Master's Thesis

Cache Affinity-aware In-memory Caching Management for Hadoop

Jaewon Kwak

Department of Computer Science and Engineering

Graduate School of UNIST

2017

Cache Affinity-aware In-memory Caching Management for Hadoop

Jaewon Kwak

Department of Computer Science and Engineering

Graduate School of UNIST

Cache Affinity-aware In-memory Caching Management for Hadoop

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Jaewon Kwak

12. 14. 2016
Approved by

Advisor
Young-Ri Choi

Cache Affinity-aware In-memory Caching Management for Hadoop

Jaewon Kwak

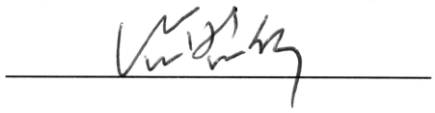
This certifies that the thesis of Jaewon Kwak is approved.

12.14.2016

signature

Advisor: Young-Ri Choi

signature



Beomseok Nam: Thesis Committee Member #1

signature

Woongki Baek: Thesis Committee Member #2

Abstract

In this paper, we investigate techniques to effectively manage HDFS in-memory caching for Hadoop.

We first revisit the current implementation of Hadoop with HDFS in-memory caching to understand its limitation on the effective usage of in-memory caching.

For various representative MapReduce applications, we also evaluate a degree of benefit each application can get from in-memory caching, i.e. cache affinity.

We then propose an adaptive cache local scheduling algorithm that adaptively computes how long a MapReduce job waits to be scheduled on a cache local node to be proportional to the percentage of cached input data for the job.

In addition, we propose a block goodness aware cache replacement algorithm that determines which block is cached and evicted based on the accessed rate and the cache affinity of applications.

Using various workloads consisting of multiple MapReduce applications, we conduct extensive 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.

Contents

List of Figures

List of Tables

List of Algorithms

I. Introduction

The MapReduce programming model makes big-data computation easy. The Apache Hadoop [1] is the one of the popular MapReduce frameworks. In the Hadoop system, usually disk read is the bottleneck. Hadoop adopted in-memory cache system after version 2.3 to reduce disk read overhead.

However, Hadoop file system's in-memory caching has some limitations. First, a user needs to run a command manually for caching files. Second, there is no cache replacement algorithm. The user also needs to run a command for deleting cached files. It would be problematic when the working set size is larger than the in-memory cache size. Even more, there is no cache locality aware scheduler. Cached blocks are used in a probabilistic way.

To solve these problems, we propose an adaptive cache local scheduling algorithm which makes cache local scheduling while it dynamically computes a waiting time to have a cache local task based a cached rate of input, and a block goodness aware cache replacement algorithm which evicts a block with the smallest cache effect by using its cache affinity and accessed rate.

We evaluate the performance of our enhanced Hadoop over various workloads composed of multiple MapReduce applications. Our experimental results show that our enhanced Hadoop in-memory caching scheme can improve the performance of MapReduce applications significantly.

The rest of this paper is organized as follows. Section II describes background and directions for improving Hadoop with HDFS in-memory caching. Section III presents our metrics to reflect the effectiveness of caching a block. Section IV presents our enhanced in-memory caching system. Section V evaluates the performance of our enhanced Hadoop over various workloads. Section VI discusses the related work, and finally Section VII concludes the paper.

II. Background

In this section, we provide background on Hadoop with in-memory caching, and then discuss the opportunity for extending Hadoop with HDFS in-memory caching.

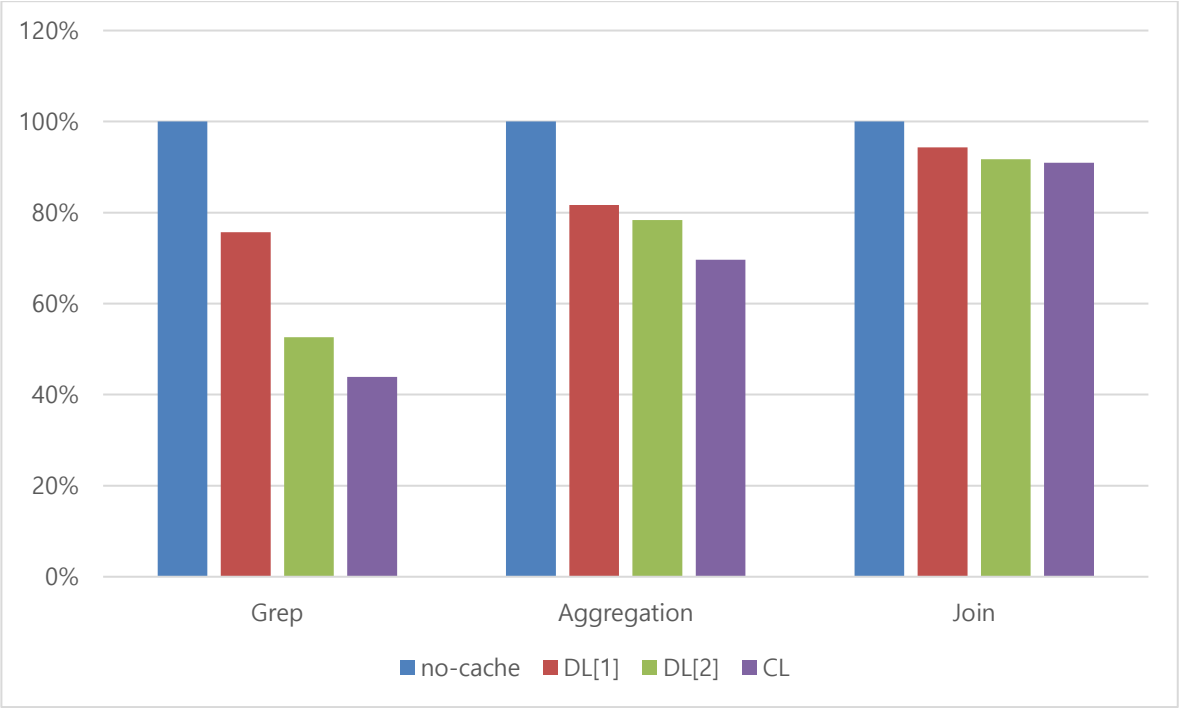


Figure 1. Runtimes over various cache replication factors

2.1 Drawback of Hadoop with In-memory Caching

Centralized cache management in HDFS allows a user to cache their files by sending specific commands. To cache the files, a user needs to send a command with the path to NameNode. Then NameNode translates the path to a set of blocks, and piggybacks the cache request message on a heartbeat to the DataNodes where the data blocks are stored. The DataNode is selected by a lottery algorithm based on remained caching spaces.

The limitation of this default in-memory caching is that a user needs to manually specify files for caching and uncaching it. Before a user uncaches the cached files, the cached files do not uncached until they exceed their expiration time which is specified on the caching command with time-to-live (TTL). When the working set size is larger than the in-memory cache size, this default management

scheme cannot effectively control the cached memory, while Linux OS can manage memory pages by LRU-like page replacement scheme.

Moreover, Yarn [2] which is the default Hadoop scheduler doesn't consider a cache locality. In the current Yarn implementation, the cached data blocks are reused in a probabilistic way. Figure 1 shows the job execution times of three MapReduce applications for the default "data-local" scheduler denoted as DL, as we vary a cache replication factor given in square brackets following the legend when the number of persistent replicas is fixed to 3. The execution times are normalized to those with no-cache which does not utilize in-memory caching. For the experiments, we run each application on 11 nodes (i.e. 1 master and 10 worker nodes) connected via a 10 Gigabit Ethernet switch with total 180 GB in-memory cache, and we increase the cache replication factor up to 2. Also, to study the effect of having the cache locality, all the input data blocks of an application have been loaded into the caches properly before running each experiment.

In this figure, as we increase the number of cached copies for a block, the execution time of each application tends to decrease. When the cache replication is 1, about 30% of tasks read input data files from the caches. Hence, the normalized execution time of the applications decreases by 16% on average, compared to no-cache. As we increase the cache replication factor, more data files can be read from the caches, resulting in performance improvement.

