Towards Adaptive Replication for Hot/Cold Blocks in HDFS using MemCached

Pinchao Liu, Adnan Maruf, Farzana Beente Yusuf, Labiba Jahan, Hailu Xu, Boyuan Guan, Liting Hu, Sitharama S. Iyengar

Florida International University
Email:{pliu002, amaru009, fyusu003, ljaha002, hxu017, bguan, lhu, iyengar}@cs.fiu.edu

***Abstract*—With the advancement of ever-growing online services, distributed Big Data storage i.e. Hadoop, Dryad gained much more attention than ever and the fundamental requirements like fault tolerance and data availability become the concern for these platforms. Data replication policies in Big Data applications are shifting towards dynamic approaches based on the popularity of files. Formulation of dynamic replication factor paved the way of solving the issues generated by existing data contention in hotspots and ensuring timely data availability. But from the empirical observations, it can be deduced that popularity of files is temporal rather than perpetual in nature and, after a certain period, content's popularity ceases most of the time which introduces the I/O bottleneck of updating replication in the disk. To handle such temporal skewed popularity of contents, we propose a dynamic data replication toolset using the power of in-memory processing by integrating MemCached server into Hadoop for getting improved performance. We compare the proposed algorithm with the traditional infrastructure and vanilla memory algorithm, as the evidence from the experimental results, the proposed design performs better i.e throughput and execution period.**

***Keywords*-Big Data; HDFS; Replication factor; MemCached**

I. Introduction

With the advent of cloud computation and development of infrastructure as a utility, we witness the emergence of Big Data industries based on data-intensive services i.e. Social Networking, Online Services especially over the last decade imposing computational, data and network traffic on the host servers. From the traditional centered file system, the industry moved towards the distributed scalable file system utilizing MapReduce framework i.e. HDFS [1], GFS [2], Dyrad [3] to cope up with the need of processing millions of user requests every second. Data replication mechanism has facilitated the way of fault tolerance and data availability in the distributed systems. Hadoop [4] is one of the state-of-the-art open source platforms to handle large scale data-intensive applications. Even though HDFS provides scalable and efficient data processing along with fault-tolerance [5], data access and data movement overheads due to high disk I/O are primary bottlenecks of its current architectural design [6], [7].

Data replication is a widely used mechanism in the distributed systems to reduce the overall bandwidth consumption, response time and increase data availability. With the advancement and development of various technologies, new data replication and replica management approaches, both static and dynamic, have been proposed to achieve adaptiveness and better performance [8].

Data replication and placement in Hadoop are uniform where the load balancing and the data locality for optimization are mostly handled by the applications. Trace driven log data analysis and experiments of different popular websites such as Yahoo! and Microsoft's Bing depict the existence of skewed popularity of different data and hotspots in clusters [9], [10]. Consequently, keeping the uniform replication for every file without considering their popularity leads to performance overhead incurring data contention in the nodes denoted as hotspots where the popular data file resides. These drawbacks of HDFS architecture have lead to different dynamic approaches for the replication management. Based on the predictive analysis while keeping redundancy of data storage, dynamic adjustment of replication and replacement has been adapted, and improved algorithms have been introduced to effectively alleviate hotspots and data contention in the existing design [10], [11], [12], [13], [14]. User access histories analysis, probabilistic prediction of data utilization have been included to effectively figure out the Hot and Cold blocks triggering the replication management [15], [16], [17]. Improvement in data availability has been achieved with replication of popular data in more locations than the default one. Replica placement also has been considered in these dynamic approaches [18].

In fact, memory access I/O bandwidth is much higher than that of disk access bandwidth while processing user requests in the clusters. Conventional way of HDFS supports saving and loading each block from the disk resulting in I/O overhead incurring significant performance issues. Moreover, existing HDFS design cannot take the full advantage of the high-performance networks efficiently due to the high latency disk access. Combining in-memory I/O processing, HDFS can potentially overcome the issues.

