2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

CAVA: Exploring Memory Locality for Big Data Analytics in Virtualized Clusters

Eunji Hwang*, Hyungoo Kim*, Beomseok Nam† and Young-ri Choi*
*Ulsan National Institute of Science and Technology (UNIST), †Sungkyunkwan University
Email: *{hwangej88,toguk,ychoi}@unist.ac.kr, †bnam@skku.edu

***Abstract*—Running big data analytics frameworks in the cloud is becoming increasingly important, but their resource managers in the current form are not designed to consider virtualized environments. In this work, we investigate various levels of data locality in a virtualized environment, ranging from rack locality to memory locality. Exploiting extra fine-grained levels of data locality in a virtualized environment, our *memory locality-aware* scheduling algorithm effectively increases the cache hit ratio and thereby reduces network traffic and disk I/O. However, a high cache hit ratio does not necessarily imply a shorter job execution time in MapReduce applications. To resolve this issue, we develop the *Cache-Affinity and Virtualization-Aware (CAVA) resource manager*, which measures the cache affinity of MapReduce applications at runtime and efficiently manages distributed in-memory caches of a limited size by assigning high priority to applications that have high cache affinity. The proposed memory locality-aware scheduling algorithm is also integrated into the CAVA resource manager. Our extensive experimental study shows that CAVA exhibits overall good performance over various workloads composed of multiple big data analytics applications by considering the fine-grained data locality levels in virtualized clusters and by efficiently using scarce memory resources.**

I. Introduction

Cloud computing, a cost-effective and elastic computing environment, enables the automatic provisioning of computing resources on demand as the workload dynamically changes. Hence, running big data analytics frameworks in the cloud is becoming increasingly important. Although running Hadoop in a virtualized environment is very common, the current design of Hadoop does not consider a virtualized environment where multiple virtual machines (VMs) share resources such as disks, networks, and main memory.

An important challenge in Hadoop clusters is to reduce network traffic and disk I/O by improving data locality. In legacy Hadoop clusters, three levels of data locality have been considered - *data local*, *rack local*, and *off-rack*. However, although the latest version of Hadoop introduced an in-memory caching feature to avoid the high cost of accessing disks, Hadoop does not consider locality in in-memory caches when it makes scheduling decisions.

When a Hadoop job is executed in a virtualized environment, considering the data locality at the current coarse-grained levels is not sufficient. Imagine a busy virtualized cluster where a data local task is not possible. If there are idle VMs running on the same physical machine as a data local VM, it is better to schedule the task in one of such VMs to avoid network communications across physical machines and disk accesses if the VM can read the cached input block from the data local VM. On the other hand, virtualized environments can degrade performance if VMs on the same physical machine compete for hardware resources. To avoid such resource contention problems in virtualized environments, numerous efforts have been made in the past decade [1], [2], [3].

In this work, we explore extra levels of data locality that take into account the VM topology so that Hadoop can utilize to the greatest extent the resources of physical machines in virtualized environments. In particular, we exploit cached data in the in-memory cache of a co-resident VM on the same physical machine. Suppose VM v_1 running on machine A has input block B for an incoming task in its cache but is not available. If its co-resident VM v_2 on machine A is available, we can assign the task to v_2 so that v_2 can reuse the cached data in the in-memory cache of v_1 via inter-VM communication. Furthermore, to make the most of cached blocks in in-memory cache, if all of the co-resident VMs of v_1 on machine A are busy, we can then assign the task to another VM, v_3, on a different machine but on the same rack as v_1, reusing the cached data in v_1 via inter-node communication.

In-memory cache is still a scarce resource. When multiple MapReduce jobs are concurrently running, the demands of in-memory cache often far surpass the capacity of the in-memory cache. To utilize the in-memory cache in Hadoop clusters efficiently, we must consider the *cache affinity* of MapReduce jobs. Cache affinity between a process and a core has been a subject extensively studied in the context of shared-memory multiprocessors [4], [5]. For a MapReduce job, cache affinity represents the performance improvement effect of in-memory cache hits.

To take advantage of distributed in-memory caches in a virtualized cluster, we develop the *Cache-Affinity and Virtualization-Aware (CAVA) resource manager* for Hadoop. CAVA measures the cache affinity of each MapReduce job at runtime and estimates its performance gain from cache hits. By assigning high priority to jobs that show high cache affinity and access popular input data, scarce in-memory caches can be more efficiently utilized. CAVA also leverages additional levels of data locality by considering the VM topology.

The main contributions of this work are as follows:

- We present how data locality optimization in Hadoop is sub-optimal because the current data locality levels in

0-7695-6410-0/18/$31.00 ©2018 IEEE
DOI 10.1109/CCGRID.2018.00017

Hadoop are ignorant of the VM topology and the cached data blocks in the Hadoop in-memory cache are ignored.

- We introduce more fine-grained virtualization-aware data locality levels for virtualized environments. In particular, the *physical machine (PM) memory locality* schedules a task on a VM u that does not have the input data block of the task in its in-memory cache, but runs together with a VM v that has the data block in its in-memory cache on the same physical machine. The *rack machine (RM) memory locality* schedules a task on a VM w that does not have the input data block in its in-memory cache, running on a different physical machine but on the same rack of VM v. We show that reading data blocks from a co-resident VM's in-memory cache on the same machine or the same rack is faster than reading them from local disks.
- We present a scheduling algorithm that exploits the PM and RM memory localities, effectively increasing the in-memory cache hit ratio and thereby reducing the disk I/O and/or network traffic across physical machines.
- We present an in-memory cache management scheme that applies cache affinity to the MapReduce framework, which assigns tasks to VMs that have more data blocks in their in-memory caches. To determine the cache affinity of applications, we present a simple but effective model that monitors the resource usage patterns at runtime and estimates the performance gain of in-memory cache hits based on the patterns. Our cache affinity-based in-memory cache management scheme requires neither expensive off-line profiling of MapReduce applications nor prior knowledge of them.
- We develop the CAVA resource manager by integrating the proposed scheduling algorithm and in-memory cache management scheme with Hadoop. Our extensive experimental study demonstrates that the CAVA resource manager improves the performance of various dynamic workloads composed of multiple MapReduce applications up to 24% by making full use of the fine-grained data locality levels in virtualized clusters and by efficiently using scarce in-memory resources.

