the performance of a distributed storage system. If many clients are requesting access to the same block of file and in addition, are being serviced by the same data node in the network topology, this would result in the creation of a hotspot. The issue with regards to hotspots is also observed when the replica counts are relatively low.

Hotspots are undesirable in a distributed file system as it limits the performance and throughput of the system where the overloaded node will determine the performance and become the lowest common denominator of the network. In addition, GFS mentions the problem occurring in practice during experiments with production traffic [2]. Additionally, the Hadoop community has faced the issue with hotspots when many clients try to read from the same block simultaneously which results in an asymmetric read load distribution [16].

Suggested approaches for resolution of hotspots in GFS and Hadoop are not up to the mark. GFS considers increasing the replication factor for hotfiles and/or using staggering start times in case the file being accessed is an executable script. On the other hand, Hadoop community came up with the solution of using randomized replicas selection which may result in high latency. Both of the proposed methodologies are not long term solutions. We believe that a long term solution is possible by making the namenode more *datanode-state* aware.

As mentioned previously, we chose the file system employed by Apache Hadoop, HDFS, to implement our proposed solution and document the benchmark results. The rationale behind this selection is that Apache Hadoop project is an open-source software for distributed computing and the implementation details of the Hadoop Distributed File System are well documented. Our contributions in this paper are as follows: We propose a novel technique to augment the current process that is employed by the namenode to route data requests to appropriate datanodes. System load of each data node is communicated to the namenode and this value affects the routing decision made by the namenode while servicing a data request. We then provide details regarding what changes we have made in the HDFS source code to make the system more *datanode-state* aware. We also provide the metric weightage which the namenode uses to take the decision while returning the list of sorted replica locations of a requested block. Once we provide the implementation details, we then present the details regarding our benchmarking strategy, testbed and experimental results.

III. TECHNICAL DETAILS

A. HDFS Basics

The Hadoop Distributed File System (HDFS) is designed to store large amounts of data and provide high-throughput access to this data. There are two main components in a HDFS cluster, a master server called the namenode which regulates file access

to the client and a cluster of replica nodes called datanodes that service the clients requests. HDFS is a block structured file system such that files are broken into blocks of fixed size, 128MB by default (as of v2.4.1) but this size is configurable. These blocks of data are stored on a cluster of datanodes. A single file, made up of a number of blocks, can be stored across a number of datanodes. To prevent the loss of an entire file due to loss of a single file block, HDFS replicates each block across a number of machines. This replication factor is 3 by default and can be adjusted. HDFS is predominantly a write-once and read-many model for data access.

In such an implementation, the metadata of the file system can be changed by a large number of users concurrently. This includes executing file system commands and other changes to the filenames, directory names, among others. To ensure that this information remains synchronized, the file system metadata is stored and maintained by the namenode in a file called FSImage. The metadata also includes the replication information and the directory structure of Hadoop. The namenode controls the entire HDFS and handles client requests using the metadata. In addition, the namenode also monitors datanode failures and manages dynamic insertion and replication of blocks. The main task of the datanode is to handle the storage replication. In particular, it writes data to the local file system in the form of blocks which are assigned a unique ID by the namenode. The datanode frequently sends its heartbeat messages and the block reports of the files in the local file system to the namenode. To prevent data loss in case of catastrophic failures, the block is stored at a number of other datanodes which is specified by the replication factor.

B. High-level working of HDFS:

There are three main communication interfaces in HDFS. There is a communication interface between the client and the namenode, the client and a datanode and lastly, between a datanode and the namenode. In order to access a file, a client contacts the namenode using the standard Java API available in HDFS. Using the APIs available in *ClientProtocol* interface, the client establishes a connection to the namenode. If the client wants to write data to the HDFS, it has to coordinate with the namenode before it can write. Specifically, it has to get permission from the namenode for the creation of a new file in the Hadoop namespace. The namenode gives the client a lease to write a particular file in the namespace on a particular datanode. Once the client gets permission from the namenode, the client stores the data in a temporary file on the local filesystem until the file size has reached one block size (default 128MB). The client then communicates with the namenode to register the file block. The namenode will then commence the creation of the file onto HDFS. If a file that the client wants to write has more data than 128MB, then the current block is flushed from the temporary file in the local file system onto the specified datanode. This creation of a 128MB sized block and flushing on to the datanode is repeated until the entire file has been written. Once the client completes writing to the datanode, it invokes *complete()* method as specified in in *ClientProtocol* and finally the lease on the file is terminated.