On the other hand, in-memory storage systems allow applications to cache the results of the queries across the cluster nodes resulting in improved performance in SQL online query processing [19]. So this introduces the pos-

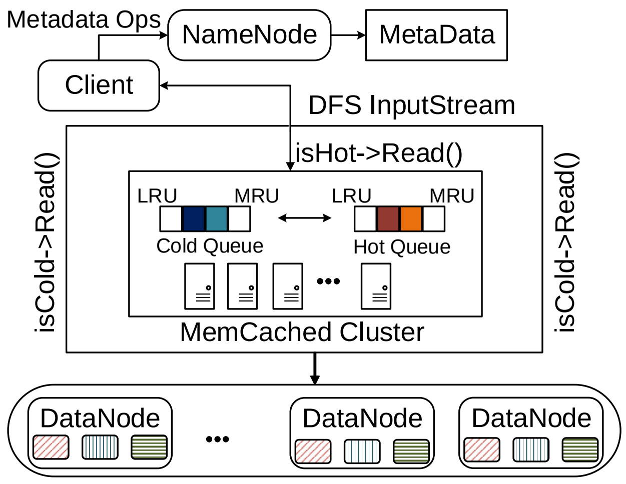


Figure 1. MemCached server integration with HDFS to dynamically replicate popular blocks.

sibility of adapting in-memory processing power in the existing design of distributed Big Data storage systems, and motivating to explore the answers of following questions:

- To improve I/O performance, can we leverage in-memory processing and caching concepts in the traditional HDFS?
- How much overall performance improvement can be achieved with dynamic replication factor for Hot blocks while keeping the existing default replication for Cold blocks?
- Can data contention be alleviated or reduced for DataNodes of HDFS by using distributed in-memory processing?

MemCached [20] is a cost-efficient, high-performance distributed memory caching system to reduce the disk I/O access. It is designed as an in-memory key-value store to speed up the data access in the dynamic real-time applications. Leveraging MemCached, popular Social Networking site Facebook obtained improved performance while providing almost real time communication for millions of users [21].

Integration of MemCached servers in the existing design can either be co-located with the DataNodes or on the separate nodes in the cluster. Placing MemCached servers in a separate location other than the DataNodes will result in reduction of contention. Utilizing MemCached as the main replication block for popular Hot blocks, to reduce the load of reading blocks from HDFS and quickly access to the data, can improve overall performance while maintaining the default replication for fault-tolerance. Overheads of adding and deleting replicas in hard disks consume more computational power and each time the popular file is accessed, it needs to be read from disk according to traditional model. Whereas using MemCached caching capabilities, popular temporal data can be loaded in memory for the faster access and updated based on the timely access to add more replicas when required or deleted when they lose the popularity. The data popularity can be modulated by implementing an efficient dynamic replication algorithm for MemCached. This is more effective and efficient than the conventional approaches of changing the replication factor frequently in HDFS, as it significantly mitigates the disk I/O bottleneck and increasing instant data availability. So in a nutshell, integration of MemCached sever along with HDFS, can be exploited to answer all the above mentioned questions and guarantee high I/O throughput, performance gain for specific Hot blocks and finally reduced data contention in hotspots. Figure 1 shows the basic ideas to integrate these components together to fulfill the expected requirements. The different steps involved in accomplishing the proposed design can be outlined as

1) Configuration of MemCached server on separate nodes in cluster with HDFS for the further experiments.
2) Implementation of effective and dynamic data replication algorithms using MemCached as a caching layer.
3) Evaluation of the proposed system's performance to compare with existing system.

II. Related Work

HDFS (Hadoop Distributed File System) gained popularity for its high performance and reliability but the default replication management system in HDFS is not sufficient to handle the congestion that arises from a high density of user requests. Especially the adaptability of data popularity needs to be handled with a time-based dynamic replication policy for improving the system performance. As a result, adaptive replica management model is needed to ensure improved performance along with fault tolerance and studied from various perspectives.