II. CAVA: System Design

First, we present the architecture of CAVA, which undertakes stringent efforts to maximize the reuse of cached data in virtualized environments. Figure 1 shows the overall system architecture of CAVA.

CAVA employs Hadoop NameNode globally to coordinate access to distributed in-memory caches in Hadoop and to manage the topology of VMs in a virtualized cluster. In Hadoop in-memory caching, users request the caching of input blocks of certain files in the OS page cache [6], [7]. The users' requests are sent to the NameNode, which knows the locations of the input data blocks. The NameNode then requests DataNodes to store the data blocks in their OS page caches. The DataNodes periodically inform the NameNode of the data blocks that are stored in their caches.

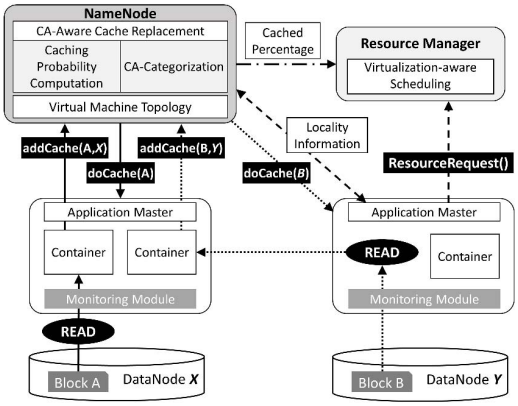


Fig. 1. CAVA Architecture

One of the critical limitations in the current design of Hadoop in-memory caching [7] is that users must explicitly specify what input files need to be cached into in-memory caches. Moreover, the current Hadoop in-memory caching scheme does not dynamically replace cached blocks, unlike OS page cache. To evict the blocks of a file from the in-memory caches, a user must explicitly run a command to uncache the file.

For better utilization of the large distributed in-memory caches in Hadoop clusters, we have each Hadoop container always send a request to cache the accessed block to the NameNode, and then have the NameNode control which data blocks are added and evicted to and from in-memory caches. We also have the DataNode that recently accessed the data block store the block in its in-memory cache. If the DataNode has the data block in its OS page cache, we call `mmap()` and `mlock()` to pin the block in its cache. This approach significantly reduces the caching overhead. Note that the original Hadoop does not consider which DataNode has the requested data block in its OS page cache, but it has the DataNode with the largest available cache space store the data block probabilistically.

Figure 1 indicates that two map tasks in DataNode X access one data block from local disks and another data block from the disks of remote server Y. Each of these tasks sends a request to cache one of the blocks. One request to cache data block A on DataNode X (i.e. $addCache(A, X)$) is sent to the NameNode, while for the other task, the request to cache B on Y is sent to the NameNode. The cache replacement algorithm in CAVA then determines whether or not we need to store them in in-memory caches, and if it decides to cache them, CAVA asks DataNode X to cache A (i.e. $doCache(A)$) and DataNode Y to cache B. Note that although both tasks run on DataNode X, B is cached in DataNode Y so that we can reuse the OS page cache.

Another modification to the NameNode realized here is the *cache affinity categorization* module. The cache affinity of each application indicates the benefit gained by each application by reusing the cached data. The NodeManager manages the cache affinity of each application using categories containing high, medium, and low values. Based on the

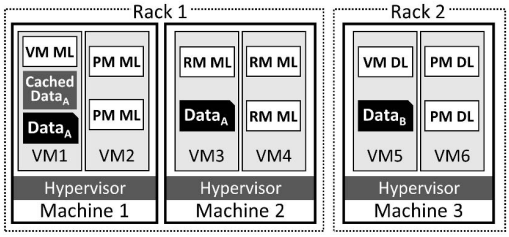


Fig. 2. Locality of input data in a virtualized cluster

resource usage patterns of applications, CAVA classifies the applications into high, medium, and low cache affinity groups at runtime.

We also added the *caching probability computation* module in the NameNode, which estimates how much of the in-memory caches each application can use for its input blocks. CAVA makes cache replacement decisions according to the cache affinity of each application, the size of the input files, the frequency of access of the files, and the size of the in-memory cache. That is, even if an application has a very high cache affinity level, if the size of its input data blocks exceeds the available in-memory caches, it becomes impossible to cache all of the blocks. Therefore, CAVA computes the probability of caching each input file and attempts to cache each file probabilistically, thus reducing the caching overhead.

In summary, when a MapReduce job is scheduled by CAVA, the ApplicationMaster communicates with the NameNode to query the locations of the input blocks, their availability in in-memory caches, and the VM topology. It then requests resources for the tasks of the job from the ResourceManager, which makes scheduling decisions based on the memory and disk locality information of the blocks in the virtualized cluster. While running the tasks in VMs, CAVA estimates their cache affinity at runtime and determines whether or not the input blocks need to be cached.

III. Virtualization-Aware Data Locality

A. Locality of Input Data in Virtualized Clusters

Figure 2 illustrates various levels of data locality in virtualized environments. The legacy Hadoop offers several levels of data locality. Hadoop attempts to schedule a task in a node that has its input data block (*data locality*). If data local nodes are not available, Hadoop runs the task on a different node but on the same rack as the node which has the data (*rack locality*). If rack local nodes are all busy, which is the least preferred case, Hadoop runs the task in a node on a remote rack (*off-rack*).

These legacy coarse-grained data locality levels do not account for the virtualized environments or the in-memory caches. In a virtualized cluster, the VM topology must be considered for an accurate determination of the data locality before we attempt the next level locality.

Suppose we replicated a data block of $Data_A$ in two VMs, `VM1` and `VM3`, and $Data_A$ is cached in `VM1`. When scheduling a task that accesses $Data_A$, the ideal case is to run the task in `VM1` so that it can reuse the cached data. To distinguish this ideal case from the legacy data locality, we refer to this locality level as *virtual machine memory locality (VM ML)*.