Similarly, when the client wants to read data from the datanode, it propagates its request for the file to the namenode. When the namenode receives the request from the client, it returns a sorted list of blocks that comprise the requested file along with the information of datanodes holding these blocks. The client then communicates with a datanode to receive data using the data transfer protocol as defined in the *DataTransferProtocol.java*. The namenode returns a sorted list of datanodes for the client read request by initially identifying which datanodes have the requested data. Once it determines which datanodes have the file, it does the sorting of the datanodes based on the their topological distance from the client. This distance can fall under one of the three categories,

- the datanode exists on the same machine as the client (weight 0)
- the datanode exists on the same physical rack as the client (weight 1), or
- the datanode exists on a different physical rack than the client (weight 2)

In HDFS local datanodes are preferred over the remote ones (nodes on different racks). Accordingly the sorting is done in an ascending order of weights. Datanodes that have same weightage assigned to them are shuffled randomly in an attempt to reduce network traffic and improve performance. We believe this approach can be improved by taking into account datanode-state along with their topological distance from the client.

The third interaction is between the datanode and the namenode. The communication is initiated by the datanode using four specific methods viz. *registerDatanode()*, *sendHeartbeat()*, *blockReport()* and *cacheReport()* in the *DatanodeProtocol* interface. By invoking the *registerDatanode()* method, the datanode registers itself to the namenode, which in turn returns a unique ID generated for the datanode. By invoking the *sendHeartbeat()* method the datanode sends a heartbeat, which contains information about block and cache state of the datanode, to the namenode every 3 seconds to mark its availability. Once the namenode receives this information, it returns a list of block oriented commands for the datanode to execute as part of the heartbeat response. The datanode also invokes the methods of *blockReport()* and *cacheReport()* to let the namenode of those details.

The workflow of how a namenode manages and uses the metadata information received via above message calls can be explained as follows. The namenode delegates work to three entities (classes), a *BlockManager*, a *DatanodeManager* and a *HeartbeatManager*. Each of these entities are separated into classes which handle different operations for the namenode. The *HeartbeatManager* is a very crucial class as it handles all the corresponding messages sent to the namenode by the datanodes in HDFS. In order to achieve storage reliability, HDFS must overcome failures which requires that these failures be detected as quickly as possible. Each datanode, sends periodic heartbeat messages; timed at 3 seconds, to the namenode, which are handled by the *HeartbeatManager*. Absence of these heartbeat messages from a datanode for a certain period of time is construed as failure of that particular datanode. The namenode using this information then flags that particular datanode as dead to ensure that it will be excluded in the list of locations when a client makes a file request and subsequently making sure that no further data access requests are not sent

to this datanode by a client. This heartbeat message consists of the following fields in the original hadoop code:

- registration- datanode registration information
- capacity- total storage capacity available at the datanode
- dfsUsed disk storage used by HDFS
- remaining remaining storage available for HDFS
- blockPoolUsed- storage used by the block pool
- xmitsInProgress- number of transfers from this datanode to others
- xceiverCount- number of active transceiver threads
- failedVolumes- number of failed volumes
- cacheCapacity- total cache capacity available at the datanode
- cacheUsed- amount of cache used

The namenode uses some of these fields to maintain the entire HDFS status. Namely, the *capacityTotal, capacityUsed, capacityRemaining, blockPoolUsed, xceiverCount, cacheCapactiy, cacheUsed and expiredHeartbeats*. When a heartbeat message is received by a namenode, these values provided in the fields of the heartbeat message are updated accordingly in the statistics that are maintained for the entire HDFS in a data structure called *Stats*. Given a datanode file descriptor, this *Stats* data structure provides two methods which are used to subtract and add statistics for the entire HDFS. When a datanode sends a heartbeat, the old statistics values for this particular datanode are subtracted first and then the new values are added.

C. Our implementation

As far as the current implementation of HDFS is concerned the only metric that is used for deciding which data node should have the highest priority for servicing a data request is the topological distance between the client and the data node. As aforementioned, this distance is classified into three broad categories: data node resides on the same machine as the client that made the data access request, the data node resides on the same node as the client and the data node resides on a different rack. As argued earlier, even though this distance is important, it is not the only metric that should be taken into account.