We divide the related work into three categories: replica allocation and management based systems, skewed popularity based systems, and caching based systems.

A. Replica Allocation and Management Based Systems

Data locality is an important factor to improve the data availability in time. To reach better data locality an algorithm DARE (Distributed Adaptive Data Replication) [12] is proposed where probabilistic sampling and competitive aging algorithm is used in each node to produce solutions of replica allocation and replica placement. Comparing to DARE, the system we propose, that leverages the memory's higher I/O speed to obtain a better replica allocation and placement result.

CDRM (Cost-Effective Dynamic Replication Management Scheme) [8] is another model that can calculate and maintain a minimal number of replica for a given availability requirement. It can adapt to the changes of environment in terms of data node failure and workload changes and maintains a rational number of replica. However, to use CDRM model, it lacks an uniform algorithm to deal with replica number and locality. To use the model we present, it is able to dynamically and uniformly adjust replica locations.

An offline replica allocation algorithm named MORM (Multi-objective Optimized Replication Management) [22] was proposed which looks for near optimal solutions by balancing different optimization factors. However, it only based on existing DFS design to balance the trade-offs. The system we propose, not only consider to optimize the replica, but also add memory component to optimize the overall performance.

B. Skewed Popularity Based Systems

ERMS (Elastic Replication Management System) [23] is an active storage model for HDFS. It dynamically increases the number of hot files and reduces the number of cold files by tracking real time data type. The work is based on probability or constant data type but we use real time data popularity in Hot/Cold block detection mechanism.

Predictive analysis is another approach to dynamically replicate the data file where the utilization of each data can be predicted by probability theory. Then using this utilization information, popular files can be replicated and non-popular files can be deleted. This process improves the availability of data files compared to the default scheme [11].

HDFS-DRM [15] devised a design to solve the hot issues in HDFS based on cloud storage where it makes use of dynamic adjustment mechanism and deletes duplicate node selection mechanism. However, the optimization is in the disk storage level, the disk I/O bandwidth will finally become the bottleneck for the overall application performance.

C. Caching Based Systems

In order to reduce I/O bottlenecks of HDFS, MEM-HDFS [19] was demonstrated using MemCached as a caching system. The main focus is to provide intelligent caching and HDFS data blocks replication with consideration of different deployment strategies for the local and remote MemCached servers. MEM-HDFS implementation resulted in increased throughput and reduced job generation time.

HDFS native cache system, Centralized Cache, which is an explicit caching mechanism that allows users to specify paths to be cached by HDFS. The NameNode will communicate with DataNodes that have the desired blocks on disk, and instruct them to cache the blocks in off-heap caches [24].

In MEM-HDFS or Centralized Cache, they make use of only caching concept while in the system we merge caching and popularity concept together. After detecting the Hot blocks we cache them in MemCached that gives more flexibility in reducing data contention.

III. Architecture Design

Although different studies have been conducted to improve the dynamic replication management, there exist different trade-off among the various design choices. Therefore, we propose a toolset integrating MemCached as a caching layer to handle data popularity for dynamic replication management with new proposed algorithm.

A. Data Generation

BigDataBench [25] is used as a benchmark for the work and generated synthetic text data using its big data generation tools named BDGS (Big Data Generator Suite). BDGS module generates data in three steps:

- Application-specific and real-world data selection.
- Generation models construction, parameters and configuration derivation from data.
- Provide extensive workload testing.

After data generation, we integrate multiple workloads in BigDataBench to process the data set. The Wikipedia entries are used as the dataset and are processed to two different workloads `Word Count` and `grep`. Different sizes of data files are generated as input into HDFS.

B. Dynamic Model to Copy Hot Blocks to MemCached

HDFS plays the role as data saving layer, it is usually shared by multiple upper level applications. That means it will be hard to expect all these applications using the consistent way to access data from HDFS. Using the predefined Vanilla model will not be able to cater to variety access patterns to the data.