If VM ML is not possible, i.e., `VM1` is not available, we can fall back to the legacy data locality and schedule the task in `VM3` so that we can read $Data_A$ from local disks. To classify the degrees of data locality into more fine-grained levels, we refer to this locality level as *virtual machine disk locality (VM DL)*. However, we claim that `VM3` is not a better choice than `VM2`. That is, if we let the co-resident VMs of `VM1` on the same physical machine read $Data_A$ from `VM1`'s in-memory cache, we can avoid expensive disk I/O. We refer to this locality level as *physical machine memory locality (PM ML)*. Moreover, if we let the co-resident VMs of `VM1` on the same rack (i.e., `VM3` and `VM4`) access the cached data block on `VM1`, we can also avoid disk I/O despite the fact that $Data_A$ is transferred via inter-node communication. We refer to this locality level as *rack machine memory locality (RM ML)*. As we will show in Section III-B, data transfers between co-resident VMs on the same physical machine or the same rack are often faster than local disk accesses because the network communication between VMs on the same machine has been optimized and does not use any network bandwidth [1]. Moreover, the disk I/O performance can become slower when numerous VMs are executed on the same machine and must compete for disk bandwidth.

In a busy cluster, a case can arise in which no instances of VM ML, PM ML, RM ML or VM DL are possible. In such a case, it is better to check if the co-resident VMs of VMs with VM DL on the same physical machine are available (`VM6` for $Data_B$ in the example). If the task runs on the co-resident VMs, it can read the data block from local disks and avoid network communication. We refer to this locality as *physical machine disk locality (PM DL)*. Note that if all of the above localities are not available, we attempt to use VMs running on the same rack of VMs with VM DL, and then off-rack VMs.

B. Overhead of Accessing the Co-resident VM's In-Memory Cache

To evaluate how much of a performance improvement we can achieve with more fine-grained virtualization-aware locality levels, we measured the execution times of various representative MapReduce applications of Grep, Wordcount, Sort, Aggregation, Join, Selection, and Pagerank. We ran the experiments on one or two physical machines, each of which has 32 cores and 128 GB of memory and uses one 7.2K RPM HDD for running VMs. The size of the input data for each of the applications is around 8GB, except for Pagerank whose input size is around 800 MB.

On a testbed machine, we created eight worker VMs with three virtual CPUs each. Each VM is configured to run a NodeManager and a DataNode. A worker VM is configured to execute up to two map tasks concurrently. We used Hadoop with the in-memory caching feature [7] in the experiments. The execution times of the legacy data local scheduling policy, denoted as VM DL, are used as a base line. To ensure that every VM has all input blocks in its local disks, we set the

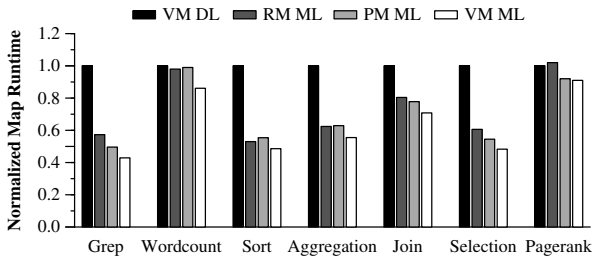


Fig. 3. VM DL vs. RM ML vs. PM ML vs. VM ML

HDFS replication factor to 8. The ideal case is when all VMs have all input blocks in their local in-memory caches, denoted as VM ML. To measure the performance of VM ML, we set the HDFS replication factor to 8 and have Hadoop instances in all VMs cache the input blocks; the cache replication factor is also 8. Because each VM has all input blocks in its local disks and its in-memory cache as well, regardless of where a task is scheduled, it does not access disks but reads an input block from the in-memory cache. Note that these experiments disregard the effects of scheduling delays because each task can be assigned to any free slot, as all VMs have the input data blocks on a local disk or in the in-memory cache.

Figure 3 shows that the normalized total execution times of map tasks with VM ML runs against VM DL runs vary across the applications. In the experiments, we have reduce tasks start only after all map tasks are completed. Wordcount and Pagerank gain relatively small performance improvements by reusing data as they are computation intensive, but other applications save up to 57.1% of the total map task execution time at the cost of excessive memory usage, i.e., redundant cached data in every VM.

Unlike VM ML, PM ML allows tasks to benefit from a co-resident VM's in-memory cache so that we can better utilize in-memory caches. To measure the performance of PM ML, we separate a NodeManager and a DataNode of a worker VM above into two VMs, one NodeManager VM with two virtual CPUs and one DataNode VM with one virtual CPU, and we set both of the HDFS and cache replication factors to 8. Thus, each DataNode has all of the input blocks of a file in its in-memory cache as well as on its disks, but map tasks running on each VM with the NodeManager should read input blocks from the in-memory cache of one of the DataNode VMs.

With regard to RM ML, tasks reuse the input blocks cached in the in-memory cache of a co-resident VM on the same rack, increasing the cache hit ratio. To measure the performance of RM ML, we use two physical machines, A and B. Each machine has a setup identical to that in the PM ML experiment. In this case, NodeManager VMs on a machine A use the HDFS running on DataNode VMs in the different machine B. Thus, for map tasks of a job, the NodeManager VMs on A should access cached blocks from the DataNode VMs on B. Note that we configure another Hadoop cluster which runs on 8 NodeManager VMs on B and 8 DataNode VMs on A to execute another job concurrently, with a contention level on each machine similar to those in the other experiments.

Figure 3 shows that the total execution time of map tasks with PM ML is slightly slower than that with VM ML for all applications and that the execution time with RM ML is slightly slower or similar to that with PM ML. On average, VM, PM, and RM memory localities improve the performance by 36.7%, 29.8% and 26.6% compared to VM disk locality for map tasks, respectively. This result implies that the time required to access the in-memory cache of a co-resident VM on the same physical machine and rack is slightly longer than the time required to access a local in-memory cache.