Our hypothesis is that besides the topological distance, metrics that convey the system state of a datanode are also important. The three new metrics that we have identified for this purpose are CPU utilization, memory utilization and I/O load. These metrics were finalized after performing extensive set of experiments to identify the system parameters which are indicative of hotspot conditions. To each datanode we assign a score calculated on the basis of the information received as a part of the heartbeat message from that datanode. The maximum score that can be associated with a datanode is 10. *Table 1* shows how much each of the above metrics contribute to this score as well as the contribution of the topological distance.

TABLE I. METRIC WEIGHTAGE

Metric	Weightage
Distance	35%
Idle CPU%age	10%
Free Memory %age	20%
I/O Utilization	35%

The rationale behind giving these metrics the above listed weightages are based on certain well known factors and conducting preliminary research by performing multiple runs using different weightages. The metrics of the distance and I/O utilization are given the most weightage out of all the metrics because they have a significant influence on the performance of the HDFS framework. These metrics can determine the performance of the datanodes due to both physical limitations as well as hardware limitations. The distance between where the datanode and the client are located has a significant effect on the performance of a distributed file system. Subsequently, we chose to assign a higher percentage to distance metric relative to the other metrics we considered.

Similarly, the I/O is considered on the same basis of it having substantial effect on the performance. Since HDFS works under the premise that the Hadoop cluster is built off of commodity hardware, the best (magnetic) hard disks have a well established limit of how well they can perform read/writes. The I/O capabilities of a datanode are only limited by how well information is retrieved from the persistent storage, in this case the hard disk. If there are many read/write requests waiting in queue to be read from the hard disk, then the entire performance of the datanode can have a detrimental effect.

The memory utilization metric is given a weightage of 20% on the basis of its frequent usage in making computation on the data. Hadoop works under the principle that we want to the bring the computation to the data not the data to the computation. Based on this principle, we assigned it this particular value. If memory required for a given job exceeds the available memory then by the way of the swapfile mechanism memory would be dumped on to the hard disk. This can significantly affect the computation being performed on a particular datanode as we once again would be reading from the hard disk, which is undesirable in this situation. We considered this fact when we decided to give memory utilization a weightage of 20%.

Lastly, the CPU utilization is assigned a weight of 10% which is the lowest because the datanodes performance is usually not bogged down significantly if the CPU has a high utilization. It can take longer for the computation requested to be completed, but this delay in computation is insignificant when compared with the other metrics that play a major role in the performance of a datanode.

All these weightages are converted to an overall score of 10. The 35% factor of the distance is mapped to a score of 3.5 based on the the logical location of the datanode in the network topology. In particular, out of the total contribution of distance metric towards the final score, if a datanode exists on the same machine as the request being made from, then it would be assigned the full score of 3.5. If a datanode exists

on the same rack or a different rack then a score of 2.8 or 2.1 is assigned, respectively.

For converting IO weightage to the range of 0-3.5 we further divided it into three sub-categories of await (max. score 1.5), *rkbps* (max. score 1) and *wkbps* (max. score 1). After analyzing the performance of commodity storage devices, we found that the average speeds for random read and write are approximately 2.72 MBps [18]. We used this value as the basis for our calculation around *rkbps* and *wkbps* scores. We also used the await time metric to determine the average time the system is taking to fulfill IO requests and allotted scores out of 2 on the basis of the same.

Similarly, the score conversion for memory is done based on the free memory percentage of a datanode. The maximum score a datanode can achieve due to memory contribution is 2. Lastly, the CPU score is based on a range of idle CPU percentage. In particular, if the idle CPU percentage of a datanode is between 80 to 100, then it would be assigned a score of 1. If the percentage falls between 60 to 79 range then 0.75 is considered as the score, between 40 to 59 range then 0.5 is assigned as score. Lastly, if the idle CPU falls under 40% then 0.25 is assigned as the score.

On each datanode, metrics that reflect the system state are gathered before they are piggybacked on the heartbeat message being sent to the namenode. We describe below how the value for CPU utilization, memory utilization and I/O load are gathered.

CPU Utilization: At a datanode, we used the output of *top* command which is a performance monitoring program and is available for most POSIX operating systems. It provides a dynamic real time view of running system. We parsed the output of the top command and collected the CPU utilization which is expressed as a percentage of the total capacity of the system. Specifically we used the value of idle CPU percentage from tops output.