To more effectively use the limited space in memory, we propose the WLRU-MRU collaborative dynamic replacement algorithm. Its main idea is that each accessed data block will have a higher possibility to be accessed again soon that will have a much higher weight in the priority queue.

Algorithm 1 shows the procedure of how to move the data block in Hot queue and Cold queue in MemCached by combining LRU and MRU algorithm to collaboratively do the replacement. LRU makes up for the deficiency of the LRU by introducing the concept of two checks. MRU replacement policy as the most recently used data block will be evicted, when a block of data is missing. The time prediction of MRU is higher than LRU. As for subsequent call for the Hot blocks get the data from MemCached server, read I/O performance will be improved as the fact is that memory I/O is much more faster than disk I/O.

C. Caching with MemCached

The proposed design leverages a caching mechanism which can improve the system performance and perform task based on time sequence. Since MemCached is a key-value in-memory storage, it is customized to store the detected Hot data blocks over a certain period of time. All DataNodes will be configured with local MemCached which will ensure data locality. When specific block becomes Hot, for the first time we will have to set the contents in MemCached as key-value pair. For all the subsequent calls, it will be read from

Algorithm 1: Dynamic collaborative replacement algorithm

```
Input: Label M, Label N
if (M==0) //datablock is not in MemCached. then
    if (N==0) //tag hit is 0 then
        LRU; //call LRU
        Replace(bottom); //replace the data at bottom
        MoveDown(other data); //other data move down in turn
    if (N==1) //tag hit is 1 then
        MRU;
        //call MRU
        Replace(top); //replace the data at top
        MoveUp(other data); //other data move up in turn
if (M==1) //datablock is in MemCached. then
    if (N==0) //tag hit is 0 then
        MRU; //call MRU
        minHeap.put(<hitData.id, hitData.weight++>);
        MoveUp(other data); //other data move up in turn
    if (N==1) //tag hit is 1 then
        LRU; //call LRU
        maxHeap.put(<hitData.id, hitData.weight++>);
        MoveDown(other data); //other data move down in turn
```

MemCached rather than DataNode till the blocks remain Hot.

Advantages of MemCached over other approaches are that it can handle high memory load for it's distributed characteristics, responds quickly and finally provides accurate expiration times. Since the main focus is replication depending on time, MemCached has the advantages over other in-memory key-value storage and it perfectly suits in the design choice.

D. Evaluation Metrics

To compare the performance of the proposed model, we need the benchmark performance which was gathered from the initial log analysis of Hadoop without any modification and the one with vanilla MemCached integration, which using a simple threshold value to control the Hot/Cold blocks. Job execution time, I/O throughput, CPU usage and memory usage are considered as the performance metric for all the workloads. All of the metrics will be recomputed using modified replication management scheme in Hadoop and will be used to evaluate the overall performance.

To evaluate the proposed design, we analyze the same performance metrics taken initially and expecting to get a performance improvement in all the metrics.

Even though we are expecting several improvements, there might be some overhead involved as follows:

- In case of cache miss, there will be extra overhead in the I/O operation since an additional layer is involved.
- Since the Hot blocks will get change over time, updating the MemCached will be another overhead.
- Due to the limited size of caching, we have to design an approximate policy for the blocks replacement in MemCached.

IV. Experimental Results

The test is performed on the server that has two Intel processors, 16GB of memory and 6TB hard drives. The different sizes (from 100GB to 500GB) of data is generated with BigDataBench BDGS module and, performed two different operations on these synthetic data. The experimental results includes i) identification of Hot and Cold blocks in HDFS (Hadoop 2.8.2 [4]), and ii) Exploration of the initial performance metrics of the system from log analysis.

A. Dynamic Hot and Cold Block Detection

To detect which blocks are hot and which are cold in HDFS, first we generated a 60 min stream of block access using randomization. Then, we analyzed the Global map to get the number of the read count for each block. In the settings, 16 data blocks are stored in the Hadoop cluster, each block contained 524MB data generated by BDGS.