The default HDFS replication factor is 3. However, the number of co-resident VMs is often greater than that. Suppose a physical cluster is composed of N PMs and each PM runs V VMs ($N \times V$ VMs in total). The probability of scheduling a task in a VM DL node is only $3/(N \times V)$, but the probabilities of scheduling a task in a PM ML node and a RM ML node are $(V-1)/(N \times V)$ and $(N-1)/N$, respectively, when the cache replication factor is 1. Therefore, in most of cases, the expected waiting times for PM ML and RM ML are much shorter than that of VM DL.

Algorithm 1 Memory Locality-aware Scheduling Algorithm

```
1: input: M_VM, // max skip count for VM memory locality
2:        M_PM, // max skip count for PM memory locality
3:        M_RM, // max skip count for RM memory locality
4:        D_VM, // max skip count for VM disk locality
5:        D_PM // max skip count for PM disk locality
6: initialize j.skip to 0 for each job j;
7: when a heartbeat is received from VM n:
8: if n has a free slot then
9:    sort jobs in increasing order of number of running tasks;
10:   for j in jobs do
11:      get current percentage of cached input data R_j for j;
12:      if j has a VM ML task t on n then
13:         launch t on n; set j.skip = 0;
14:      else if j has a PM ML task t on n and j.skip ≥ R_j × M_VM
         then
15:         launch t on n; set j.skip = 0;
16:      else if j has a RM ML task t on n and j.skip ≥ R_j × M_PM
         then
17:         launch t on n; set j.skip = 0;
18:      else if j has a VM DL task t on n and j.skip ≥ R_j × M_RM
         then
19:         launch t on n; set j.skip = 0;
20:      else if j has a PM DL task t on n and j.skip ≥ D_VM then
21:         launch t on n; set j.skip = 0;
22:      else if j has unlaunched task t and j.skip ≥ D_PM then
23:         launch t on n;
24:      else
25:         set j.skip = j.skip + 1;
26:      end if
27:   end for
28: end if
```

C. Memory Locality-aware Scheduling

Algorithm 1 presents our memory locality-aware scheduling algorithm. Using more fine-grained data locality levels does not always guarantee available computing resources. Therefore, our scheduling algorithm extends a well-known delay scheduling scheme [8]. For a job j, the scheduler initially attempts to run a task on a VM memory local node. If the VM memory local node is busy, it then tries to launch the task on a PM memory local node. If none of the PM memory local nodes are available, it attempts to schedule the task on

a RM memory local node. If none of the RM memory local nodes are available, it attempts to run the task on VM disk local, PM disk local, and then rack local nodes in that order.

As in delay scheduling, our algorithm attempts to schedule a given task on higher level local (i.e., faster) nodes several times before attempting the next level local nodes. The number of retries for each job and each locality level is dynamically determined such that the numbers of retires for VM ML, PM ML and RM ML are proportional to the current percentage of cached input blocks for job j.

If the percentage of cached input data for j is low, the chance to achieve VM ML, PM ML, and RM ML levels is low. Thus, by considering the percentage of cached input blocks, we can reduce the number of retries adaptively for VM ML, PM ML, and RM ML, and consequently reduce the job scheduling delay for the job. In CAVA, for each application, the NameNode maintains the percentage of cached input data in the distributed in-memory caches and periodically sends the information to the ResourceManager.

Note that the priority of RM ML might be changed for a virtualized cluster where disk I/O contention between VMs or network performance of VMs is low.

D. VM Topology

The VM topology is available in private clouds and virtualized clusters. However, this information is mostly unknown in public clouds. On Amazon EC2, unless we use dedicated hosts and place VMs on a specific host [9], information about which VMs are running together on the same host is not known. If this is the case, CAVA must generate the topology of VMs using the traceroute technique at each VM to other VMs in the cluster and checking the first hop, as was done in earlier work [1], [10]. To find VMs running on the same rack as well as those on the same host, a topology management system which measures data transfer rates and estimates the distances between VMs to compute the VM topology can be used [11].

IV. Cache Affinity-Aware In-memory Cache Management

A. Runtime Cache Affinity Categorization

The cache affinity of a MapReduce application represents the performance gain of the application by reusing input data cached in in-memory caches [12]. By reading input blocks from in-memory caches without disk access, the execution time of a map task can be reduced, but the effect of in-memory cache hits on the execution time of map tasks varies widely depending on the MapReduce applications used, as shown in Figure 3.

To estimate the cache affinity, we can use off-line profiling, as was done in an earlier study [12], to compute the ratio of the job execution time for each run with and without in-memory cache hits. However, such an off-line profiling method requires the running of the same application at least twice, one with hot cache and the other without cache. Furthermore, off-line profiling must be performed on each of the clusters with different physical system configurations, as the cache affinity of an application may vary if the physical configuration of a cluster changes. Considering that big data applications are long-running tasks, the off-line profiling methods are not very practical.

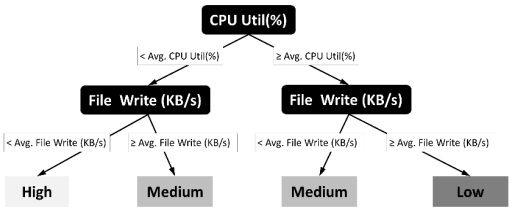


Fig. 4. Decision Tree

TABLE I
Resource usage patterns of single applications

Application	CPU Util (%)	File Write (KB/s)	CA
Grep	7.03	4.00	High
Wordcount	51.47	5.89	Medium
Sort	16.68	2885.42	Medium
Aggregation	26.04	169.36	High
Join	29.88	345.77	Medium
Selection	18.48	5.77	High
Pagerank	35.56	1756.41	Low
Average	26.45	738.95	

Instead, we propose a novel dynamic cache affinity categorization method that estimates the cache affinity of applications by analyzing their resource usage patterns at runtime. The algorithm is simple to the point that it allows the use of a decision tree. We observe that the applications realizing a substantial performance improvement by reusing cached input data often have low CPU utilization, generate small intermediate results, and thereby do not require high network bandwidth in the shuffle phase. That is, disk read throughput is the only potential performance bottleneck.