Memory Utilization: Since a DFS cluster is almost always a heterogeneous cluster i.e., each machine has different capabilities in terms of processing power, main memory capacity storage capacity, comparing different datanodes only on the basis of used memory space is not very informative. We needed to consider the amount of free memory as a percentage of the total memory available on a machine. For this reason, we gather both the total memory and the free memory of the datanode by using the contents of the */proc/meminfo* file. This file stores a large amount of valuable information about the RAM usage of the system including the total memory and free memory. Both these parameters are added to the information being currently sent to the name node with each heartbeat message.

I/O utilization: For gathering information related to the I/O load on a system we use the output of the *iostat command*. *The iostat command* is used for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates. The availability of *sysstat* package on underlying operating system is a new dependency on Hadoop deployment introduced as a result of our implementation. The values for the following 3 parameters are gathered from the output of the *iostat* command.

- await: The average time (in milliseconds) for I/O requests issued to the device to be served. This includes the time spent by the requests in queue and the time spent servicing them.
- rkB/s: The number of kilobytes read from the device per second.
- wkB/s: The number of kilobytes written to the device per second.

Even though CPU utilization and IO utilization can also be obtained from files within */proc* but the readily available solutions were all written in C and our main aim was to verify and establish our hypothesis rather than focusing on using native solutions for collection of system metric.

The basic premise of our solution was to make the namenode more datanode-state aware. In order to accomplish this, we traced through the source code and modified the implementations to accommodate our requirements. All the inter-node communication in Apache Hadoop are made via Protocol Buffers, the *protos*. These *protos* are used by two entities involved in the communication for serializing the structured data and performing remote procedure calls.

The initial stage of our implementation was to modify these *protos* to transfer data that we needed for making a decision of which datanode to select. We added all the required arguments viz. *memFree, memTotal, cpuUsed, svctmIO, awaitIO, wsecpsIO, rsecpsIO* to the *DatanodeProtocol.proto*. The corresponding changes, to transfer the required parameters between datanode and namenode, were also required in *DatanodeProtocol.java* interface and its subclass *DatanodeProtocolClientSideTranslatorPB.java*. As aforementioned, a heartbeat is sent by datanode to namenode every 3 seconds and this is done by running a thread in class *BPServiceActor.java*. This is the initiation point of heartbeat messages from datanode, therefore, corresponding changes to the function definition and implementation were made here as well.

After this data is transferred over wire to namenode, it is deserialized and *sendHeartbeat()* method is invoked on class *DatanodeProtocolClientSideTranslatorPB.java*, which in turn calls *NameNodeRpcServer.java* which in turn passes it down the pipeline until it reaches the *DatanodeInfo* class where all the metrics are stored in primitive datatypes. We have added multiple variables named same as new argument parameters mentioned above of data types long or double. These values are updated each time a heartbeat is received from datanode.

Now when a client wants to read a certain file in its entirety or a part of it, it invokes *open()* on *DFSClient.java* which overrides it from *ClientProtocol.java*. The open request is handled via DFSClient which then calls *callGetBlockLocations()* after multiple calls to internal functions. This request for blocks is sent over the wire by *ClientNamenodeProtocolTranslatorPB.java* and is handled by *NameNodeRpcServer* on namenode side. *NameNodeRpcServer* upon receiving this request invokes *getBlockLocations()* which finds out all the blocks where the requested file is located and then before returning response to the client, it internally calls *sortLocatedBlocks()* in *DatanodeManager*. In the original Hadoop code the final sorting takes place in *NetworkToplogy.java* on the basis of three-level topology distance, as mentioned earlier.

We have augmented this decision by using this response from *NetworkTopology* class and adding the weightage of CPU, IO and memory to each datanode to sort the entire list of located blocks based on their scores out of 10, in a descending manner. The individual scores of CPU, IO and memory are collected from *DatanodeInfo* class by calling public getter functions of the required variables. This sorted list is then returned to the client which can start the data read process by contacting the first datanode in the sorted list. If the communication with the first datanode fails because of some reason, then the client can try to establish communication with the second datanode in the list and so on.

IV. Performance Evaluation

Benchmarking is the quantitative foundation of any computer system research and is especially critical when evaluating performance for a critical system component such file system. At the same time benchmarking a file and storage system is difficult because complex interactions exist between file systems, I/O devices, caches, kernel daemons and other operating system components. What makes it rather difficult to analyze is the fact that some of these operations may be performed asynchronously and this activity is not always captured in benchmark results. Because of this inherent complexity associated with benchmarking file system, many factors must be taken into account while performing benchmarking and analyzing the results.[11]

In this section we present the testbed and benchmarking strategy adopted to compare the performance of original Hadoop implementation and modified Hadoop for data reads in the presence of hotspots.

