JOB-DRIVEN SCHEDULING FOR VIRTUAL MAPREDUCE CLUSTERS

P.Umarani, Dr. J.C.Miraclin Joyce Pamila M.E., Ph.D.,
PG scholar, Assistant Professor (Sr.Grade)
Department of Computer Science &Engg
Government College of Technology, Cbe
pumarani90@gmail.com

Abstract

Virtual private servers (VPSs) rented from VPS provider is cost-efficient for a tenant with a limited budget to establish a virtual MapReduce cluster. To provide an appropriate scheduling scheme for this type of computing environment, we propose in this paper a job-driven scheduling scheme (JoSS) from a tenant's perspective. JoSS provides not only job level scheduling, but also map-task level scheduling and reduce-task level scheduling. JoSS classifies MapReduce jobs based on job scale and job type and designs an appropriate scheduling policy to schedule each class of jobs. The goal is to improve data locality for both map tasks and reduce tasks, avoid job starvation, and improve job execution performance. Two variations of JoSS are further introduced to separately achieve a better map-data locality and a faster task assignment. Extensive experiments are conducted to evaluate and compare the two variations with current scheduling algorithms supported by Hadoop.

Keywords- *MapReduce, Hadoop, virtual MapReduce cluster, map-task scheduling, reduce-task scheduling*

I. INTRODUCTION

Map-reduce is a distributed programming model proposed by Google to process vast amount of data in a parallel manner. Due to programming-model simplicity, built-in data distribution, scalability, and fault tolerance .MapReduce and its open-source implementation called Hadoop have been widely employed by many companies, including Facebook, Amazon, IBM, Twitter, and Yahoo to process their business data. MapReduce has also been used to solve diverse applications, such as machine learning, data mining, bioinformatics, social network, and astronomy.

MapReduce enables a programmer to define a MapReduce job as a map function and a reduce function, and provides a runtime system to divide the job into multiple map tasks and reduce tasks and perform these tasks on a MapReduce cluster in parallel. Typically, a MapReduce cluster consists of a set of commodity machines/nodes located on several racks and interconnected with each other in a local area network (LAN). In this paper, we call this a conventional MapReduce cluster. Due to the fact that building and maintaining a conventional MapReduce cluster is costly for a person/organization with a limited budget, an alternative way is to establish a virtual MapReduce cluster by either renting a MapReduce framework from a MapReduce service provider (e.g., Amazon) or renting multiple virtual private servers (VPSs) from a VPS provider (e.g., Linode or Future Hosting). Each VPS is a virtual machine with its own operating system and disk space. Due to some reasons, such as availability issue of a datacenter or resource shortage on a popular datacenter, a tenant might rent VPSs from different datacenters operated by a same VPS provider to establish his/her virtual MapReduce cluster.

II. HADOOP FRAMEWORK

Apache Hadoop is an open source framework for distributed storage and processing of large sets of data on commodity hardware. Hadoop enables businesses to quickly gain insight from massive amounts of structured and unstructured data. Numerous Apache Software Foundation projects make up the services required by an enterprise to deploy, integrate and work with Hadoop.

A. Map/Reduce Programming Model

MapReduce paradigm is based on sending the computer to where the data resides. MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage. During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster. The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes. Most of the computing takes place on nodes with data on local disks that reduces the network traffic. After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.

B.HDFS File System

Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly fault tolerant and designed using low-cost hardware. HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

C. Input/Output Read and Write

The MapReduce framework operates on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types. The key and the value classes should be in serialized manner by the framework and hence, need to implement the Writable interface. Additionally, the key classes have to implement the Writable-Comparable interface to facilitate sorting by the framework. Input and Output types of a MapReduce job: (Input) <k1, v1> map -><k2, v2> reduce -><k3, v3>(Output).