To overcome the drawbacks of current Hadoop scheduling algorithm, we can add another level of data locality – "cache locality". The scheduler first tries to schedule a task on a node which has a cached data block in its memory. This modified scheduler works in the same way as the delay scheduling does except that it first checks cache locality before data locality.

In Figure 1, we evaluate the performance improvement using the modified Hadoop scheduler (denoted as CL) that employs cache local constraints. As in DL[1], the cache replication factor is set to one for CL, but its performance is better than DL[2] in all the applications, because the cache local scheduling effectively leverages the available cached data blocks even if there exists only one copy of a cached data block. It shows if Hadoop scheduler considers the cache locality, caching redundant data blocks across multiple worker nodes becomes unnecessary and the saved in-memory cache spaces can be better utilized by caching a larger number of different data blocks.

2.2 Block Granularity HDFS In-memory Caching

To efficiently utilize HDFS in-memory cache, T. K. Yoo [4] redesigned the HDFS caching mechanism to enable fine-grained block granularity caching, i.e., the unit of caching is a data block, not an entire file. Unlike the explicit HDFS caching mechanism that requires users to specify file

paths to be cached, in fine-grained block granularity caching, a request of caching a data block also can be used. Using this, in our modified caching scheme, a request of caching a data block is initiated by a map task that first accesses the block as input, and the NameNode controls which data blocks should be cached and replaced in the in-memory caches of DataNodes based on the current workloads. Using block granularity caching scheme, multiple competing jobs can utilize the in-memory cache in a fair way as in the OS page cache.

III. Cache Affinity and Block Goodness

In this section, we introduce two cache metrics to manage HDFS in-memory cache more effectively. Cache Affinity (CA) indicates how the MapReduce application can achieve performance improvement using cached blocks, and Block Goodness (BG) reflects the performance improvement by caching a block.

Table 1. Cache affinity of MapReduce applications

Application	Affinity
Aggregation	0.321
Grep	0.638
Join	0.182
KMeans	0.019
PageRank	0.117
Sort	0.134
WordCount	0.087

3.1 Cache Affinity for MapReduce Applications

Each of MapReduce applications achieves different degree of performance benefit by using in-memory caching. The resource usage patterns and composition of multiple jobs for the application affect this effectiveness. For a MapReduce application, the cache affinity indicates how a MapReduce application can achieve performance improvement by caching its input files. And it can be computed as follows:

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

where $R_{no-cache}$ and R_{CL-hot} are the runtimes of the application without caching the input files and after uploading whole input files into cache, respectively as in previous work by J. Kwak et al. [3]. The higher value of this metric means the application can achieve high performance benefit by caching the files. Table 1 shows the computed cache affinity values of the MapReduce applications.