The monitoring module collects the resource utilization information of all concurrent applications and categorizes the applications into high, medium, and low cache affinity groups using the decision tree shown in Figure 4. When an application starts, we measure the resource utilization of the tasks for a short amount of time. The monitoring module at each worker VM uses the Linux tool "`pidstat`" and Hadoop performance metrics to monitor resource usage patterns, including CPU utilization and I/O write traffic amounts.

Because we collect resource utilization information at runtime, the cache affinity of a MapReduce application can change depending on what other applications are concurrently running. That is, the same application can be classified into the low cache affinity group or the high cache affinity group depending on the concurrent workload.

Table I shows the average CPU utilization and I/O write traffic, as measured during the map phase, for the representative MapReduce applications (used in our experiments in Section V). It also shows the cache affinity level of each application, which is categorized using the decision tree shown in Figure 4. For the experiments, we run each application in a MapReduce cluster that consists of 104 VMs across four

physical machines. More details about the cluster configuration are described in Section V-A. For an application composed of multiple MapReduce jobs such as Join, we measure the resource usage pattern of the map tasks only in the first job. For the results, the resource usage pattern of each application was measured by executing it without any co-running applications in the cluster and was then used for the decision tree.

The cache affinity categorization can be reinvoked if any change is made to the workload. For example, if a new application is submitted, we monitor the resource usage patterns again and change the cache affinity groups if necessary. Thus, the CAVA resource manager always uses the most recently computed cache affinity values of applications.

It should be noted that the decision tree is used only when most of input blocks for an application are not yet available in the in-memory cache. If a large fraction of input blocks is already cached, CPU utilization of the application is likely to be high regardless of the cache affinity of the application, as disk reads are not needed for input blocks. Therefore, if an application finds that a large fraction of its input blocks is already cached, the monitoring module only checks if an application incurs more frequent disk writes than average. If so, we consider it as write I/O bound and categorize it into the low cache affinity group. Otherwise, we consider it is in the high cache affinity group.

Additionally, if we find that an application has been frequently submitted to a cluster in the past, the resource usage patterns measured during the previous executions can be used to estimate the cache affinity level of the application as in earlier research [13] to avoid monitoring overhead.

B. Cache Affinity Aware Cache Replacement Algorithm

Benefit of Caching Data In data analytics platforms, it has been reported that some data is far more frequently accessed, i.e., *hot data* [14], [15], [16]. For such data, it is generally advantageous to cache the data even if the estimated cache affinity is low, as the data can be accessed later by multiple applications, including those that exhibit high cache affinity levels. Therefore, a cache replacement algorithm must take into account both the cache affinity of applications and the access frequency of the input data so that we can effectively manage scarce cache resources for multiple jobs.

In this work, we define the *benefit* of caching input data D for application A_i as follows,

$$CA_i \times FREQ(D)$$

where CA_i is the cache affinity value of application A_i and $FREQ(D)$ is the number of accesses to input data D during the previous sliding time window. The caching benefit of application A_i increases proportionally to the frequency of access to D. Note that the size of the sliding time window should be large enough such that hot data can be accessed many times by various applications within the window.

When input data are commonly shared by multiple applications, as is commonly the case in data analytics, the frequency of access to certain data is increased by multiple applications. If the multiple applications have different cache affinity values, the performance gain achievable by reusing the cached data can vary among applications. In CAVA, we set the benefit of caching an input file as the access frequency multiplied by the highest cache affinity value of the applications that used the file. That is, if an application with high cache affinity and an application with low cache affinity share the same input file, the caching benefit of the latter application is considered to be as high to match that of the former application.

However, it is possible for a file to be accessed by a high cache affinity application only once and later to be accessed only by low cache affinity applications. In such a case, the caching benefit of the low cache affinity applications should be degraded, because the performance gain by caching the file in the in-memory cache becomes small. Therefore, each time the sliding window moves, we reset the cache affinity value to the highest value of the applications that accessed the file during the previous sliding time window.

Caching Probability Computation Recall that the in-memory cache in a MapReduce cluster is still a scarce resource, as the demands of in-memory cache from multiple MapReduce jobs often far surpass the capacity of the in-memory cache in the cluster. Hence, the CAVA resource manager determines which application will use what portion of the in-memory cache for its input blocks according to the caching benefit of the applications.

To compute the probability of caching input blocks for each application, we take into account the cache replacement policy, which assigns priority to the high caching benefit. When multiple applications are concurrently running, we sort them in decreasing order of their caching benefit. For each application in that order, CAVA determines how much in-memory cache space can be used. Particularly, CAVA determines the size of the in-memory cache for each application based on three parameters: the total in-memory cache size, the total size of the input blocks that are already cached by applications with the same or higher benefit but not used by the current workload, and the estimated size of the in-memory cache that will be used by applications with higher benefit of the current workload.

If the estimated in-memory cache size that can be used by a particular application is larger than or equal to the size of its input data, the caching probability is 100%. Otherwise, we basically compute the caching probability by dividing the estimated cache size by the size of its input data. When two or more applications have the same caching benefit and the in-memory cache size that can be used for them is smaller than the total size of their input files, we let the applications use an equal size of the in-memory cache. If the same file is used by several applications, it becomes necessary to count the file only once during the above estimation process. Note that CAVA recomputes the caching probability when any change to the workload is made and also periodically recomputes it to exclude cached input blocks whose benefit is zero.

Replacement Algorithm Our cache affinity aware cache replacement algorithm evicts the input data blocks of applica-

tions that have a lower caching benefit value and caches input data blocks of applications that have a higher benefit value. For a request to cache an input data block by application A_i at VM n, if the block is already cached in the in-memory cache, the algorithm only needs to update the access time.

For an uncached data block, we probabilistically attempt to cache the block based on the caching probability of A_i, i.e., P_i. Input blocks of an application with a low P_i are highly likely to be evicted before they are reused, even if they are added to the in-memory cache. Thus, those blocks will be added to the in-memory cache with a low probability. This can help reduce the overhead of caching and replacing blocks.