System Configuration: We conducted our experiments on Amazons Elastic Compute Cloud (EC2). 4 machines were configured to work as a cluster out of which 3 were Fixed Performance Instances (m3.medium) and 1 was Burstable Performance Instance (t2.medium). An m3.medium machine has 1 dedicated vCPU and 3 EC2 Compute Units (ECU) available to it for computation while in case of t2.medium even though it has 2 dedicated vCPU but the ECU component available to it is variable and is allotted as CPU credits on daily basis which gives you burst of CPU needed for performing compute-intensive applications, if any.

The operating systems were Ubuntu Server 14.04 LTS with Hardware-assisted virtualization (HVM). The system was running a 3.13.0-29-generic kernel. Machines had 30GB of EBS General Purpose (SSD) volume type configured as storage.

A. Benchmarking Procedure

Existing Hadoop benchmarking programs can be roughly categorized into two classes [12]

- *Micro-benchmarks:* The performance is tested against a particular workload that is meant to represent some real-world workload. They are important tests for evaluating Hadoop performance.
- *Synthetic workloads:* They are used to evaluate the effects of the interactions of concurrent Hadoop jobs. They model the characteristics of a Hadoop cluster by collecting data from all the jobs being run on a cluster over an extended period of time.

```
14/07/30 02:29:02 INFO fs.TestDFSIO: ----- TestDFSIO ----- : read
14/07/30 02:29:02 INFO fs.TestDFSIO: Date & time: Wed Jul 30 02:29:02 UTC 2014
14/07/30 02:29:02 INFO fs.TestDFSIO: Number of files: 5
14/07/30 02:29:02 INFO fs.TestDFSIO: Total MBytes processed: 5120.0
14/07/30 02:29:02 INFO fs.TestDFSIO: Throughput mb/sec: 36.082256268587294
14/07/30 02:29:02 INFO fs.TestDFSIO: Average IO rate mb/sec: 36.13706207275390
14/07/30 02:29:02 INFO fs.TestDFSIO: IO rate std deviation: 1.4489020034343776
14/07/30 02:29:02 INFO fs.TestDFSIO: Test exec time sec: 146.998
```

Fig. 1. TestDFSIO Sample Output

For the purpose of this project we only chose to perform micro-benchmarks for now but for better comparison with the original hadoop, benchmarking concerning synthetic workloads should also be performed. It will help establish the correctness of our implemented solution in case of simultaneous reads and writes as well.

The Hadoop distribution which was used for the implementation has a number of benchmarking tools that are bundled with the open source code. These benchmarks are bundled in *hadoop-*test*.jar* and *hadoop-*examples*.jar*. The bundle *hadoop-*test*.jar* contains several testing programs out of which we chose to use *TestDFSIO*.

The *TestDFSIO* benchmark can be used to test both read and writes on HDFS. As already mentioned, it is provided as part of the Hadoop tests for stress testing HDFS and observe any performance bottlenecks with regards to the network and to verify Hadoop setup of namenode and datanodes. *TestDFSIO* also provides statistics in terms of the I/O performance of the entire Hadoop cluster, namely, the throughput in mb/sec and the average I/O rate. These statistics are on the basis of how long it takes for files to be written or read by the map tasks on the Hadoop cluster.

TestDFSIO benchmark test is executed as a MapReduce job to perform a read or write test on HDFS. In order to conduct a *TestDFSIO* benchmark, a write MapReduce job has to be executed first in order to create files of a certain size, which is determined by the command line parameter passed, on the Hadoop cluster. After doing so, a read MapReduce job can be executed on the files generated by write job, by issuing a command with the same parameters as the write MR job, to determine the I/O performance of Hadoop cluster. The *TestDFSIO* benchmark is designed as a 1-to-1 mapping of files to map tasks. In particular, it uses 1 map task per file where the splits are defined in manner where each mapper only gets one file name when the files are created or read as part of the write or read MapReduce jobs respectively.