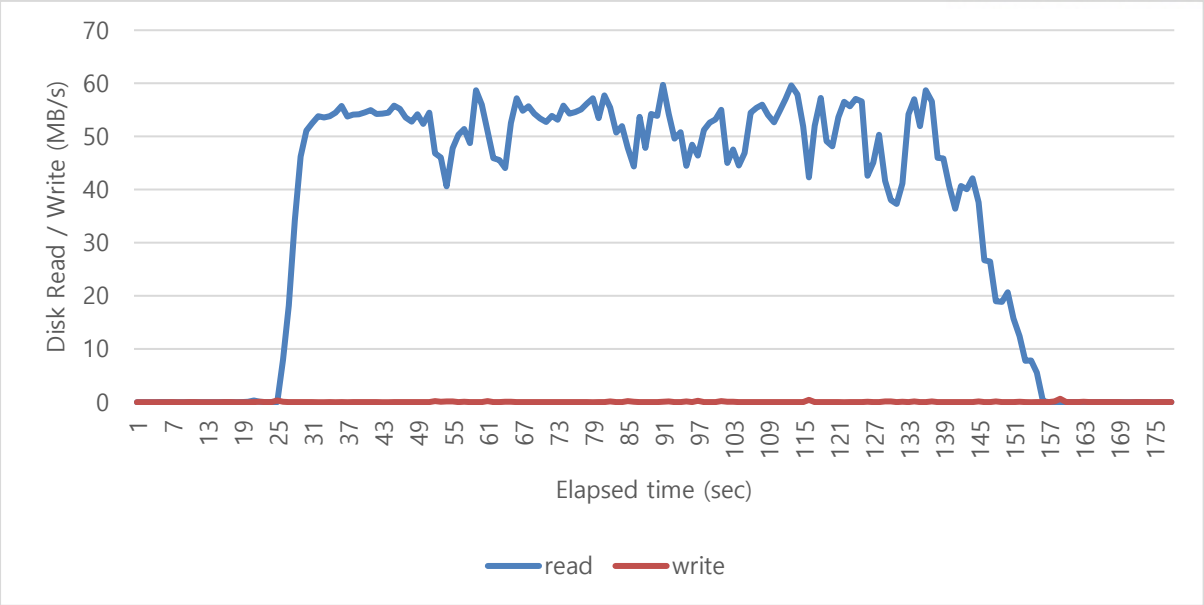


Figure 2. The disk resource usage pattern of grep for cold cache

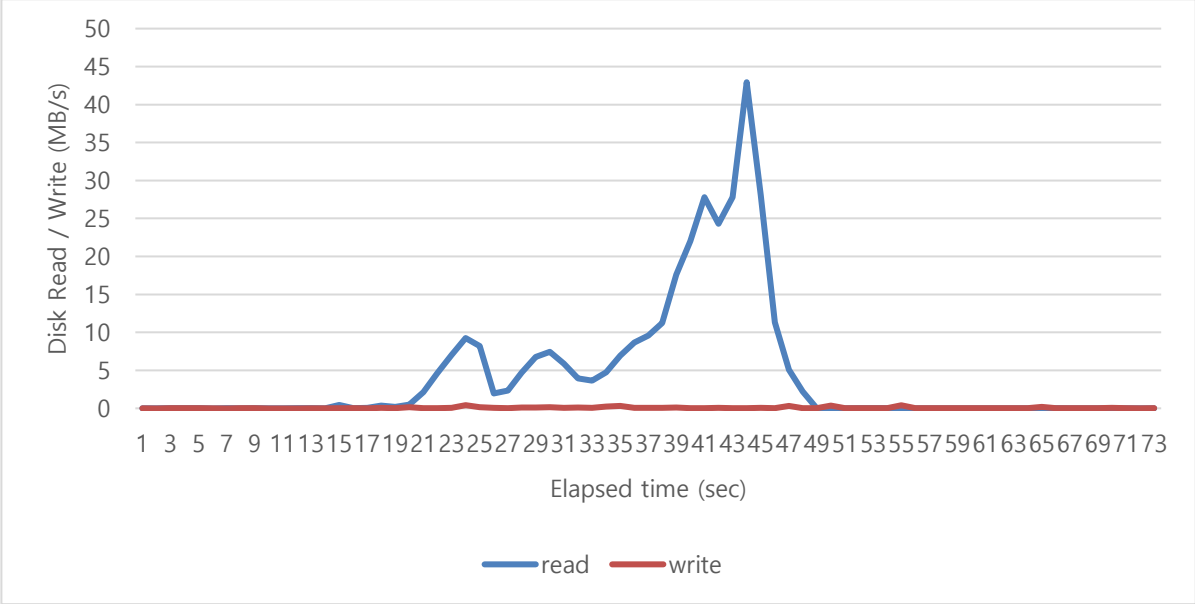


Figure 3. The disk resource usage pattern of grep for hot cache

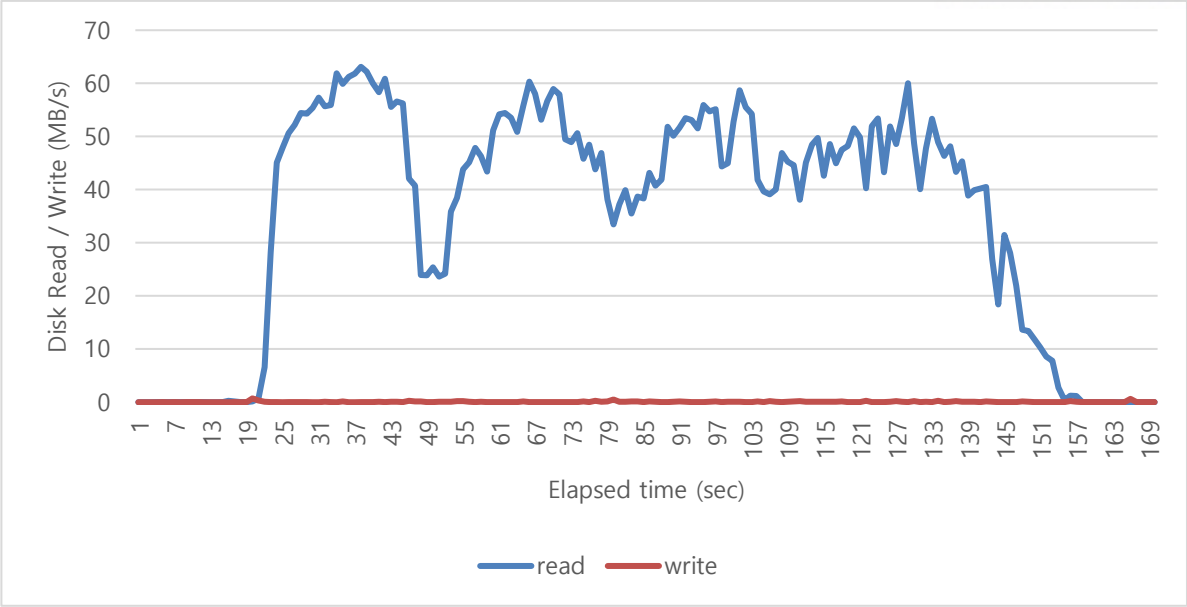


Figure 4. The disk resource usage pattern of WordCount for cold cache

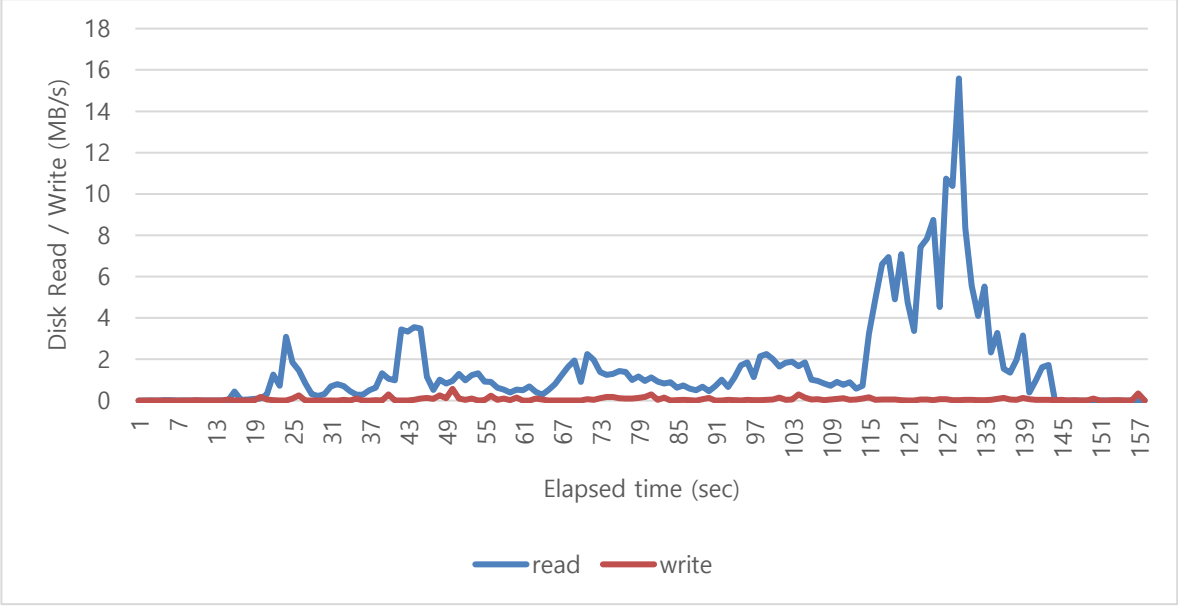


Figure 5. The disk resource usage pattern of WordCount for hot cache

The resource usage pattern greatly affects the CA value. Figure 2 shows the disk resource usage pattern of grep application without caching the input files. Grep application requires high disk read throughput during runtime. We can expect performance improvement by reducing this high disk

resource requirement. Figure 3 shows the disk resource usage pattern of grep application after caching all the input files. Because the total read requirement from disk is dramatically reduced by caching the input files, grep doesn't need such a high disk read throughput. Finally, it was high cache affinity value with 0.638.