If the algorithm determines to cache the requested block on VM n, but n has no free space, it computes which block can be evicted from the in-memory cache of n. It initially determines whether there is any cached block that has not been accessed during the previous sliding time window (i.e., with a zero benefit value). If there is no such block, it finds which block has the lowest caching benefit value on n. If all cached blocks on n have benefit values which are higher than or equal to the benefit value of the requested block, the request to cache the block will be dropped. Among the candidate cached blocks for eviction either with a zero benefit value or with the lowest cache affinity value, we can select one of the blocks as a victim according to the least recently used (LRU) policy. Note that in our algorithm, a data block will not be evicted by another data block from the same application, and only one copy of a data block is allowed to be cached in the in-memory cache so as to cache a large number of different data blocks for multiple MapReduce jobs.

V. Evaluation

A. Methodology

In our experiments, we use a cluster that consists of four physical machines connected via one GE switch. Each of the machines has four Intel Xeon Octa-core E5-4610v2 processors and 128 GB of memory. For each machine, one 7.2K RPM HDD is used for VMs. The Xen hypervisor (version 4.5.2) is installed on each machine, and a para-virtualized Linux with kernel version 3.19.0-25-generic is installed on the guest operating system for VMs.

A virtualized MapReduce cluster used in our study is composed of 104 VMs, a master VM that runs the ResourceManager and NameNode, and 103 worker VMs, each of which runs a NodeManager and a DataNode. The number of VMs running on each of the physical machines is 26. The master VM is configured with two virtual CPUs and 6 GB of memory, while each worker VM is configured with two virtual CPUs and 3.8 GB of memory. In each worker VM, 2 GB of memory is assigned to a NodeManager, and up to 1.5 GB of memory can be used for the in-memory cache. Thus, the total size of the in-memory cache in the MapReduce cluster is 154.5 GB. For map and reduce tasks, 2 GB of memory is configured to run each of them. For HDFS, the block size is 64 MB, and there are three replicas. We extended Hadoop 2.4.1 to implement the CAVA resource manager [17].

TABLE II
MapReduce applications and input files used in experiments

File Type	MapReduce Applications	File Size (GB)
Text	Grep, Sort, Wordcount	40.7 $\sim$ 87.3
DB	Aggregation, Select, Join	43.0 $\sim$ 67.3
Web Graph	Pagerank	8.7

TABLE III
Workloads used in our experiments

Workload	Cache Affinity	Hot Data	Total Input (GB)	Cross Sharing
W1	CA-H	HD-H	264.3	○
W2	CA-H	HD-H	362.6	×
W3	CA-H	HD-L	425.1	×
W4	CA-M	HD-H	321.8	○
W5	CA-M	HD-L	347.9	○
W6	CA-M	HD-L	437.5	×

Table II shows the seven MapReduce applications used in our experiments. For each application of Grep, Aggregation, Selection, and Join, there are two types, while for Wordcount, there are four. Each type uses a different condition to process input data. In the case of Wordcount, each type counts either the occurrence or length of words in an input file. Aggregation, Join, and Selection are generated by HiveQL queries (as in [18]). This table also shows the file type processible by each application. Applications with the same file type can share input files. For each file type, we generate some number of files with several different sizes. The range of file sizes used is also given in the table.

For the experiments, each workload consists of 30-40 applications randomly selected from Table II. For each of the selected applications, an input file is also randomly chosen from among the files which can be processed by the application. For each experiment, we start concurrently to execute six MapReduce applications of a given workload without any input data added into the in-memory cache. If one of the applications is done, we execute the next application, and this is done until we run all of the applications in the workload. Thus, one-sixth of the MapReduce cluster resources are used to execute each of the applications. Note that we assigned a value which is larger than the total execution time of any workload to the size of the sliding time window.

Table III shows the six dynamic workloads used in our experiments. Depending on the percentage of high cache affinity applications, each workload is sorted into the CA-H or CA-M category. For a CA-H workload, we generate a workload such that applications with high, medium, and low cache affinity levels correspondingly have 70%, 20%, and 10% selection probabilities, while for a CA-M workload, we generate a workload such that applications with high, medium, and low cache affinity levels likewise have 20%, 70%, and 10% selection probabilities.

Depending on the percentage of hot data, each workload is categorized as HD-H or HD-L. If the percentage of hot data exceeds 50%, a workload is HD-H. Otherwise, it is HD-L. In the experiments, we consider input data as hot data if it

is accessed more than three times during the previous sliding time window. The total size of input data for each workload is also given in the table. For all workloads, the total input file sizes exceed the total in-memory cache size of the cluster. In addition, for W1, W4, and W5, we allow applications with different cache affinity categorizations to share input files, whereas for W2, W3, and W6, we have applications with the same cache affinity categorization share input files. Note that we configured most of the workloads to have relatively large hot data files, as it was reported that in real world workloads, the size of a hot file is usually large [14].

We evaluate the performance of the proposed resource manager (CAVA), which computes the cache affinity of each application at runtime; considers VM, PM and RM memory localities; and replaces blocks based on the benefit of caching, with computation based on the cache affinity of applications and the frequency at which input files are accessed. We also compare its performance with those of the following Hadoop versions.

- Hadoop No-Cache: This baseline Hadoop does not use the in-memory cache when running MapReduce applications, and only considers VM disk locality on scheduling tasks.
- CA-Only: This behaves in precisely the same manner as CAVA except that it does not consider how frequently input files are accessed when computing the caching benefit.
- Static-CA: This behaves identically to CA-Only except that the cache affinities of MapReduce applications are not measured at runtime, and off-line profiled cache affinity values are given as the input [12]. The cache affinity of an application is computed as $MAX(1 - \frac{R}{R_{No-Cache}}, 0)$, where $R_{No-Cache}$ is the runtime of the application with No-Cache and R is its runtime with our memory locality-aware scheduling when all of the input blocks of a job have been cached in the in-memory cache. The cache affinity values of Grep, Wordcount, Sort, Aggregation, Join, Selection, and Pagerank are 0.377, 0.198, 0.146, 0.133, 0.073, 0.237, and 0.041, respectively.
- CAVA-LRU: This works exactly in the same manner as CAVA, except that it uses the traditional least recently used (LRU) policy instead of the cache affinity aware cache replacement policy.