III. BACKGROUND

The FIFO algorithm [2] is a default scheduling algorithm provided by Hadoop MRv1. It follows a strict job submission order to schedule each map task of a job and meanwhile attempts to schedule a map task to an idle node that is close to the corresponding map-input block. However, the FIFO algorithm only focuses on map-task scheduling, rather than reduce-task scheduling. Hence, when FIFO is adopted in a virtual MapReduce cluster, its low reduce-data locality might cause a long job turnaround time. Besides, FIFO is designed to achieve node locality and rack locality in conventional MapReduce clusters, rather than achieving the VPS-locality and Cen-locality in a virtual MapReduce cluster. Consequently, the map-data locality of FIFO might be low in a virtual MapReduce cluster. In addition to the FIFO algorithm, Hadoop also provides the fair scheduling algorithm and the capacity scheduling algorithm-*-

The former is proposed by Facebook to fairly assign computation resources to jobs such that all jobs obtain an equal share of resources over time. The latter, introduced by Yahoo!, also allows multiple users to share a Map-Reduce cluster. It supports multiple queues and allocates a fraction of a cluster’s computation resources to each queue, i.e., all jobs submitted to a queue can only access to the resource allocated to the queue. Similar to these two algorithms, JoSS allows multiple jobs to simultaneously share the computation resource of a virtual MapReduce cluster. But different from the two algorithms, JoSS further provides reduce-task scheduling to improve job performance. There have been many studies [3], [4], [6], [7], [11], [14] on MapReduce task scheduling.

Zaharia et al. [3] presented the delay scheduling algorithm to improve data locality by following the FIFO algorithm but relaxing the strict FIFO job order. If the scheduling heuristic cannot schedule a local map task, it postpones the execution of the corresponding job and searches for another local map task from pending jobs. A similar but improved approach is further introduced in [4]. However, similar to FIFO, this approach did not provide reduce-task scheduling. Jin et al. [5] proposed the BAlance-Reduce (BAR) algorithm, which produces an initial task allocation for all map tasks of a job and then takes network state and cluster workload into consideration to interactively adjust the task allocation to reduce job turnaround time. In order to simplify BAR, the authors assumed that all local map tasks spend identical execution time. But this assumption is not realistic since the map-task execution time fluctuates even though when the processed input size is the same. Besides, reduce-task scheduling was not addressed by BAR. Tian et al. [6] proposed a MapReduce workload prediction mechanism to classify MapReduce workloads into three categories based on their CPU and I/O utilizations and then proposed a Triple-Queue Scheduler to improve the usage of both CPU and disk I/O resources under heterogeneous workloads.

Guo [7] presented an optimal map-task scheduling algorithm, which converts a task assignment problem into a Linear Sum Assignment Problem so as to find the optimal assignment. Nevertheless, applying this algorithm to real-world MapReduce clusters needs to carefully determine an appropriate time point to conduct the algorithm since slaves might become idle at different time points. Ehsan and Sion [8] introduced a co-scheduler called LiPS, which utilizes linear programming to simultaneously co-schedule map-input data and map tasks to nodes such that dollar cost can be minimized. But their assumption, i.e., MapReduce jobs and their input data are submitted together, might increase job turnaround time since replicating the data to the distributed filesystem of the cluster needs to take a while. Polo et al. [9] introduced a task scheduler to dynamically predict the performance of concurrent MapReduce jobs and adjust the resource allocation for the jobs. The goal is to allow MapReduce jobs to meet their performance objectives without over-provisioning of physical resources. Some other studies aim to enhance the performance of MapReduce in a cloud environment.

Palanisamy et al. [10] presented a MapReduce resource allocation system called Purlieus, which enables a cloud provider to place MapReduce input data to appropriate physical machines and then place VMs to the physical machines so as to provide both map locality and reduce locality. Different from Purlieus, JoSS presented in this paper is designed from the perspective of a tenant who rents VPSs from a VPS provider to build a virtual MapReduce cluster, rather than from the perspective of a cloud provider. Park et al. [11] introduced a locality-aware dynamic VM reconfiguration technique for virtual clusters running the Hadoop platform by dynamically changing the computing resource of a VM to maximize the data locality of map tasks. Bu et al. [12] proposed a task scheduling strategy called ILA to mitigate interference between virtual machines and meanwhile preserve MapReduce task data locality. Similar to [10], the schemes proposed in [11] and [12] were designed from the viewpoint of a cloud provider since the data locality in all layers including node locality, rack locality, and off-rack are clear to the provider. However, in a virtual MapReduce cluster considered in this study, a tenant does not know all of the above mentioned data-locality levels.