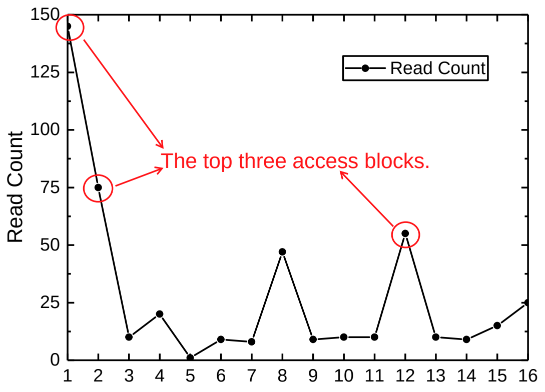


Figure 2. Read count of HDFS data blocks

Figure 2 shows the read count history of each 16 data blocks of the HDFS. Here, data block 1, 2, and 12 are the top three blocks accessed during the time period. Meanwhile, data block 5 is the least accessed block. Global key-value store is used, so every node in the Hadoop cluster can access the status of Hot and Cold Blocks.

Using any other kinds of storage for keeping record of Hot/Cold status, like file will result in more I/O overhead and might hurt the improvement gain. So using the in-memory key-value store for Hot and Cold block detection is leveraged for the higher performance. This will enable the function to analyze the HDFS file access in real time and detect the Hot blocks in the system.

B. *Vanilla Model to Copy Hot Blocks to MemCached*

In this vanilla model, MemCached is used as a caching system to alleviate the loads of user requests based on a predefined threshold value. Algorithm 2 describes the Hot block detection mechanism. When DFSClient requests for a specific data block, NameNode looks for all the available DataNodes options and return the LocatedBlocks as a list to the client. Then the client processes to choose the best node and passes the request to DFSInputStream to handle packet transfer from DataNode. DFSInputStream uses BlockReader and PacketReceiver sequentially to read the packets from the stream. Then checking the access pattern in DFSInputStream read function to update the access counts in a map as key-value. When the access count for any block reaches the predefined threshold, the system will apply caching techniques.

We compare the proposed design with this vanilla model and the default HDFS infrastructure. Since MemCached has the expiration time, if a block becomes Cold after certain period, it will be automatically deleted which let the vanilla model also serves the purpose of simple dynamically changing replication.

Figure 3 shows the process of moving in the Hot block and moving it out when it becomes Cold. This process does not require any complicated calculation, but it is required to define the Hot block threshold and the expire time in advance. These setups require the experience of using the HDFS applications, meanwhile, the applications accessing data blocks method tends to be consistent during the life cycles.

All data blocks are not accessed uniformly in HDFS. Depending on the popularity, different blocks could be accessed more frequently than the others making the residing DataNodes hot. So, detecting the Hot and Cold blocks dynamically using the access history is one of the main challenges. Dynamic detection steps include populating the file access stream with a randomized scheme over a certain period of time to represent the real life scenario. During the streaming time, we analyze the file access requests from the DFSClient (Distributed File System Client) and the most frequently accessed file will be identified as Hot blocks for the future steps.

C. *Performance Analysis*

To gather the current system performance metrics, we used two workloads from BDGS: `Word Count` and `Grep`. The workloads are applied on different sizes of data stored in HDFS. `Word Count` data source is generated by the program which randomly picks words from a dictionary file, and then counts the number of occurrences of each word in

Algorithm 2: Dynamic Hot and Cold block detection

```
Input: BlockID and SequenceNo of
       CurrentLocatedBlock that is being read from
       DFSClient, Map with (BlockID,AccessCount),
       threshold
Output: Set status of BlockID
1  $key$ = $CurrentLocatedBlock.BlockID + SequenceNo$;
2  if ($key$ exists in $Map$ ) then
3      $AccessCount$ = Map.get($key$) + 1 ;
4      Map.put($key, AccessCount, timer$) ;
5  else
6      Map.put($key, 1, timer$) ;
7  end
8  $AccessCount$ = Map.get($key$);
9  if $AccessCount > threshold$ then
10     $status(BlockID)$ = hot;
11     Set BlockID content in MemCached ;
12 else
13     $status(BlockID)$ = cold;
14     Normal operation;
15 end
```