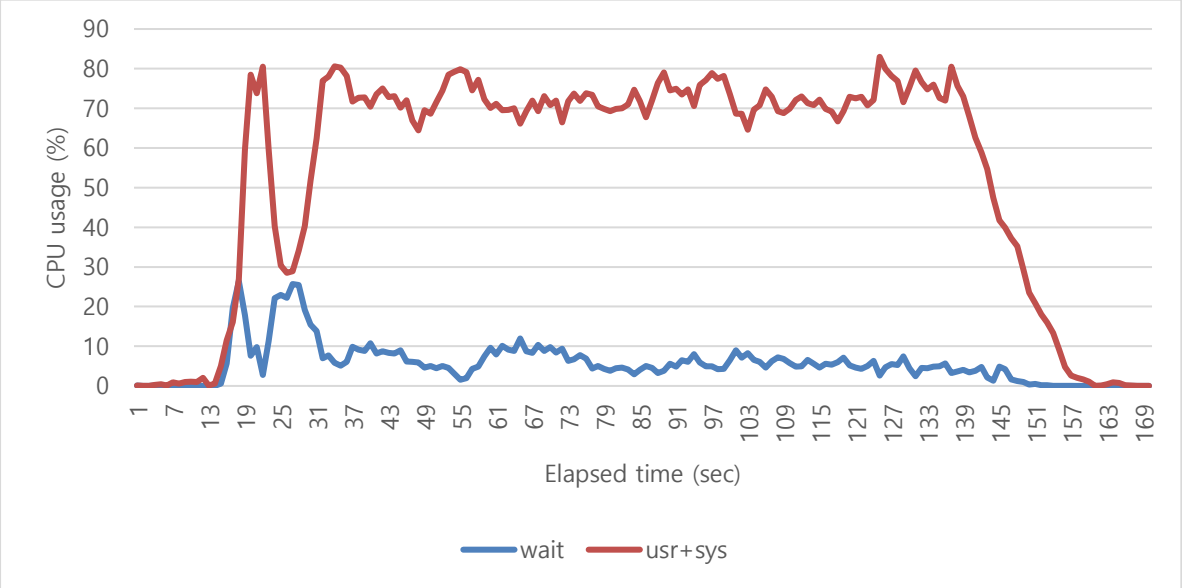


Figure 6. The CPU resource usage pattern of WordCount for cold cache

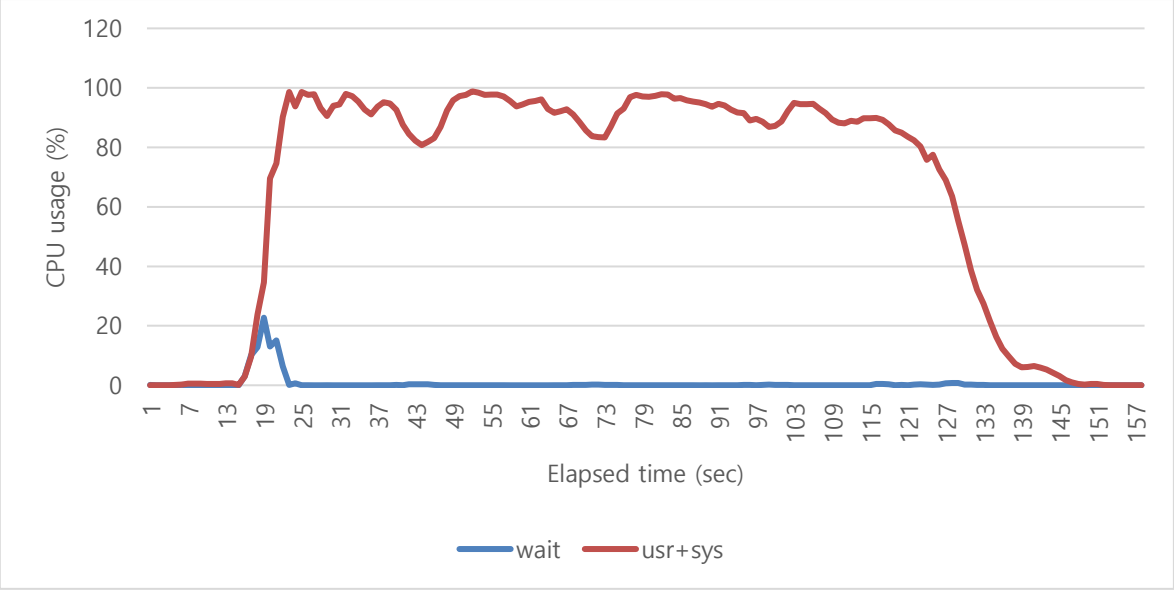


Figure 7. The CPU resource usage pattern of WordCount for hot cache

Figure 4 shows the disk resource usage pattern of WordCount without caching the input files. WordCount shows almost similar high disk read throughput with Grep. Figure 5 shows the disk resource usage pattern of WordCount after caching all the input files. Due to the in-memory cache effects, the requirement of disk read is dramatically reduced and the throughput of disk read also dramatically reduced. But it results only 0.087 cache affinity value. Figure 6 shows the reason of low cache affinity. It shows the CPU resource usage pattern before the input files are cached. The CPU wait time is low on WordCount, but the usr and sys time is dominantly high. Figure 7 shows the CPU resource usage pattern after caching the all input files. Due to the in-memory cache, the CPU wait time is greatly reduced. But the CPU time of usr and sys has almost 100 percent usage. This resource usage pattern shows that CPU is the bottleneck of WordCount application.

Throughout our resource usage pattern analysis, we demonstrate the cache affinity depends on which resource the application uses dominantly.

3.2 Block Goodness for HDFS Blocks

The cache affinity metric does not have the information about which input files are accessed and when they accessed. It also could be a problem when sharing input files. To solve these problems, we propose a new metric, Block Goodness (BG). The Block Goodness metric indicates how much a block is worth when it is cached. It can be computed as follows:

$$in\ a\ \ moving\ window\ duration, \quad \sum_{i}^{AccessedApplications} CA_i \times accessed\_count_i$$

In our experiments, we set the size of moving window to 30 minutes. NameNode records all the accessed events which contain a CA value for each block. The BG value of a block is computed by adding these accessed application counts and cache affinity of them within a recent moving window. For example, a block which is accessed by application A which has 0.5 CA value once and application B which has 0.3 CA value three times within recent 30 minutes has 1.4 BG value. The higher value of BG means the block can achieve high performance benefit from in-memory caching.

IV. Enhanced In-memory Caching System

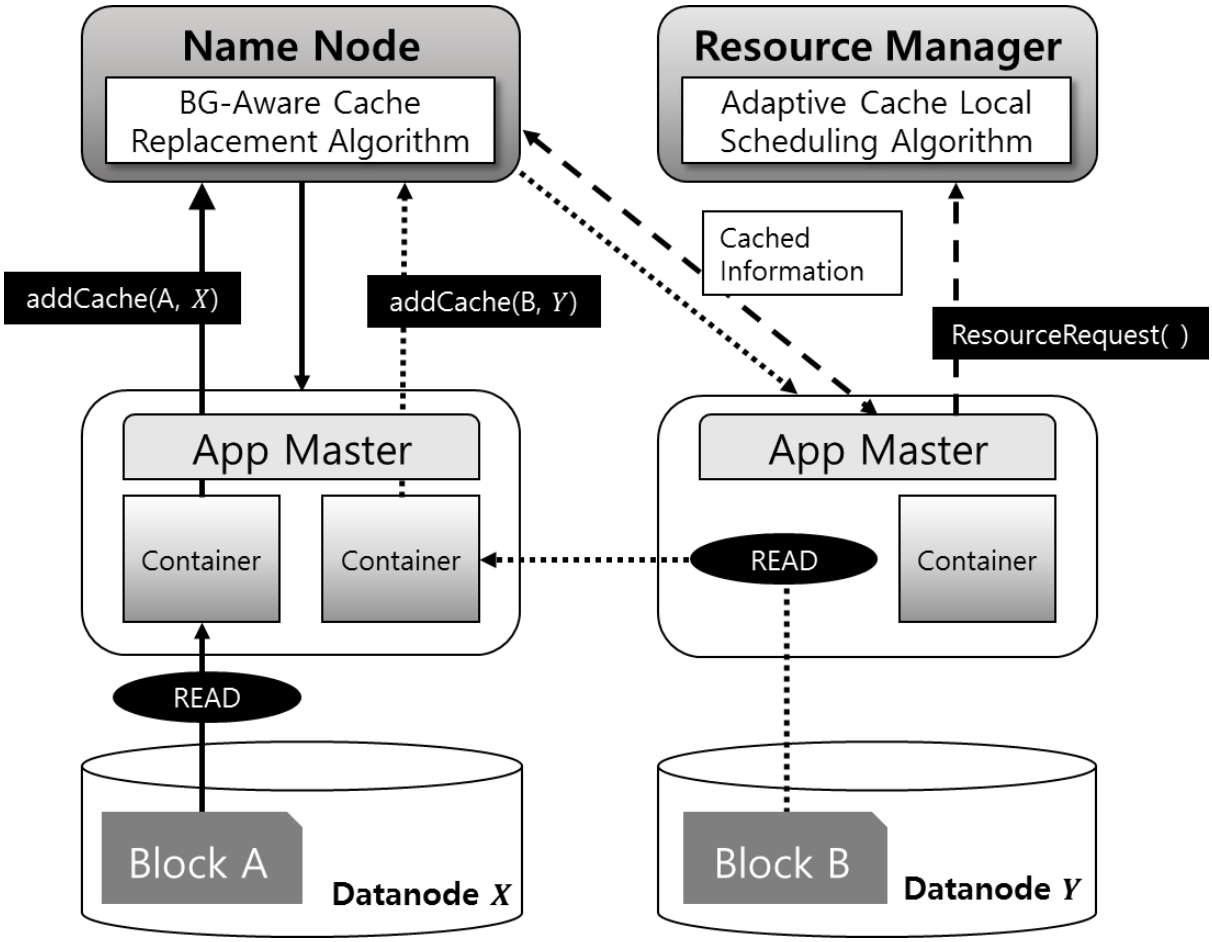


Figure 8. Architecture overview

In this chapter, we propose an our enhanced in-memory caching system which can more effectively utilize HDFS in-memory cache. We first discuss the overall architecture, and then present an adaptive cache local scheduling algorithm, and a block goodness aware cache replacement algorithm.

Figure 8 shows our overall system architecture for Hadoop with HDFS in-memory caching. In our system, a container which is launched to run a map task for a data block always sends a request to cache the block. (Even when the block is already cached, this request is used to update the access time and block goodness value of the block as discussed in detail below.) To reduce caching overhead while executing map tasks of a MapReduce application, we 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. As in the figure, for a data local task of block A on node X, the request to cache A on X (i.e. $addCache(A,X)$) is sent to the NameNode, while for a map task of block B on node X, reading the block from a remote node Y, the request to cache B on Y (i.e. $addCache(B,Y)$) is sent to the NameNode.

To execute map tasks for a MapReduce application, its application master queries the cache status of its input blocks, and sends the cache locality information for its data blocks to the resource manager on requesting resources. The scheduler in the resource manager then tries to first meet cache local constraints for the map tasks, before scheduling them on data-local nodes, based on the information. Also, for each application, the percentage of cached input data (i.e. the percentage of cached blocks over all the input data blocks) is maintained by the NameNode. The NameNode periodically sends the information about the percentage of cached input data of each application to the resource manager, and consequently the resource manager uses this information to make a scheduling decision for the applications.