Usage: *hadoop jar HADOOP_HOME/hadoop-test.jar TestDFSIO -read | -write | -clean [-nrFiles N] [-fileSize MB] [-resFile resultFileName] [-bufferSize Bytes]*

Here, the most notable metrics are *Throughput mb/sec* and *Average IO rate mb/sec*. Both of them are based on the file size written (or read) by the individual map tasks and the elapsed

```
14/07/30 00:36:05 INFO dfsioe.TestDFSIOEnh: 0,0,0,0,0,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,0,0,0,0,0,0,0

14/07/30 00:36:05 INFO dfsioe.TestDFSIOEnh: Average of Aggregated Throughput:
32.25806265968398 mb/sec

14/07/30 00:36:05 INFO dfsioe.TestDFSIOEnh: Standard Deviation :
14.079764157922101 mb/sec

14/07/30 00:36:05 INFO dfsioe.TestDFSIOEnh: Time spots Counted in Average :
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111
```

Fig. 2. En-TestDFSIO Sample Output

time to do so.

$$Throughput(N) = \frac{\sum_{i=0}^{N} filesize_i}{\sum_{i=0}^{N} time_i} \quad (1)$$

As mentioned above, *TestDFSIO* is configured as 1-to-1 mapping of files to map tasks so the number of map jobs that are run is same as the number of files provided as argument to the command, 5 in above case. Each mapper job on start up polls HDFS to get the location of datanodes to read from and starts reading the assigned 1GB chunk. The output of mapper task, mainly the time taken to read the file and the size of file, are stored on local file system and next map job is scheduled. After all the map jobs are done, a single reducer job is invoked which is an accumulator reducer in the case of *TestDFSIO*, it just adds up the data present in output files of map jobs and appends the final output in file *part-00000*. This data is read and printed on console by *TestDFSIO* job as a part of *analyzeResults()* function.

In addition to *TestDFSIO*, we also used another Hadoop benchmark suite called *HiBench*[12]. This benchmark contains 9 typical Hadoop workloads. Among these, *enhanced DFSIO* is an HDFS benchmark that tests the HDFS throughout of a Hadoop cluster by generating a large number of tasks for performing reads and writes simultaneously. Like *TestDFSIO*, *enhanced TestDFSIO* measures the average I/O rate of each map task, the average throughput of each map task, and the aggregated throughput of HDFS cluster. However, unlike *TestDFSIO*, enhanced *TestDFSIO* collects interim values for each run and thus provides results with a much better standard deviation than *TestDFSIO* is capable of. We tested both the original implementation and our implementation using *enhanced DFSIO* for the scenario where the system is under CPU, memory and I/O stress at the same time. The throughput for both the systems was comparable across a number of runs of the test. Sample output of the enhanced HDFS is shown in Figure 2. In the given timeframe, we could not conduct experiments for all 8 configurations using *enhanced DFSIO* but we predict them to be of similar nature as *TestDFSIO*. To further establish the correctness and accuracy of our results we plan to conduct extensive set of testing using both *enhanced DFSIO* and synthetic workloads.

B. Experimental Results

Figure 3 shows the results of the various experiments performed to compare the original Hadoop implementation to the modified version of Hadoop. The bar graph represents the throughput in megabytes per second that was achieved for each of the experiments conducted. As our implementation uses the metrics of CPU, Memory and I/O utilization, the experiments were conducted for various combinations of synthetic workloads. In particular, experiments were conducted with the combinations of CPU Stress, Memory stress and I/O stress turned on and off individually. The CPU stress experiments were conducted by using a POSIX package that was developed by researchers at Harvard [14]. By using *stress* command, a workload is generated that is configurable to determine the amount of stress on the system. The memory stress was maintained on the testbed by mounting a file onto the memory and the size of this file depended on the total memory of the system. As a result of loading files onto the memory, it eventually depleted which resulted in the invocation of the swapfile mechanism. Lastly, the I/O stress was configured by running a script that starts multiple threads and each thread tries to read a file that exists on the distributed file system.

Figure 3 shows that for scenarios where the system was placed under I/O stress only, our implementation achieves a higher throughput than the original implementation. This is also true for the scenario where the system is under CPU, memory and I/O stress simultaneously. For all other scenarios, our implementation is at par with the original Hadoop implementation. An experimental setup, without virtualization, will help us perform controlled experiments to fine tune our implementation and better establish the accuracy of the same.