For No-Cache, the maximum skip count for VM DL, D_{VM}, is set to 103 (which is the number of worker VMs in our Hadoop cluster). For CAVA, CA-Only, Static-CA, and CAVA-LRU, M_{VM}, M_{PM}, M_{RM}, D_{VM} and D_{PM} are set to 1, 3, 6, 52, and 103, respectively. In our setup, when a data block is cached in a VM, the probabilities of scheduling a task in a VM with PM ML and RM ML are high, and the performance of PM ML and RM ML is comparable to that of VM ML. Thus, M_{VM}, M_{PM}, and M_{RM} were experimentally tuned but assigned small values. For CAVA and CA-Only, the resource usage patterns are monitored for a small fraction of map tasks (i.e., 50 map tasks in our experiments) for applications to classify their cache affinity levels at runtime, unlike Static-CA.

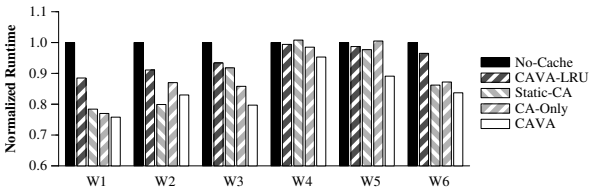


Fig. 5. Normalized total runtimes of six workloads

TABLE IV
PERFORMANCE IMPROVEMENT OF MAPREDUCE APPLICATIONS

Application	Average	Maximum
Grep	33%	64%
Wordcount	13%	38%
Aggregation	17%	46%
Join	15%	37%
Select	19%	49%

B. Experimental Results

Figure 5 shows the total runtime of each workload, normalized to that with No-Cache. Compared to No-Cache, the performance improvements of CAVA-LRU, Static-CA, CA-Only and CAVA for the six workloads are 5.4%, 10.9%, 10.7% and 15.6% on average, respectively. From these results, it is apparent that CAVA shows good performance for all the workloads compared to the other Hadoop versions, and dynamic cache affinity categorization does not degrade the performance.

The performances of CAVA, CA-Only and Static-CA can be similar when high cache affinity applications frequently access hot data. For all three versions, in W1, hot input files are cached and then accessed frequently by high cache affinity applications, resulting in good performance. However, CA-Only and Static-CA do not consider the data access frequency when making a replacement decision. If a hot file is continually accessed by a medium cache affinity application while a cold file is accessed by a high cache affinity application, as in W3-W5, CA-Only and Static-CA cache the blocks of the cold file instead of those of the hot file in the in-memory cache, resulting in lower performance. For W2, Static-CA starts to cache an input file used by high cache affinity applications from the beginning, while CAVA gradually caches the file as its frequency increases. Thus, the performance of Static-CA is slightly higher than that of CAVA. CAVA-LRU generally performs poorly compared to CAVA, as CAVA-LRU simply evicts the least recently used input block without considering the possible performance gain by reusing the block.

Table IV presents the average and maximum performance improvements of each application by CAVA compared to those of No-Cache. In these results, we compute the normalized average runtime of each application over all runs executed in the six workloads, including the runs for which no input data were added to the in-memory cache. There is a considerable

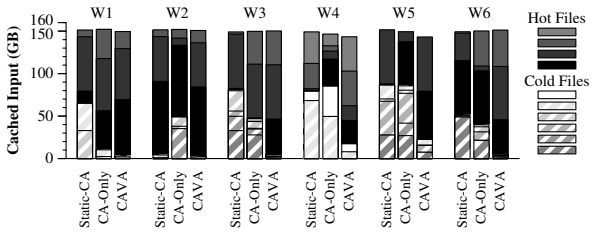
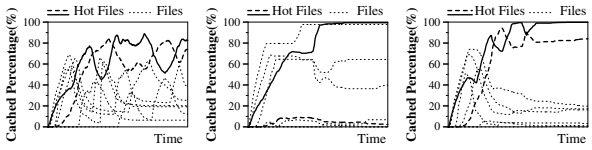


Fig. 6. The amount of cached input data



(a) CAVA-LRU (b) Static-CA (c) CAVA

Fig. 7. The amount of input data cached in in-memory cache over time

performance improvement for not only high cache affinity applications, but also for medium cache affinity applications, as the overall disk I/O performance bottleneck is mitigated by effectively reusing cached data in the in-memory cache and input data accessed and cached by high cache affinity applications can be reused by medium or low cache affinity applications.

Figure 6 presents the final cached status of hot and cold files when all applications in a workload are completed for Static-CA, CA-Only and CAVA. The percentages of hot data cached in the in-memory cache with Static-CA, CA-Only and CAVA are 50.6%, 56.1% and 81.2%, respectively. With Static-CA and CA-Only, in W3-W5, a large amount of cold files accessed by a high cache affinity application is cached in in-memory cache, unlike CAVA. In the case of W6, the amount of cold data cached by CA-Only and Static-CA is relatively small when compared with W5, which has the same workload categorization. Thus, CA-Only and Static-CA perform better in W6 than in W5. Note that with Static-CA, in W1, an application with the highest cache affinity value accesses a new file and evicts some of hot data at the end of the workload, with its final amount of cached hot data becoming smaller than in the other cases.

Figure 7 shows the amounts of cached data for two hot files and other cold files over time during the execution of W5. In the figure, CAVA-LRU continues to add and evict hot blocks to and from the in-memory cache, incurring high overhead. In the case of Static-CA, one of the hot files is not cached, as the off-line profiled cache affinity value of the file is not high and the frequency is not considered. For CAVA, it caches both hot files which show a high caching benefit.

VI. Related Work

Several techniques to improve the performance of MapReduce applications in the cloud have been studied [1], [19], [20]. Techniques to achieve data locality of input blocks when executing tasks have also been investigated [19], [20]. In a virtualized cluster, another level of data locality where a task runs on a co-resident VM that has a block in its local storage (referred to as PM DL in our work) is leveraged upon task scheduling, but the locality of input data in the in-memory caches is not considered [1].

