2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

In-memory Caching Orchestration for Hadoop

Jaewon Kwak, Eunji Hwang, Tae-kyung Yoo, Beomseok Nam, and Young-ri Choi
School of Electrical and Computer Engineering
Ulsan National Institute of Science and Technology (UNIST)
Ulsan, Korea
Email: {jwkwak,hwangej88,tkyoo,bsnam,ychoi}@unist.ac.kr

***Abstract*—In this paper, we investigate techniques to effectively orchestrate HDFS in-memory caching for Hadoop. We first evaluate a degree of benefit which each of various MapReduce applications can get from in-memory caching, i.e. cache affinity. We then propose an adaptive cache local scheduling algorithm that adaptively adjusts the waiting time of a MapReduce job in a queue for a cache local node. We set the waiting time to be proportional to the percentage of cached input data for the job. We also develop a cache affinity cache replacement algorithm that determines which block is cached and evicted based on the cache affinity of applications. Using various workloads consisting of multiple MapReduce applications, we conduct experimental study to demonstrate the effects of the proposed in-memory orchestration techniques. Our experimental results show that our enhanced Hadoop in-memory caching scheme improves the performance of the MapReduce workloads up to 18% and 10% against Hadoop that disables and enables HDFS in-memory caching, respectively.**

***Index Terms*—Hadoop; In-memory caching; Cache locality; Cache Affinity; Scheduling policy; Cache replacement algorithm; Performance evaluation**

I. Introduction

Apache Hadoop [1], an open source MapReduce framework, has become one of the most dominant software platforms for large scale dataset processing due to its easy programming model, robustness, and scalability. To mitigate the expensive disk I/O phase and increase the memory utilization, the in-memory caching feature has been developed in Hadoop 2.3.0 [2]. However, the centralized in-memory cache management in HDFS is still far from satisfactory, as most Hadoop jobs see marginal improvement in terms of end-to-end latency [3], due to its low utilization of cached data, high startup costs, high CPU computations, slow shuffle and reduce phases, lack of cache replacement schemes considering workload patterns, etc.

In this work, we redesign Hadoop with HDFS in-memory caching in order to take better advantage of HDFS in-memory cache. We first analyze a degree of benefit which each of various MapReduce applications can get from in-memory caching, i.e. *cache affinity*. We then present an *adaptive cache local scheduling algorithm*, which adaptively computes how many times a MapReduce job is skipped to be scheduled on a cache local node (which has a cached data block for the job in its memory). The number of skips is set to be proportional to the percentage of cached input data for the job. We also develop a *cache affinity aware cache replacement algorithm*, which evicts data blocks of MapReduce applications with low cache affinity. Finally, we evaluate the performance of our enhanced Hadoop over various workloads composed of multiple MapReduce applications. Our experimental results show that our enhanced Hadoop improves the performance of the MapReduce workloads up to 18% and 10% against Hadoop that disables and enables HDFS in-memory caching, respectively.

The main contributions of this work are as follows. First, we extend the HDFS in-memory caching system such that the unit of caching is a data block, not an entire file. This enhancement enables fine-grained *block granularity caching*, that is, concurrent jobs can request to cache blocks on demand with small overhead based on workload patterns as in OS page cache. Second, we devise an adaptive cache local scheduling algorithm, which tries its best efforts to utilize cached data blocks, while reducing the scheduling overhead. Third, we develop a cache affinity aware cache replacement algorithm, which allows MapReduce applications with high cache affinity to fully exploit in-memory caches. Finally, we conduct experimental study to show the effects of the efficient orchestration of Hadoop in-memory caching.

There have been enormous efforts such as [4, 5] to improve Hadoop by utilizing in-memory caching for reusing previously computed data. In PACMan [6], cache eviction policies, which evict data blocks from large incomplete inputs based on "all-or-nothing" property, were investigated for parallel jobs, and an unlimited number of cache replicas is allowed for a block to achieve cache locality. In Twister [7], input data are categorized as static and dynamic, and only static data are cached for reuse. In our work, by employing a cache local scheduling, we avoid redundant cached blocks, utilizing the saved in-memory cache spaces for other blocks, and also utilize in-memory caches efficiently based on application characteristics.