Small vertical lines on top of bars represents standard deviation across each metrics. Standard deviation represents how tightly the numbers are clustered around the mean in a set of data. If the numbers are pretty tightly bunched together,standard deviation is small and when they are spread out standard deviation is large. We repeated each experiment thrice and calculated the mean from it. We have used standard deviation for our experiments to measure the variation across the three experiments. We observed a slight variation in reported numbers across multiple runs of the tests and hence we felt that standard deviation can represent the extent of dispersion observed in experimental data.

$$StandardDeviation = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2} \quad (2)$$

where, N: total number of experiments, x_i: represent value from ith experiment, $\overline{x}$: is the mean of experiments.

V. Conclusion

The problem of hotspots is well documented across numerous distributed file systems that have been suggested. The current suggested solutions to solve this problem are not adequate in all settings. The authors of GFS suggest to increase the replication factor which in turn may require increasing storage capabilities. However, this solution is not viable in all scenarios, although the framework uses commodity hardware, it is still an expensive solution. In this paper we propose a technique to mitigate hotspots, for Hadoop Distributed File System, by making the namenode more *datanode-state* aware. We have offered a general view of the proposed idea of using different system metrics of the datanode to help the namenode

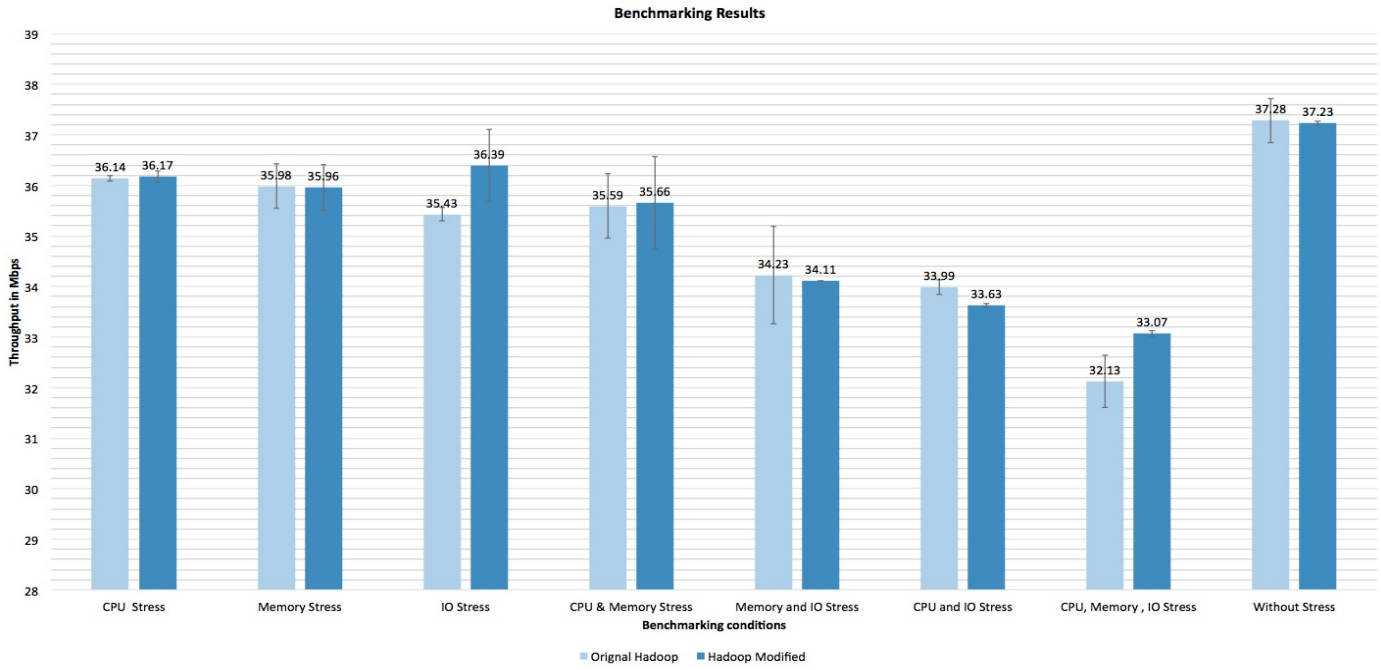

Fig. 3. Benchmarking Results

make a better selection of blocks for the client and also what role these metrics play in the decision making process of the namenode. In addition, we have provided implementation details with regards to the workflow of our proposed solution along with benchmarking results.