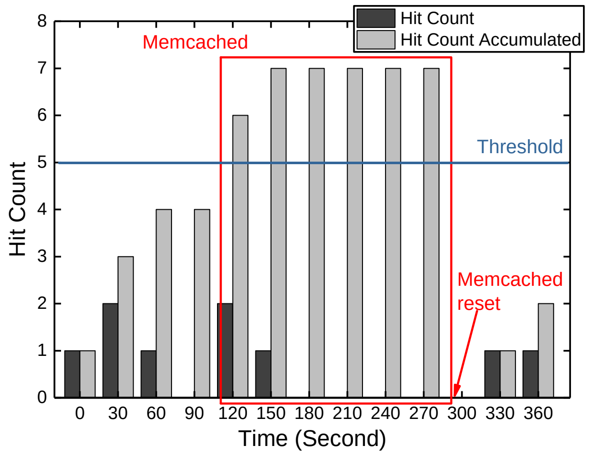
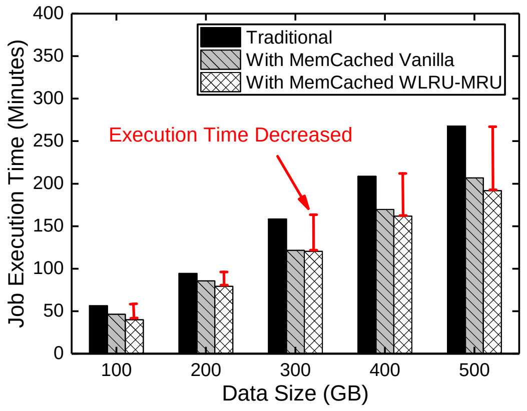


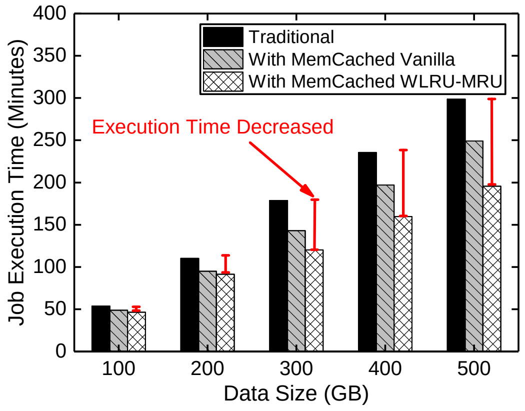
Figure 3. After hit count is over the predefined threshold, the block is defined as Hot and MemCached records it. After a specific period, once the hit count stops increasing, MemCached will reset the records and release the occupied memory.

the given input set. `Grep` extracts matching strings from text files and counts how many time they occurred.

Figure 4 shows the jobs execution time comparison results between original HDFS framework and the proposed dynamical model design. Figure 4(a) is about `Word Count` execution time. It shows after applied new design, the execution time decreased, especially for the bigger data size input, the decreased time is much more remarkable. For 400GB and 500GB file size input, the execution time decreases 36% and 29% respectively comparing to the default setup, even 10%

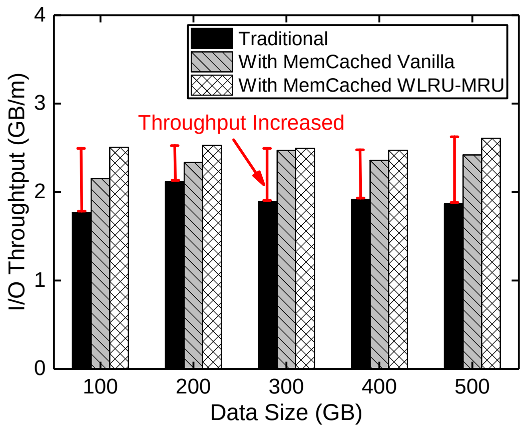


(a) Word count execution time.