II. Background

Centralized cache management in HDFS allows users to specify what file blocks should be cached in OS page cache [2]. When a client requests to cache a specific file, the cache request message is transmitted to the central NameNode which knows where the partitioned data blocks for the file are located. The NameNode piggybacks the cache request message on a heartbeat response and delivers it to DataNodes where the data blocks are stored. The DataNodes store the data blocks in their OS page caches (by using `mmap()` and `mlock()`) and periodically send the NameNode *cache reports* that describe

978-1-5090-2453-7/16 $31.00 © 2016 IEEE
DOI 10.1109/CCGrid.2016.73

TABLE I
CACHE AFFINITY OF EACH APPLICATION

App	Aggre.	Grep	Join	KMeans	PageRank	Sort	WordCount
Affinity	0.303	0.561	0.090	0.041	0.000	0.000	0.019

what data blocks are cached in their *in-memory caches*. The NameNode collects the cache reports and manages the global information about the cluster cache state. The number of cached copies for a data block can be specified as an additional parameter, *cache replication factor*, whose value should be smaller than or equal to the number of persistent replicas for the block in HDFS.

One of the limitations in the current HDFS in-memory caching implementation is that users should manually specify what files should be cached and uncached. For an input file of each application that users want to cache, they must use a command to add the file to the in-memory cache. Moreover, while the OS page cache employs an LRU-like algorithm for its cache replacement, HDFS in-memory caching does not replace previously cached data blocks unless users ask to uncache data blocks of certain files. Along with the inconvenience of explicit requests for caching and uncaching certain files, this policy does not work well if multiple jobs require their own files to be cached and the working set size is larger than the in-memory cache size.

Another limitation in the current implementation of Hadoop is that the latest *Yarn* scheduler does not consider the cache locality in its scheduling algorithms although the global cluster cache state is available in the NameNode and the cache state information can be used to schedule incoming tasks. The current Hadoop scheduler in Yarn requests a computing resource on a specific node or rack considering the *data locality*, but not the *cache locality*. If the data locality constraints can not be met, i.e., a task cannot run on a DataNode that hosts an input data block or a replica, the scheduler tries to run the task on a node in the same rack (*rack local*), or any other node in the cluster (*off-rack*) if all nodes in the rack are also busy (as known as delay scheduling [8]). Hence, only when all persistent replicas are cached in HDFS in-memory cache, we can take the maximum benefits of cached data.

III. CACHE AFFINITY OF MAPREDUCE APPLICATIONS

To evaluate the performance of single MapReduce application workloads using in-memory caching, we have extended Hadoop-2.4.1 (denoted as `CL`) via implementing *block granularity caching* in which the unit of caching is a data block, not an entire file, and a simple cache local scheduling algorithm, which skips up to C times to achieve the cache locality before trying to launch a data local task for a MapReduce job. (Block-level caching for Hadoop was similarly implemented in [9].) The detail of our caching mechanism is described in Section IV.

We measure the performance of our modified Hadoop `CL`, and Hadoop `no-cache` which does not utilize in-memory caching for seven MapReduce applications. The same experimental setup for multiple MapReduce application workloads, which is discussed in detail in Section V, is used. In the experiments, the maximum number of skips to enforce the data locality is set to 10 for all cases. For `CL`, the runtime of each application is measured for *hot cache*, in which all the input data blocks of the application have been loaded into the caches to warm up the caches before running the application, and C is set to 5.

For a MapReduce application, we then define a simple *cache affinity (CA)* metric, which indicates how much performance improvement it can get using in-memory caching, as follows:

$$CA = MAX(1 - \frac{R_{CL-hot}}{R_{no-cache}}, 0)$$

where $R_{no-cache}$ and R_{CL-hot} are the runtimes of the application with `no-cache`, and `CL` for hot cache, respectively. The higher value this metric has, the larger benefit the application gains. Table I shows the computed cache affinity values of the MapReduce applications. The performance of Grep can be improved by around 56% against `no-cache`, while the performance gain of PageRank and Sort is none.