4.1 Adaptive Cache Local Scheduling Algorithm

Algorithm 1 Adaptive Cache Local Scheduling Algorithm

```
1.  max skip C for cache locality and D for data locality are given as input
2.  initialize j.skip to 0 for each job j.
3.  when a heartbeat is received from node n:
4.  for (i = 1 to MaxAllocations) do
5.      if n has a free slot then
6.          sort jobs in increasing order of number of running tasks
7.          for j in jobs do
8.              get current percentage of cached input data R_j for j
9.              set C_j = R_j × C
10.             if j has a cache-local task t on n then
11.                 launch t on n
12.                 set j.skip = 0
13.             else if j has a node-local task t on n and j.skip > C_j then
14.                 launch t on n
15.                 set j.skip = 0
16.             else if j has unlaunched task t and j.skip > D then
17.                 launch t on n
18.             else
19.                 set j.skip = j.skip + 1
20.             end if
21.         end for
22.     end if
23. end for
```

Algorithm 2 Block Goodness Aware Cache Replacement Algorithm

1. for each application A_i, its cache affinity value CA_i is given as input
2. moving window duration mw is given as input
3. when a request to cache block u, which has block goodness value BG_u, for application A_k on node n is received at time t
4. $u.addAccessedEvent(CA_k, t)$
5. **if** u is already cached in some node **then**
6. return
7. **end if**
8. **if** node n has space to cache u **then**
9. add u to in-memory cache of n
10. **else**
11. **for** cb in cached blocks of node n **do**
12. $cb.removeAccessedEventOutsideMovingWindow(t, mw)$
13. $cb.updateBlockGoodness()$
14. **end for**
15. $u.removeAccessedEventOutsideMovingWindow(t, mw)$
16. $u.updateBlockGoodness()$
17. **if** ∃ block v whose block goodness value is BG_v, where $BG_v < BG_u$, in n **then**
18. compute a set B_{evict} of blocks with lowest BG in n
19. **end if**
20. **if** $B_{evict} = \{\ \}$ **then**
21. return // skip
22. **end if**
23. find a block w with oldest access time in B_{evict}
24. evict block w
25. add u to in-memory cache of n
26. **end if**

Algorithm 1 presents our adaptive cache local scheduling algorithm. (The pseudocode is based on the simple delay scheduling [1].) The maximum numbers of skips to enforce cache and data localities are initially given as C and D, respectively. For a job j, it computes the number of times to skip for achieving the cache locality (C_j) dynamically. The computed C_j is proportional to the current percentage of cached input data of job j. When the percentage of cached input data is low, the probability to launch a cache local task also becomes low. Thus, by reflecting the percentage of cached input data, it is better to reduce the number of times to skips adaptively, resulting in reducing the job scheduling delay for the job. 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.

When a node sends a heartbeat to Resource Manager, it tends to schedule all the containers on the node. If at least one off-rack container is scheduled, the default Hadoop scheduler skips the node.

Because of this, for example, if an application which uses half of the cluster resource are first submitted on the cluster, it scheduled on half of whole cluster nodes. It results in highly unbalanced in-memory cache for the application and low cache hit ratio. Finally, it cannot achieve good cache benefit from in-memory caching due to a low cache hit ratio.

To maximize cache effect, we make the scheduler to schedule containers more balanced. While the default yarn scheduler tries to schedule all containers on a node as it receives a heartbeat from the node, our scheduler schedules at most $MaxAllocations$ containers on the nodes at a time for each MapReduce job. (In this paper, we set this $MaxAllocations$ to 2.) It results in more balanced scheduling between nodes, more balanced cached blocks, and better performance improvement from in-memory cache.

4.2 Block Goodness Aware Cache Replacement Algorithm

Algorithm 2 presents our block goodness aware cache replacement algorithm. When an application reads a data block from HDFS, $addAccessedEvent(CA_k, t)$ function records the event into the block information with CA of the application A_k and access time t. When the replacement algorithm needs to evict a block from a node n, NameNode calculates BG of all the cached blocks of node n. This process consists of following two steps. First, it removes accessed events before the current moving window duration mw. Second, it calculates the BG value of each cached block and updates the value. Using these updated BG, we evict a block which has the lowest BG value. If there are many blocks which has the same lowest BG value, one with the oldest access time will be evicted like LRU policy.

Block goodness has two better effects than cache affinity. The first thing is that block goodness deals with some scenarios in which two or more applications share their input. If a block is shared between two or more applications, it should be assigned a higher value. The second thing is that block goodness gives a higher value to frequently accessed blocks. If a block which is more frequently accessed will have more priority than other cached blocks. Due to these advantages, the block goodness aware cache replacement algorithm can achieve high overall cluster throughput using the in-memory cache.

V. Evaluation

In this section, we analyzed performances of our enhanced Hadoop in-memory caching system.

5.1 Experimental Setup

For our experiments, we use a cluster of 20 nodes. Each node is configured with two Intel Xeon octa-core E5-2650, 32GB memory. These nodes are connected 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 13GB, respectively. The block size of files in HDFS is 128MB, and the replication factor is three. For other Hadoop configuration parameters, the default values are used. The size of in-memory cache is set to 16 GB for each worker node in the cluster, having total 304 GB of HDFS in-memory cache.

Table 2. MapReduce applications used in experiments

Application	Input (GB)
Aggregation	87.4
Grep	121.6
Join	103.7
KMeans	46.5
PageRank	18.1
Sort	88.4
WordCount	121.6

5.2 Applications

Table 2 shows unique input sizes of used MapReduce applications. Aggregation and Join based on HiveQL queries can share part of the input data, and Grep, Sort and WordCount can share the same input data.

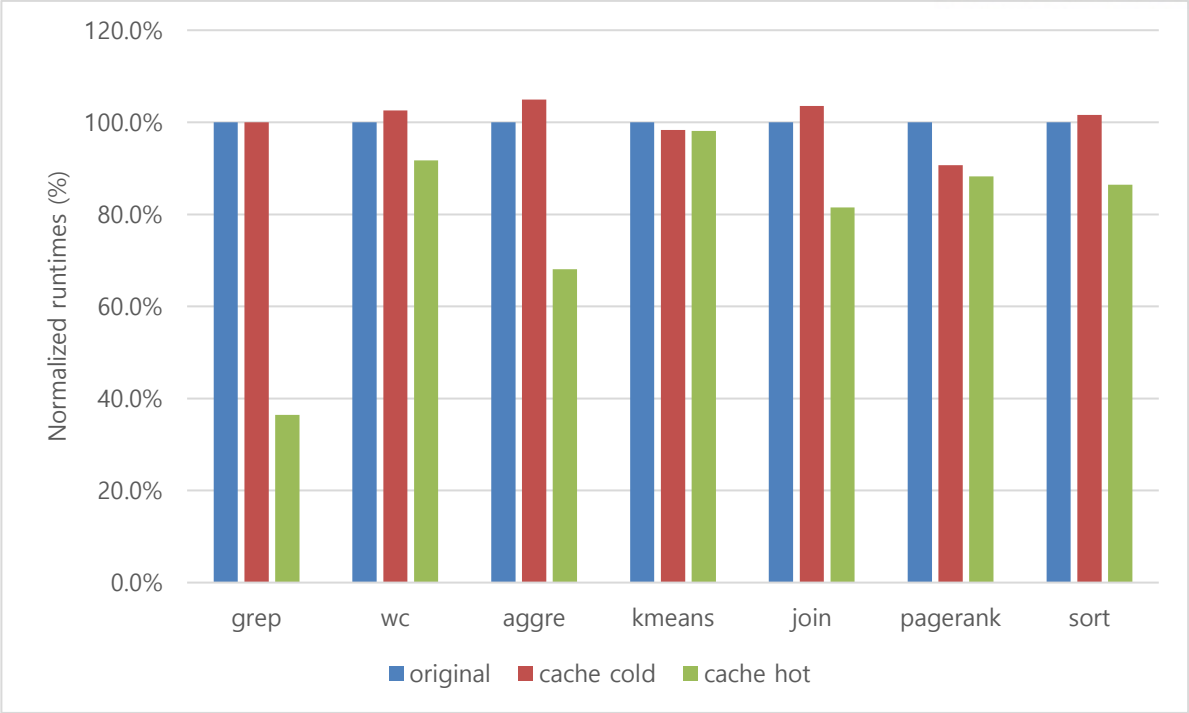


Figure 9. Normalized runtimes of applications

5.3 Single Workload Experimental Results

Figure 9 shows the normalized runtime graph of experiments on single MapReduce applications. In case of cache cold, without any input data loaded into in-memory cache, there are additional overheads by caching the inputs than original one. The overhead is largest for Aggregation with 4.9%, smallest for PageRank with -9.3%.