IV. THE EXISTING SCHEME

A.Hadoop default FIFO scheduler

The Hadoop default FIFO scheduler has already taken data locality into account. When a slave node with empty map slots sends the heartbeat signal, the MapReduce scheduler checks the first job in the queue. If the job has map tasks whose input data blocks are stored in the slave node, the scheduler assigns the node one of these local tasks. If a slave node has more unused map slots, the scheduler will keep assigning local tasks to the node. However, if the scheduler can no longer find a local task from the first job, it assigns the node one and only one non-local task during this heartbeat interval, no matter how many free slots the node has. This default FIFO scheduler, however, has deficiencies. First of all, it follows the strict FIFO job order to assign tasks, which means it will not allocate any task from other jobs if the first job in the queue still has an unassigned map task. Secondly, the data locality is randomly decided by the heartbeat sequence of slave nodes. If we have a large cluster that executes many small jobs, the data locality rate could be quite low. As mentioned, in a MapReduce cluster, tasks are assigned to a slave node in response to the node's heartbeat. With the FIFO scheduler, heartbeats are also processed in a FIFO order and a node is assigned a non-local map task when there is no local task from the first job. In a large cluster many nodes heartbeat simultaneously. However, a small job has less input data that are stored in a small number of nodes. It is thus a high probability event that the scheduler assigns tasks to slave nodes that do not have the small job's input data but give heartbeats first. A slave node with empty map slots that sends in a heartbeat first will always be assigned at least one map task, local or non-local. It is highly likely that the job's tasks will be assigned to many of the nodes which do not have the input data blocks before a node even gets a chance to grab a local task from the job.

B.The Fair Scheduler

The Fair Scheduler aims to give every user a fair share of the cluster capacity over time. If a single job is running, it gets all of the cluster. As more jobs are submitted, free task slots are given to the jobs in such a way as to give each user a fair share of the cluster. A short job belonging to one user will complete in a reasonable time even while another user's long job is running, and the long job will still make progress. Jobs are placed in pools, and by default, each user gets their own pool. A user who submits more jobs than a second user will not get any more cluster resources than the second, on average. It is also possible to define custom pools with guaranteed minimum capacities defined in terms of the number of map and reduce slots, and to set weightings for each pool. The Fair Scheduler supports preemption, so if a pool has not received its fair share for a certain period of time, then the scheduler will kill tasks in pools running over capacity in order to give the slots to the pool running under capacity.

C.The Capacity Scheduler

The Capacity Scheduler takes a slightly different approach to multiuser scheduling. A cluster is made up of a number of queues (like the Fair Scheduler's pools), which may be hierarchical (so a queue may be the child of another queue), and each queue has an allocated capacity. This is like the Fair Scheduler, except that within each queue, jobs are scheduled using FIFO scheduling (with priorities). In effect, the Capacity Scheduler allows users or organizations (defined using queues) to simulate a separate MapReduce cluster with FIFO scheduling for each user or organization. The Fair Scheduler, by contrast, enforces fair sharing within each pool, so running jobs share the pool's resources.

V. PROPOSED SYSTEM

The proposed system implements JoSS-T in Hadoop-0.20.2 and conduct extensive experiments to compare them with several known scheduling algorithms supported by Hadoop, including the FIFO algorithm, Fair scheduling algorithm, and Capacity scheduling algorithm. The experimental results demonstrate that JoSS-T outperform the other tested algorithms in terms of map-data locality, reduce-data locality, and network overhead without causing too much overhead, regardless of job type and scale.

The contributions of this proposed system are as follows.

1. We introduce JoSS to appropriately schedule Map-Reduce jobs in a virtual MapReduce cluster by addressing both map-data locality and reduce-data locality from the perspective of a tenant.
2. By classifying jobs into map-heavy and reduce heavy jobs and designing the corresponding policies to schedule each class of jobs, JoSS increases data locality and improves job performance.

Furthermore, by classifying jobs into large and small jobs and scheduling them in a round-robin fashion, JoSS avoids job starvation and improves job performance.

3. A formal proof is also provided to determine the best threshold for classifying MapReduce jobs.
4. JoSS-T is proposed to achieve two conflicting goals: speeding up task assignment and further increasing the VPS-locality.
5. We refer to a set of MapReduce benchmarks to create two different MapReduce workloads for evaluating and comparing JoSS-T with three known scheduling algorithms supported by Hadoop. Moreover, a set of metrics showing data-locality, network overhead, job performance, and load balance are used to achieve a comprehensive comparison.