There were a few technical challenges we encountered during our implementation and benchmarking phase that were not anticipated during our preliminary research. The Apache Hadoop (v2.4.1) source code had different packages that were linked together and it was not trivial to determine the workflow. Crucial information such as the Network Topology and the Datanode Manager implementations were under different packages of Hadoop-common and Hadoop-hdfs respectively. Due to lack of proper documentation of Hadoop component interaction, it was very difficult to understand the dependencies. During our implementation phase, we had difficulty retrieving the system metrics accurately. There was no standard way to calculate the CPU utilization in the Ubuntu kernel accurately. All the solutions available for usage varied significantly viz. *top, ps, sysstat*. During our benchmarking and testing phase, there were noticeable fluctuations in the performance of Amazon Web Services, namely, the t2.medium instances. These performance fluctuations were not realized until multiple experiments had been conducted.

The benchmarking of the implemented system could have been carried out more rigorously in combination with the heterogeneity of the testbed. Also, the topological distance between a client and data node is very broadly classified into just three categories as discussed earlier. Round trip times (RTT) are a much more accurate measure of the distance between two network entities. We wanted to develop this feature as a part of current implementation but due to time constraints, couldnt achieve this. As future work, we would first like to explore the venue of measuring the actual RTT between the client and datanode. In addition, we would like to use realtime I/O metrics rather than stale information that is currently available as part of the */proc* files in Ubuntu. Lastly, we would like to perform benchmarking by conducting read operation in a manner that ensures equal node-local, rack-local and off-switch data replicas.

REFERENCES

[1] http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm. *Accessed* 07-29-14.

[2] Ghemawat, S., Gobioff, H., Leung, S. T. (2003, October). The Google file system. In *ACM SIGOPS Operating Systems Review* (Vol. 37, No. 5, pp. 29-43). ACM

[3] Schmuck, F. B., Haskin, R. L. (2002, January). GPFS: A Shared-Disk File System for Large Computing Clusters. *In FAST* (Vol. 2, p. 19).

[4] Shvachko, K., Kuang, H., Radia, S., Chansler, R. (2010, May). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on (pp. 1-10). IEEE.*

[5] George Gillson, M. D., Wright, J. V., DeLack, E., Pharm, G. B. B. (1999). Transdermal histamine in multiple sclerosis: Part oneclinical experience. *Alternative Medicine Review*, 4(6), 424-428.

[6] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., Maltzahn, C. (2006, November). Ceph: A scalable, high-performance distributed file system.*In Proceedings of the 7th symposium on Operating systems design and implementation* (pp. 307-320). *USENIX Association*.

[7] Tatebe, O., Hiraga, K., Soda, N. (2010). Gfarm grid file system. *New Generation Computing,* 28(3), 257-275.

[8] Shepler, S., Eisler, M., Robinson, D., Callaghan, B., Thurlow, R., Noveck, D., Beame, C. (2003). *Network file system (NFS) version 4 protocol*. Network.

[9] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R. (2010, June). Benchmarking cloud serving systems with YCSB. *In Proceedings of the 1st ACM symposium on Cloud computing* (pp. 143-154). ACM.

[10] Noronha, R., Panda, D. K. (2008, September). IMCa: a high performance caching front-end for GlusterFS on InfiniBand.*In Parallel Processing, 2008. ICPP'08. 37th International Conference on (pp. 462-469)*. IEEE.

[11] Traeger, A., Zadok, E., Joukov, N., Wright, C. P. (2008). A nine year study of file system and storage benchmarking.*ACM Transactions on Storage (TOS)*, 4(2), 5

[12] Huang, S., Huang, J., Dai, J., Xie, T., Huang, B. (2010, March). The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on* (pp. 41-51). IEEE.

[13] *http://news.magnaglobal.com/articledisplay.cfm?articleid=1463*

[14] *http://people.seas.harvard.edu/ apw/stress/ Accessed* 07-30-14.

[15] *White, T. (2009). Hadoop: the definitive guide: the definitive guide. " O'Reilly Media, Inc."*

[16] *HDFS-4253 https://issues.apache.org/jira/browse/HDFS-4253 Accessed* 07-27-14.

[17] *http://goo.gl/omivWC Accessed* 07-29-14.

[18] *http://www.anandtech.com/show/5729/western-digital-velociraptor-1tb-wd1000dhtz-review/2* Accessed *07-19-14.*