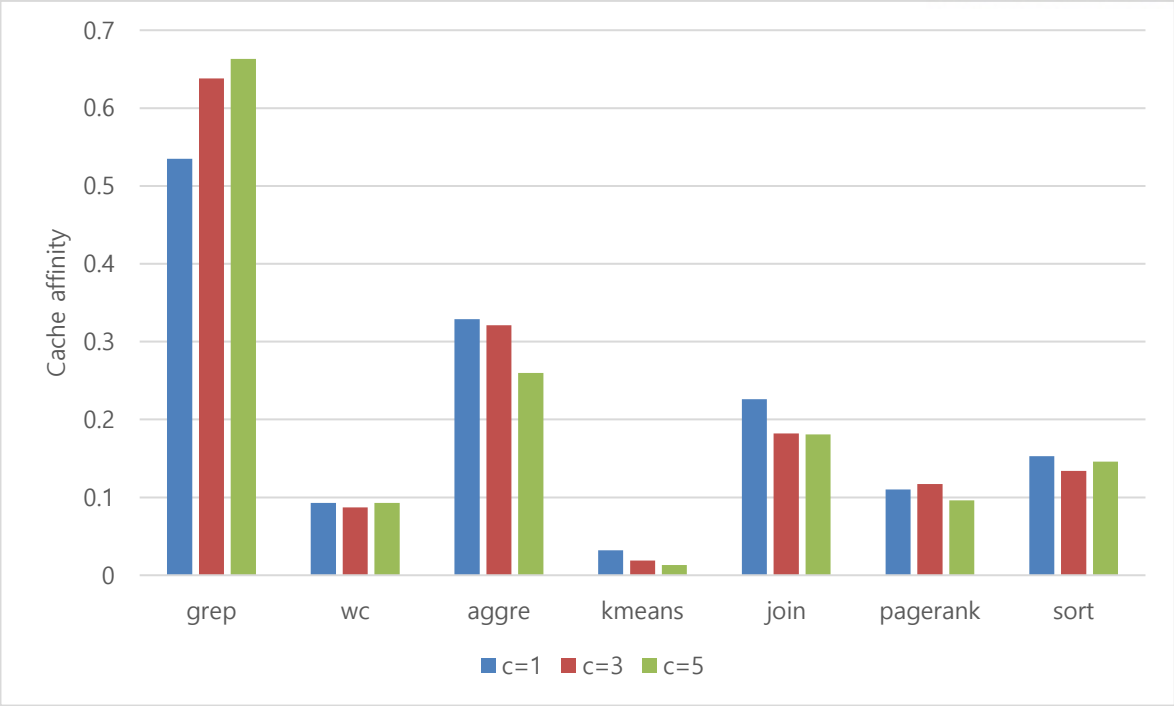


Figure 10. Cache affinities of applications over various C value

Table 3. Workloads composed of multiple MapReduce applications

Workload	App1	App2	App3	App4	# unique inputs	Total size (GB)
W1	Aggre.	W.C.	Join	Grep	3	377.1
W2	Aggre.	W.C.	Grep	Grep	3	377.1
W3	P.R.	W.C.	Aggre.	Grep	4	273.6
W4	Aggre.	W.C.	Sort	Grep	3	377.1
W5	Aggre.	Sort	Grep	Grep	4	452.2
W6	P.R.	Sort	W.C.	Aggre.	3	273.6
W7	KMeans	W.C.	Grep	Join	4	411.3

For adaptive cache local scheduling algorithm, we need to carefully select the C value. For that we did the experiment on 1, 3, and 5 for C. Figure 10 shows the result. If we select small C value, then we can expect low cache hit ratio, but we can get low scheduling overhead. With large C value, we

can expect high cache hit ratio, but we suffer from high scheduling overhead. For the grep application, the cache hit ratio is more dominant than scheduling overhead. On the other hand, aggregation and join shows scheduling overhead is dominant when C is large. Finally, we selected C to 3 on rest experiments.

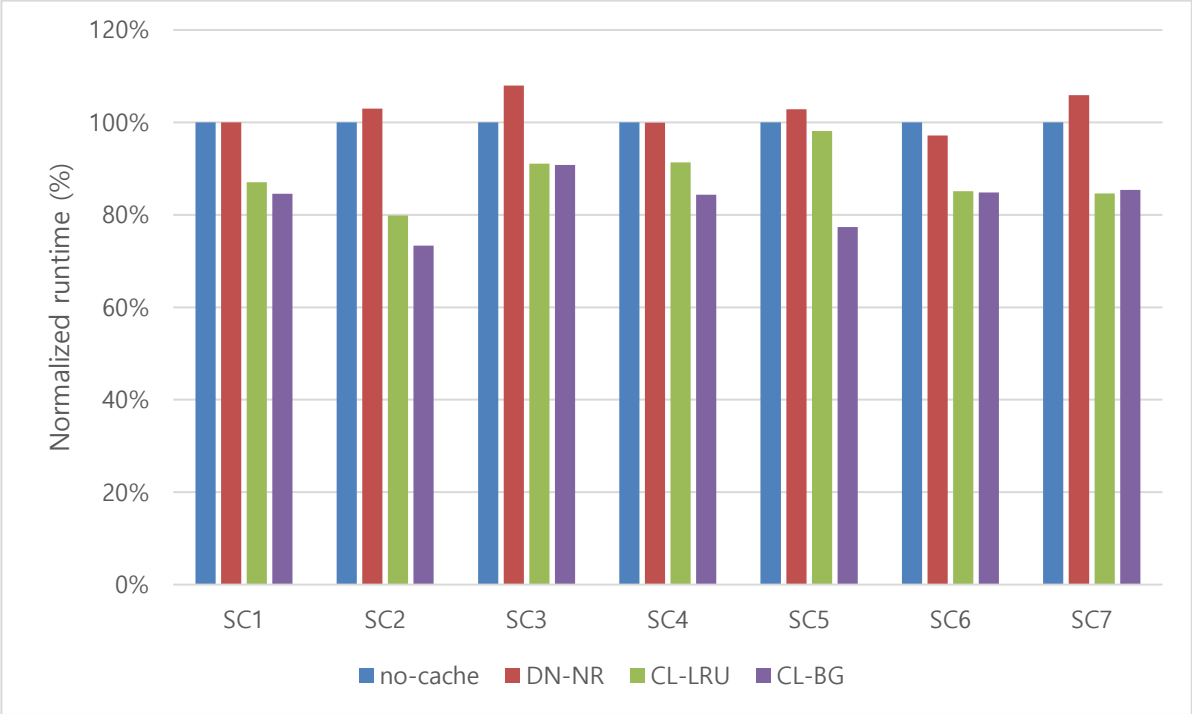


Figure 11. Average normalized runtimes of MapReduce workloads

5.4 Multiple Workload Experimental Results

In this section, we evaluate the performance of our enhanced Hadoop over various workloads where multiple MapReduce applications concurrently run. For each workload composed of four MapReduce applications, each application is allocated 25% of the cluster resources.

Table 3 shows seven workloads of multiple MapReduce applications. The number and size of unique input files for each workload are given in the table. The grayed applications indicate they shares the input files.

We compare the performance of our modified Hadoop CL-BG, which employs fine-grained block granularity caching, adaptive cache local (CL) scheduling, and block goodness (BG) aware cache replacement proposed in previous section, 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 data local scheduling policy. Before running DL-NR, we uploaded the input files. Because there is no cache replacement, some part of input blocks from four applications will not be cached when the total input size is larger than the total in-memory cache size.
- Hadoop CL-LRU: It works exactly in the same way as CL-BG, but it uses traditional LRU policy instead of the block goodness aware cache replacement policy.