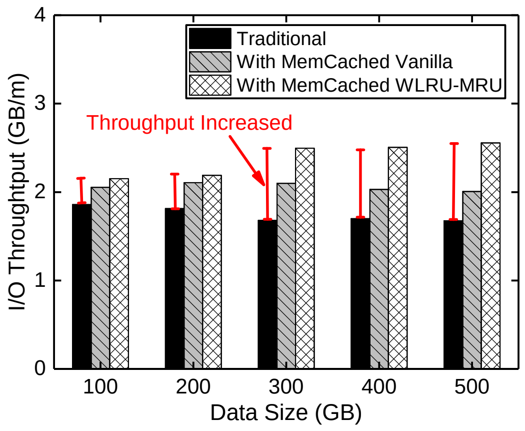


(b) Grep execution time.

Figure 4. Job execution time for word count and grep.



(a) Word count I/O throughput.



(b) Grep I/O throughput.

Figure 5. I/O throughput for word count and grep.

and 17% comparing to the vanilla MemCached integration. Figure 4(b) is about `Grep` execution time. Although there is no clear trend of the impact for execution time along with data size increasing, the performance still increased after integrating with MemCached.

Figure 5 shows the evaluation of read performance in terms of throughput. The throughput factor is not much sensitive with the changes of input data size. Worth to note is, that after implementing MemCached, both `Word Count` and `Grep` got increased throughput, since Hot block data are available to be retrieved from memory which far quickly than retrieving data from the disk.

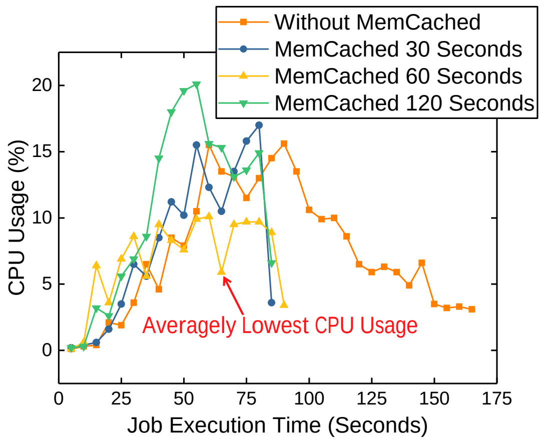
For the overhead analysis, Figure 6(a) shows that 60s expiration window is the optimal one in the testing cases and CPU time is the lowest for this configuration which is even lower than the original default HDFS setup. The reason can be explained as the I/O wait time is negligible because of the in-memory caching. That means deployed MemCached would cost some CPU resource, but the saved I/O wait time can neutralize the impact. Figure 6(b) shows that, after implementing MemCached into HDFS, the memory cost is increased, that because these memory cost is majorly used to save Hot block dynamically. This is the trade-off for the system performance increase.

V. Conclusion

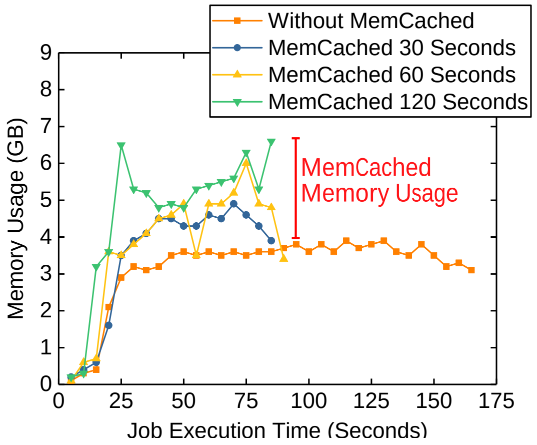
In this paper, we analyze the possibility to improve the HDFS data blocks replication mechanism. The caching strategy MemCached is leveraged in the design to effectively replicate the popular blocks in memory. The experimental results show that the proposed design is able to improve the HDFS based applications' performance from different perspectives and has the potential to overcome the critical issues of traditional Big Data storage systems.