IV. ORCHESTRATING HADOOP IN-MEMORY CACHING

Enhanced in-memory caching system In our system, a map task for processing a data block always sends the NameNode a request to cache the block. To reduce caching overhead, we always attempt to cache the block in the in-memory cache of a node which has recently read the block from the disk storage and so the block is likely still cached in the OS page cache. Thus, if a map task reads the block from a remote node, the task requests to cache the block in the remote node, not the node currently executing it. The NameNode controls which data blocks should be cached and replaced in the requested node using a cache replacement algorithm. The NameNode also maintains and periodically sends the *percentage of cached input data* (i.e. the percentage of cached blocks over all the input data blocks) for each application to the resource manager. On requesting resources to execute map tasks for a MapReduce job, its application master queries the cache status of the input blocks, and sends the status to the resource manager. The scheduler in the resource manager makes a scheduling decision based on the percentage of cached input data and cache status for the job.

Adaptive Cache Local Scheduling Algorithm Our adaptive cache local scheduling algorithm computes the number of times to skip for achieving the cache locality for a MapReduce job dynamically based on its percentage of cached input data. In the algorithm, the maximum numbers of skips to enforce cache and data localities are initially given as C and D, respectively. For a job j, the number of skips to enforce the cache locality (C_j) is adapted to be proportional to the current percentage of cached input data of job j. When the percentage of cached input data of j is currently low, the probability of launching a cache local task also becomes low, and thus, C_j is better decreased to reduce its job scheduling delay. If the job skips up to C_j times, and there is a free slot with data locality,

it launches a data local task. If the job skips up to D times, it launches a task on any node as in the delay scheduling [8].

Cache Affinity Aware Cache Replacement Algorithm Our cache affinity aware cache replacement algorithm basically replaces data blocks of applications with low cache affinity with those with high cache affinity. In the algorithm, the cache affinity metric value and the probability P_i of caching a data block of each application A_i are provided as input. The probability P_i needs to be predefined based on the cache affinity for each application. For example, in our experiments in Section V, we divide the seven applications into three categories of high, medium, and low cache affinity, and use three different probabilities of 100%, 50%, and 0%, respectively, for them. For a request of caching a block of an application A_i on a node n, if the block is already cached in some node, which can be different from n, of the cluster, the access time for the block is updated, and the *time-to-live* (TTL) of the block, which is a future expiration time, is also extended.

For an uncached data block, the algorithm attempts to cache the block with probability P_i. This step can help reduce the overhead of replacing cached blocks, since if data blocks from an application that has low P_i are cached, the blocks will be likely replaced by blocks from applications with high cache affinity. Thus, the algorithm makes such blocks with low P_i have a small chance to be added to the in-memory cache.

Once the algorithm decides to cache the block, if node n has enough space, the block is cached in n. Otherwise, it first searches cached blocks with expired TTL on node n. However, if there is no such block, it searches cached blocks with the lowest cache affinity. In node n, if the cache affinity values of all the cached blocks are higher than or equal to that of the application for the requested block, the block will not be added to the in-memory cache. If there are multiple candidates for eviction, we use the least recently used (LRU) algorithm to select a victim block. Note that using TTL, we make sure that cached data blocks with a higher cache affinity will be eventually evicted if they are not accessed for a long time.

V. Evaluation

In this section, we evaluate the performance of our enhanced Hadoop over various workloads where multiple MapReduce applications concurrently run.