Figure 11 shows the overall result of these scenario experiments. DN-NR shows it cannot effectively utilize cached data. Even some cases, DN-NR performs lower than original Hadoop. Because the yarn scheduler on DL-NR does not know about the cached information, the cached data are used in probabilistic way. The original Hadoop can use empty spaces for Linux page cache. Overall, Linux page cache works better than DL-NR. CL-LRU results 11.82% performance improvement than no-cache. CL-LRU achieves 20.14% maximum performance improvement than DL-NR on SC2. While LRU scheme works well on almost scenarios, it achieves only 1.85% performance improvement on SC5. When in-memory cache size is much lower than scenario working set size, LRU cannot properly select the victims for eviction. Using block goodness, we can more effectively select victim blocks for eviction. CL-BG achieves 17.04% performance improvement than DL-NR, and 6.03% than CL-LRU.

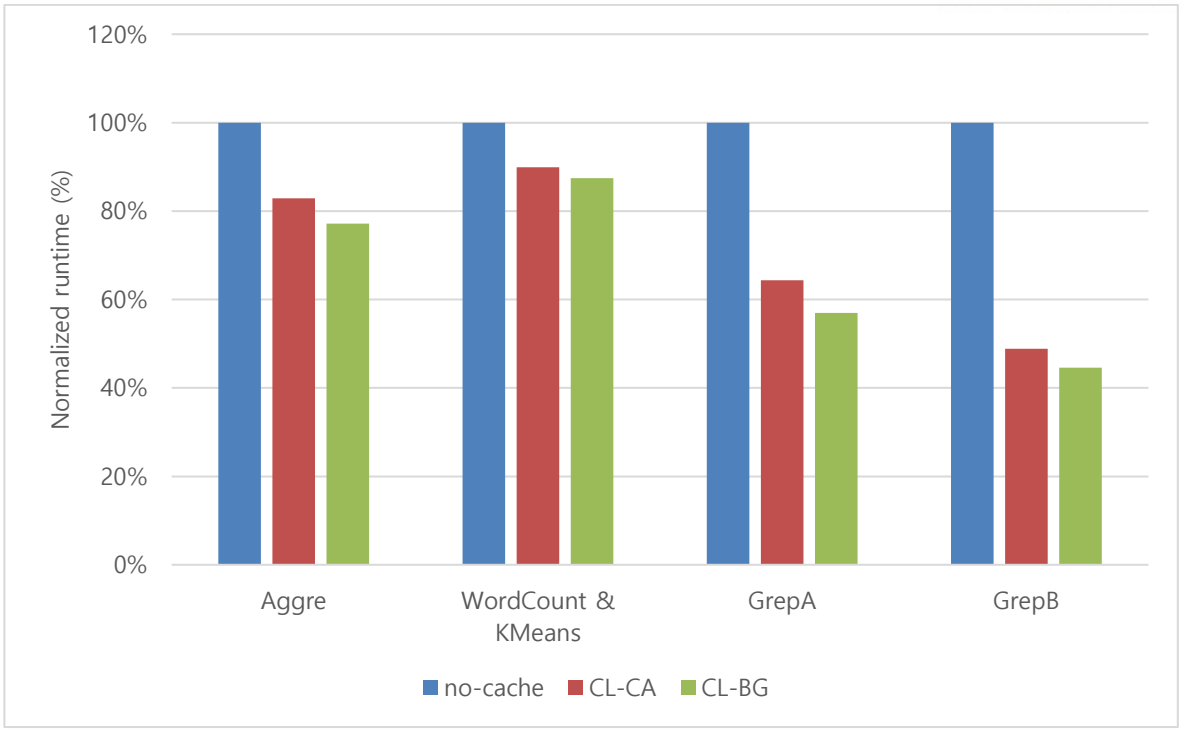


Figure 12. Normalized runtimes of a dynamic workload

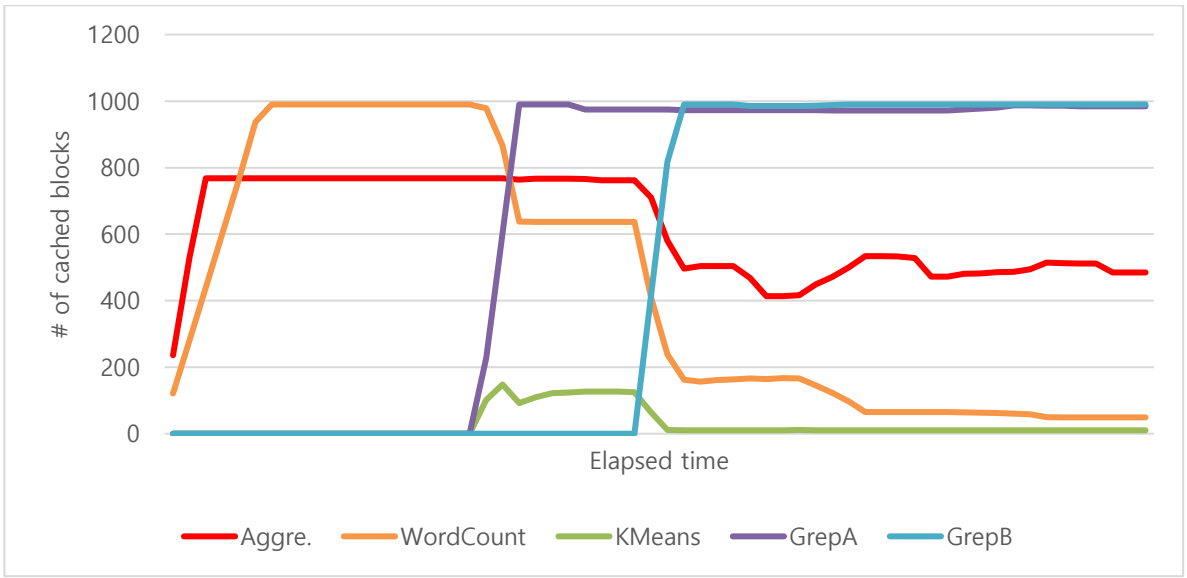


Figure 13. Number of cached blocks over time for each application with CL-CA

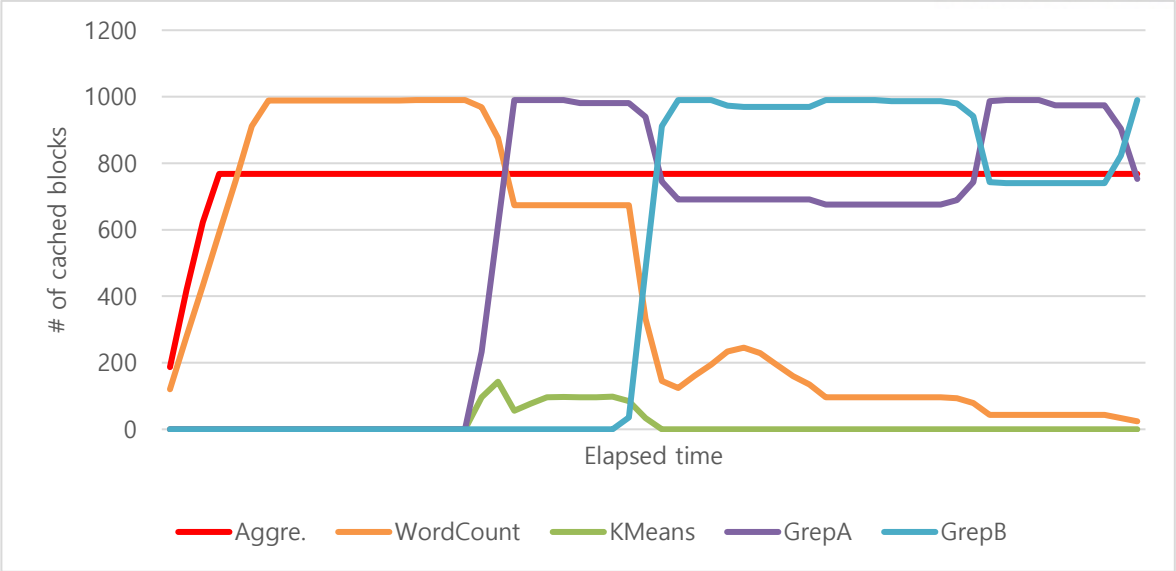


Figure 14. Number of cached blocks over time for each application with CL-BG

5.5 Dynamic Workload Experimental Results

In this section, we evaluate the performance our enhanced Hadoop over a workload where multiple MapReduce applications are dynamically submitted. Four users submit four applications with specific frequency. One of the users submits iteratively WordCount and KMeans submit such that they do not overlap.