The legacy data locality levels are not sufficient for Hadoop with the in-memory caching feature. To mitigate this problem, cache locality, which considers which node has cached data in main memory, has been investigated in a recent study [12]. However, additional levels of data locality in a virtualized environment, such as PM ML, are not supported during scheduling tasks. In the same work, a cache replacement scheme based on cache affinity, similar to our algorithm, was also studied. However, the cache affinity of each MapReduce application is computed based on off-line profiling, which is not practical (as discussed in Section IV-A), and a value of the probability of caching blocks for the application should be input by users.

For data-intensive analytic jobs, several techniques such as Quincy [21], have been proposed to consider data locality for improved performance outcomes. While increasing the data locality has been considered to be crucial, it was suggested to use in-memory cache for data, and exploit cache locality for efficiency [22].

PACMan is a coordinated management system for distributed in-memory caches, the goal of which is to provide memory locality for tasks of a parallel job for performance improvements [14]. Cache eviction policies, which evict data blocks from large incomplete inputs based on the "all-or-nothing" property, were investigated for parallel jobs. This approach can be effective, especially for short jobs which can run all of their tasks in parallel. PACMan allows an unlimited number of cache replicas for a block to increase the probability of achieving memory locality. In our work, with fine-grained memory locality-aware scheduling, we avoid redundant cached blocks and utilize the saved in-memory cache spaces for other blocks, and our cache replacement algorithm can reach decisions based on the cache affinity of MapReduce applications to utilize scarce in-memory cache resources effectively.

There were extensive earlier efforts to utilize in-memory caching for large-scale data analytics [23], [24], [25], [26], [27]. A distributed cache system known as HDCache was implemented on HDFS [24]. In HDCache, MapReduce applications must be integrated with a client library to access distributed caches. Spark [26] is a framework which implements resilient distributed datasets (RDDs) [27], 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 [23], Hadoop is extended to have a cache layer to reuse intermediate results. In ReStore [28], the intermediate and final outputs of MapReduce jobs are stored in a distributed file system and reused to speed up incoming subsequent jobs. In HaLoop [29], loop-invariant data is cached and reused for iterative MapReduce jobs in order to reduce the I/O cost. Main memory Map Reduce (M3R) is a MapReduce framework that employs in-memory

caches to store intermediate results generated by map tasks, reducing the cost of shuffling, while also storing input and output data [25]. In Twister [30], the input data is categorized as static and dynamic data, and because the static data is fixed throughout all workflows, it is loaded once and reused to prevent reloading for each iteration. A technique to improve the write throughput by avoiding data replication in HDFS and caching hotspot locally was also studied [31].

VII. Concluding Remarks

In this work, we explored various levels of locality of input data in a virtualized environment for big data analytics frameworks. We designed and implemented the Cache-Affinity and Virtualization-Aware (CAVA) resource manager. In CAVA, memory locality-aware scheduling is used to leverage extra levels of data locality in virtualized clusters. CAVA also categorizes the cache affinity of MapReduce applications at runtime without any prior knowledge of the applications and efficiently manages in-memory caches of a limited size for multiple MapReduce applications based on the cache affinity. Analyzing the effects of RM ML over different hardware settings for network and storage and also over different VM configurations is considered as our future work.

Acknowledgment

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

References

[1] X. Bu, J. Rao, and C.-z. Xu, "Interference and locality-aware task scheduling for MapReduce applications in virtual clusters," in *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, 2013.

[2] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for QoS-aware clouds," in *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[3] J. Han, S. Jeon, Y.-r. Choi, and J. Huh, "Interference management for distributed parallel applications in consolidated clusters," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[4] V. Kazempour, A. Fedorova, and P. Alagheband, "Performance implications of cache affinity on multicore processors," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, 2008.

[5] M. S. Squillante and E. D. Lazowska, "Using processor-cache affinity information in shared-memory multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[6] A. Wang and C. McCabe, "In-memory caching in HDFS: Lower latency, same great taste," in *Hadoop Summit*, 2014.

[7] "Apache Hadoop centralized cache management in HDFS." *http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html*.

[8] 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 *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[9] "Amazon EC2 dedicated hosts." *https://aws.amazon.com/ec2/dedicated-hosts/*.

[10] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[11] M. Li, D. Subhraveti, A. R. Butt, A. Khasymski, and P. Sarkar, "CAM: A topology aware minimum cost flow based resource manager for MapReduce applications in the cloud," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, 2012.

[12] J. Kwak, E. Hwang, T.-k. Yoo, B. Nam, and Y.-r. Choi, "In-memory caching orchestration for Hadoop," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2016.

[13] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[14] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[15] J. Liang, J. Luo, M. Drayton, R. Nishtala, R. Liu, N. Hammer, J. Taylor, and B. Jia, "Storage and performance optimization of long tail key access in a social network," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, 2013.

[16] R. T. Kaushik and M. Bhandarkar, "GreenHDFS: Towards an energy-conserving, storage-efficient, hybrid Hadoop compute cluster," in *Proceedings of the USENIX annual technical conference*, 2010.

[17] "Apache Hadoop." *http://hadoop.apache.org/*.

[18] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2009.

[19] J. Park, D. Lee, B. Kim, J. Huh, and S. Maeng, "Locality-aware dynamic VM reconfiguration on MapReduce clouds," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, 2012.

[20] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for MapReduce in a cloud," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[21] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[22] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Disk-locality in datacenter computing considered irrelevant," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[23] Y. Zhao and J. Wu, "Dache: A data aware caching for big-data applications using the MapReduce framework," in *Proceedings of INFOCOM*, 2013.

[24] J. Zhang, G. Wu, X. Hu, and X. Wu, "A distributed cache for Hadoop distributed file system in real-time cloud services," in *13th International Conference on Grid Computing*, 2012.

[25] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat, "M3R: Increased performance for in-memory Hadoop jobs," *Proceedings of the VLDB Endowment*, 2012.

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2010 USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[28] I. Elghandour and A. Aboulnaga, "ReStore: Reusing results of MapReduce jobs," *Proceedings of the VLDB Endowment*, 2012.

[29] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, 2010.

[30] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.

[31] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014.