Methodology For our experiments, we use a cluster of 11 nodes with Ubuntu 12.04 LTS. Each node is configured with two Intel Xeon octa-core E5-2650, 64 GB memory, and 1 TB HDD, and is connected with each other via a 10 Gigabit Ethernet switch. For Hadoop, the amounts of memory configured for a map task, a reduce task, and a node manager are 1 GB, 2 GB, and 12 GB, respectively. The block size of files in HDFS is 128 MB, and the number of replicas is three. For other Hadoop configuration parameters, the default values are used. The size of in-memory cache is set to 18 GB for each worker node in the cluster, having total 180 GB HDFS in-memory cache. Note that we configured a resource shared cluster such that Hadoop runs with a co-running application, which uses 32 GB memory all the time, on each worker node.

TABLE II
Workloads composed of multiple MapReduce applications

W	App1	App2	App3	App4	# unique inputs	cache affinity			Total size (GB)
						High	Medium	Low	
W1	Grep(A)	PR	Sort(A)	WC(B)	3	1	1	2	194.9
W2	Aggre.	Grep(A)	PR	WC(B)	4	2	1	1	280.0
W3	Aggre.(C)	Grep(A)	Join(C)	WC(B)	3	2	2	0	277.7
W4	Grep(A)	Join	KMeans	WC(B)	4	1	3	0	314.8
W5	Aggre.	Grep(A)	Sort(A)	WC(B)	3	2	1	1	261.9
W6	Aggre.	PR	Sort(A)	WC(A)	3	1	1	2	191.6

To evaluate the performance of Hadoop with HDFS in-memory caching, we use seven MapReduce applications of `Aggregation`, `Grep`, `Join`, `KMeans`, `PageRank`, `Sort`, and `WordCount` from the Intel HiBench [10]. Their input sizes are in the range of 18 GB and 101 GB. For KMeans, the number of iterations is set to five and for PageRank, it takes two iterations to converge. For Join, it has two sets of input files, one of which can be shared with Aggregation. For Grep, Sort, and WordCount, they can use the same input data, which is a synthetically generated text file.

For a workload composed of four MapReduce applications, each application is allocated 25% of the cluster resources. Table II shows six workloads of multiple MapReduce applications, along with the number of unique input files and total input size. For each of the applications that can share the input with other applications, a file name is provided with it. For the experiments, we categorize Grep and Aggregation as high cache affined applications, WordCount (WC), Join, and KMeans as medium cache affined ones, and PageRank (PR) and Sort as low cache affined ones.

For each experimental run, an application in a workload starts to run without any input data loaded into in-memory cache (i.e. *cold cache*), and continues to run for around 30-100 minutes. Thus, we submit and execute the applications except KMeans multiple times, depending on the runtime of each application in a workload, so that the applications in the same workload complete their execution similarly.

We compare the performance of our modified Hadoop `CL-CA`, which employs fine-grained *block granularity caching*, *adaptive cache local (CL) scheduling*, and *cache affinity (CA) aware cache replacement* proposed in Section IV, against the following versions of Hadoop:

- Hadoop `no-cache`: It does not utilize HDFS in-memory caching, and it is used as a baseline.
- Hadoop `DL-NR`: It utilizes file granularity caching and the data local *(DL)* scheduling policy. In `DL-NR`, a request to cache input files for each of MapReduce applications in a workload is initiated concurrently at the start of the workload, and no cached block is replaced *(NR)* once it is cached during an experimental run. When the total input size is larger than the total in-memory cache size, some part of input blocks from four applications will be cached in a random way. The cache replication factor for `DL-NR` is fixed to 1.
- Hadoop `CL-LRU`: It works in the same way as `CL-CA`,

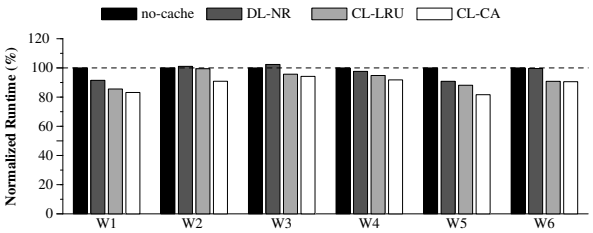

Fig. 1. Average normalized runtimes of MapReduce workloads

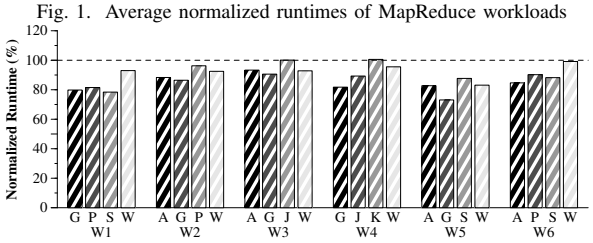

Fig. 2. Normalized runtimes of applications in each workload

except that it uses the traditional LRU policy instead of our cache affinity aware cache replacement policy.