VI. Acknowledgment

The authors would like to thank the anonymous reviewers for their feedback that improved the paper significantly. We'd thank Florida International University Graduate & Professional Student Committee and School of Computing and Information Sciences for the travel award to present this



(a) CPU usage comparison.



(b) Memory usage comparison.

Figure 6. CPU and memory usage comparison for the different MemCached periods with default HDFS setup.

work. This work is supported by Department of Homeland Security (DHS 2017-ST-062-000002).

References

[1] D. Borthakur *et al.*, "Hdfs architecture guide," *Hadoop Apache Project*, vol. 53, 2008.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] M. Isard *et al.*, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS operating systems review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[4] "Hadoop," http://hadoop.apache.org.

[5] K. Shvachko *et al.*, "The hadoop distributed file system," in *MSST*. IEEE, 2010, pp. 1–10.

[6] J. Shafer, S. Rixner, and A. L. Cox, "The hadoop distributed filesystem: Balancing portability and performance," in *IS-PASS*. IEEE, 2010, pp. 122–133.

[7] X. Liu and J. Wang, "The study on capacity enhancement of distributed systems cloud services," in *LEMCS*. Atlantis Press, 2014.

[8] Q. Wei *et al.*, "Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster," in *CLUSTER*. IEEE, 2010, pp. 188–196.

[9] A. K. Karun and K. Chitharanjan, "A review on hadoophdfs infrastructure extensions," in *ICT*. IEEE, 2013, pp. 132–137.

[10] G. Ananthanarayanan *et al.*, "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proc. 6th conf. on Computer systems*. ACM, 2011, pp. 287–300.

[11] D.-M. Bui *et al.*, "Replication management framework for hdfs based on prediction technique," in *Advanced Cloud and Big Data*. IEEE, 2015, pp. 58–63.

[12] C. L. Abad, Y. Lu, and R. H. Campbell, "Dare: Adaptive data replication for efficient cluster scheduling," in *CLUSTER*. Ieee, 2011, pp. 159–168.

[13] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.

[14] H. H. H. Aung and N. N. Oo, "Edas: Efficient data access scheme of data replication for hadoop distributed file system (hdfs)," 2015, pp. 177–183.

[15] M. Li, Y. Ma, and M. Chen, "The dynamic replication mechanism of hdfs hot file based on cloud storage," *International Journal of Security and Its Applications*, vol. 9, no. 8, pp. 439–448, 2015.

[16] J. Appavoo *et al.*, "Providing a cloud network infrastructure on a supercomputer," in *19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 385–394.

[17] S. Ding and Y. Li, "Lru2-mru collaborative cache replacement algorithm on multi-core system," in *CSAE*, vol. 2, May 2012, pp. 395–398.

[18] D.-W. Sun *et al.*, "Modeling a dynamic data replication strategy to increase system availability in cloud computing environments," *Journal of computer science and technology*, vol. 27, no. 2, pp. 256–272, 2012.

[19] N. Islam *et al.*, "In-memory i/o and replication for hdfs with memcached: Early experiences. in 2014 ieee intl," in *IEEE BigData*, 2014.

[20] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[21] R. Nishtala *et al.*, "Scaling memcache at facebook." in *nsdi*, vol. 13, 2013, pp. 385–398.

[22] S.-Q. Long, Y.-L. Zhao, and W. Chen, "Morm: A multi-objective optimized replication management strategy for cloud storage cluster," *Journal of Systems Architecture*, vol. 60, no. 2, pp. 234–244, 2014.

[23] Z. Cheng *et al.*, "Erms: An elastic replication management system for hdfs," in *CLUSTER WORKSHOPS*. IEEE, 2012, pp. 32–40.

[24] "Centralized cache management in hdfs," https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html.

[25] "Bigdatabench," http://prof.ict.ac.cn/.