Figure 12 shows the results of a dynamic workload experiment. CL-CA employs the adaptive cache local (CL) scheduling and cache affinity (CA) aware cache replacement. In the cache affinity aware cache replacement algorithm, data blocks of applications with low cache affinity are replaced with those with high cache affinity. It does not consider sharing block between applications or accessed frequency, but it has time-to-live (TTL) of the cached block.

In Figure 12, CL-CA and CL-BG shows high performance improvement by in-memory caching than no-cache. CL-BG shows better performance improvement than CL-CA on every application. This gap between CL-BG and CL-CA comes from different frequency between applications. From Figure 13 and 14, we can found the reason of this difference. Figure 13 shows the number of cached blocks over time for each application of CL-CA. In this dynamic scenario, Aggregation has the highest frequency. Because the cache affinity of Aggregation is lower than Grep application, the cached blocks of Aggregation are evicted after grep application submitted and the working set size is larger than in-memory cache size. Figure 14 shows the number of cached blocks over time for each application of

CL-BG. The aggregation keeps its input blocks in the cache, since in the CL-BG, block goodness of Aggregation is much larger than Grep by accessing Aggregation more frequently.

VI. Related Work

There have been earlier efforts to utilize in-memory caching for large-scale data analytics [10, 7, 5, 6, 11]. A distributed cache system, called HDCache, was implemented on HDFS [7]. In HDCache, MapReduce applications require to integrate with a client library to access distributed caches. Spark [6] is a framework to implement Resilient Distributed Datasets (RDDs) [11], which are in-memory data objects. Intermediate results indicated by users can be stored in in-memory cache, and reused by multiple MapReduce applications. In Dache [10], Hadoop was extended to have a cache layer for reusing intermediate results. In ReStore [12], the intermediate and final output of MapReduce jobs is stored in the distributed file system and reused to speed up incoming subsequent jobs. In HaLoop [13], the loop-invariant data is cached and reused for iterative MapReduce jobs to reduce the I/O cost. Main memory Map Reduce (M3R) is a MapReduce framework that employs in-memory for storing intermediate results generated by map tasks, reducing the cost of shuffle, and for storing input and output data [5]. It was proposed that the input data is categorized as static and dynamic data, and since the static data is fixed throughout the whole workflows, it is loaded once and reused to prevent reloading in each iteration [14]. A technique to improve write throughput was also studied by avoiding data replication in HDFS and caching hotspot locally [15, 16]. For data-intensive analytic jobs, several techniques such as Delay scheduling [9] and Quincy [17], have been proposed to consider data locality for improving the performance. While increasing data locality has been considered to be crucial, it was suggested to use in-memory cache for data, and exploit cache locality for efficiency [18].

Like our work, PACMan is a coordinated management system for distributed in-memory caches, aiming to provide memory locality for tasks of a parallel job for performance improvement [8]. Cache eviction policies, which evict data blocks from large incomplete inputs based on “all-or-nothing” property, were investigated for parallel jobs. This approach can be effective especially for short jobs which can run all their tasks in parallel. PACMan allows an unlimited number of cache replicas for a block to increase the probability to achieve cache locality. In our work, by employing a cache local scheduling, we avoid redundant cached blocks, to utilize the saved in-memory cache spaces for other blocks, and reduce on demand caching overhead as discussed in II. Our cache replacement policy makes a decision based on the block goodness which considers cache affinity of MapReduce applications and accessed frequency to utilize in-memory caching effectively for multiple concurrent applications with different characteristics.

There were some efforts to implement block-level caching for Hadoop [19, 20]. BigCache [20] introduces SSD-based caching for big-data systems. For an eviction policy, it does not consider the

characteristics of applications for the effectiveness of using caching. In our work, using the block granularity caching with a small overhead, we developed an efficient cache replacement algorithm based on the block goodness.

VII. Conclusion

In this work, we enhanced Hadoop with in-memory caching via implementing the adaptive cache local scheduling algorithm which tries to enforce cache locality while reducing the scheduling overhead by adaptively computing a skip count, and the block goodness aware cache replacement algorithm that evicts a block which has the lowest block goodness.

Our experimental results show that our enhanced Hadoop in-memory caching scheme improves the performance of static MapReduce workloads up to 26.64% and the performance of a dynamic MapReduce workload by 25.86% against the original Hadoop.

Reference

1. “Apache Hadoop,” Jun. 2011. [Online]. Available: http://hadoop.apache.org
2. V. K. Vavilapalli, “Apache Hadoop YARN: Yet Another Resource Negotiator,” in Proc. SOCC, 2013.
3. J. Kwak, E. Hwang, T. K. Yoo, B. Nam and Y. R. Choi, "In-Memory Caching Orchestration for Hadoop," 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, 2016, pp. 94-97.
4. T. K. Yoo, "Performance analysis of MapReduce with in-memory caching in HDFS", M.S. thesis, Dept. Comput. Eng., UNIST., Ulsan, Korea, 2015.
5. A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat, ‘‘M3R: Increased performance for in-memory Hadoop jobs,’’ in Proc. VLDB Endowment, Aug. 2012, vol. 5, no. 12, pp. 1736–1747
6. M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, “Spark: cluster computing with working sets,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-53, May 2010.
7. J. Zhang, G. Wu, X. Hu and X. Wu, "A Distributed Cache for Hadoop Distributed File System in Real-Time Cloud Services," 2012 ACM/IEEE 13th International Conference on Grid Computing, Beijing, 2012, pp. 12-21.
8. G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In Proc. USENIX NSDI, 2012.
9. M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling,” in EuroSys 2010.
10. Y. Zhao, J. Wu and C. Liu, "Dache: A data aware caching for big-data applications using the MapReduce framework," in Tsinghua Science and Technology, vol. 19, no. 1, pp. 39-50, Feb. 2014.
11. M. Zaharia et al., ‘‘Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,’’ in Proc. 9th USENIX Conf. Netw. Syst. Des. Implement., 2012, p. 2.
12. I. Elghandour and A. Aboulnaga, “Restore: reusing results of mapreduce jobs,” Proc. VLDB Endow., vol. 5, no. 6, pp. 586–597, Feb. 2012.

13. Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, “HaLoop: Efficient Iterative Data Processing on Large Clusters,” VLDB, pp. 285–296, 2010.

14. J.Ekanayake, H.Li, B.Zhang et al., “Twister: A Runtime for iterative MapReduce,” in Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010, Chicago, Illinois, 2010.

15. H. Li, A. Ghodsi, M. Zaharia, E. Baldeschwieler, S. Shenker, and I. Stoica, “Tachyon: Memory throughput i/o for cluster computing frameworks,” memory, vol. 18, p. 1, 2013.

16. H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, “Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks,” in ACM Symposium on Cloud Computing, New York, 2014.

17. M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: Fair Scheduling for Distributed Computing Clusters,” Proc. ACM Symp. Operating System Principles (SOSP ’09), Oct. 2009.

18. G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, “Disklocality in datacenter computing considered irrelevant,” in Proc. USENIX Workshop on Hot Topics in Operating Syst. (HotOS), 2011.

19. “Gridgain-real time big data,” Nov. 2013. [Online]. Available: https://gridgaintech.wordpress.com/2013/11/07/hadoop-100x-faster-how-we-did-it/.

20. M. A. Roger, Y. Xu and M. Zhao, "BigCache for big-data systems," 2014 IEEE International Conference on Big Data (Big Data), Washington, DC, 2014, pp. 189-194.