For `CL-CA` and `CL-LRU`, the maximum number of skips (C) for cache locality is experimentally tuned to 3, but for Sort and PageRank whose cache affinity is 0, C is set to 0. Only initial input files (not output files used as input for subsequent jobs in iterative applications) are cached in in-memory cache.

Results The average normalized runtimes of the six multiple workloads with `no-cache`, `DL-NR`, `CL-LRU`, and `CL-CA` are presented in Figure 1. (For each of the applications in a workload, its runtime is normalized to that with `no-cache`, and then the normalized runtimes of the applications in the workload are averaged.) On average, the performance improvement with `CL-CA`, compared to `no-cache`, `DL-NR`, and `CL-LRU`, is 11.29%, 8.60% and 3.87%, respectively. The improvement of `CL-LRU` is 7.59% against `no-cache`, which is lower than that of `CL-CA`, but is higher than that of `DL-NR`. The improvement of `DL-NR` is small as 2.83% on average against `no-cache`. Also, in Figure 2, the normalized runtime of each application with `CL-CA` is shown for each workload.

From the experiments of the six workloads, the followings are generally observed: (1) The performance of Grep and Aggregation with `CL-CA` is improved in all cases. However, the degree of performance improvement in a workload of multiple applications is mostly smaller than that in a workload of a single application. (2) Sort and PageRank have no gain to use the in-memory cache when they are executed alone. However, the performance of Sort and PageRank improves up to 21.6% and 18.5%, respectively, in a workload of multiple applications. The performance of these "I/O heavy" applications requiring large write throughput can be improved if they are executed with other applications with different resource usage patterns (i.e. CPU bound). Moreover, in case of Sort, it can get benefits of reusing cached data blocks by using the same input file as Grep and WordCount. (3) The performance improvement with `CL-CA` depends on co-running applications in a workload, especially for WordCount and Join. (4) When most of data blocks can be cached as in W1 and W6, `CL-CA` and `CL-LRU` behave similarly, showing similar performance.

Throughout our performance study, we demonstrate that by orchestrating in-memory caching effectively, the performance of MapReduce applications can be improved significantly, even for low cache affined applications, since using in-memory caching alleviates I/O performance bottleneck, and also show that the performance can be strongly affected by co-running MapReduce jobs.

VI. Concluding Remarks

In this work, we extended Hadoop with in-memory caching via implementing the adaptive cache local scheduling algorithm that tries to enforce cache locality while reducing the scheduling overhead, and the cache affinity aware cache replacement algorithm that favors MapReduce applications with high cache affinity, and showed its performance improvement over the current Hadoop. Studying a more sophisticated cache affinity metric and an efficient scheduling algorithm that considers the effects of co-runners as well as the cache locality is considered as our future work.

Acknowledgment

This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1501-04.

References

[1] Apache Hadoop. *http://hadoop.apache.org/*, 2016.

[2] Apache Hadoop Centralized Cache Management in HDFS. *http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html*, 2016.

[3] Andrew Wang and Colin McCabe. In-memory caching in HDFS: Lower latency, same great taste. Hadoop Summit, 2014.

[4] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3R: Increased performance for in-memory Hadoop jobs. *Proceedings of the VLDB Endowment*, 2012.

[5] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[7] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[8] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010.

[9] GridGain-Real Time Big Data. *https://gridgaintech.wordpress.com/2013/11/07/hadoop-100x-faster-how-we-did-it/*, 2016.

[10] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Proceedings of the IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, 2010.