A.Job Classification

Before introducing the algorithm of JoSS, first describe how JoSS classifies jobs and schedules each class of jobs. Let *Sreduce* and *Smap* be the total reduce-input size and the total map-input size of J, respectively. Based on the ratio of *Sreduce* over *Smap*, J can be classified into either a reduce heavy job or a map-heavy job. If J satisfies Eq. (1), implying that the network overhead is dominated by J's reduce-input data, then J is classified as a reduce-heavy job (RH job for short). Otherwise, J is classified as a map-heavy job (MH job for short). Note that td is a threshold to determine the classification, td ≥ 0.

$$\frac{Sreduce}{Smap} > td$$
(1)

In fact, $Smap = \sum_{i=1}^{m} | Bi |$ where |Bi| is the size of Bi, and $Sreduce = \sum_{i=1}^{m}(| Bi | \cdot FPi)$ where FPi is the filtering percentage of Bi showing the ratio of Mi's map-output size over Mi's map-input size, FPi ≥ 0. In order to reduce Eq. (1) and the above classification, we chose five MapReduce benchmarks: Word-Count, Grep, Inverted-Index, Sequence-Count and Permu from PUMA to conduct experiments. Eq. (1) can be reduced as based on the analysis from [1],

$$\frac{\boldsymbol{Sreduce}}{\boldsymbol{Smap}} = \frac{\sum_{i=1}^{m}(| Bi | \cdot FPi)}{\sum_{i=1}^{m} | Bi |} = FPj > td \qquad (2)$$

and the condition used to classify J can be reduced as

$$\text{J}=\begin{cases} a\ RH\ job, & if\ FPj\ > td \\ a\ MH\ job, & else. \end{cases}$$
(3)

Based on the input scale of J to Navg VPS, which is the average datacenter scale of a virtual MapReduce ,cluster the classification rule is below,

$$J = \begin{cases} a\ small\ job, if\ m \le Navg\_VPS \\ a\ large\ job, \qquad else. \end{cases}$$
(4)

B.Scheduling Policies

JoSS utilizes the following three scheduling policies.

- *Policy A*

This policy is designed for a small RH job. If J is a small RH job, it would be better that each reducer of J is close to all mappers of J since the reducer can more quickly retrieve its input data from all the mappers. But this also implies that all mappers of J should be close to each other. Hence, policy A works as follows. It first chooses *cenw*, which is a datacenter having the least amount of unprocessed tasks among all the k datacenters, *cenw*, belongs to *cen1,cen2, . . . ,cenk*. Then it schedules all tasks of J to *cenw* by putting J's map tasks and J's reduce tasks at the end of *MQw,0* and *RQw,0*, respectively. In this way, all these tasks can be executed only by the VPSs at *cenw*, and each reducer of J can retrieve its input data from its local datacenter (i.e., reduce-data locality can be improved).

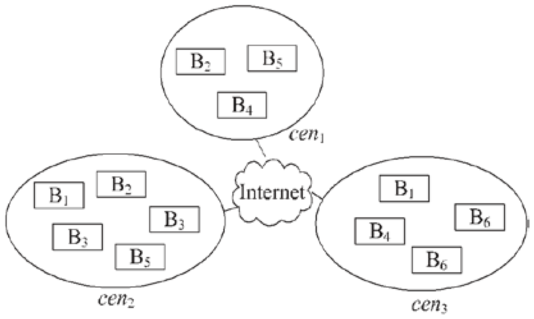

Fig. 1. An example showing block locations of job Y in a virtual MapReduce cluster comprising three datacenters.

- *Policy B*

This policy is designed for a small MH job. If J is a small MH job, it would be better that each mapper of J is close to its input block, and each reducer of J is close to most mappers of J . Hence, policy B works as follows: It schedules J 's map tasks based on the number of unique input blocks of J held by each datacenter. If a datacenter holds more unique blocks of J , more map tasks of J will be scheduled to the VPSs at this datacenter. The purpose is allowing each mapper of J to retrieve its input block from its local datacenter. In addition, to make J 's reducers close to most J 's mappers, policy B schedules all reduce tasks of J to the datacenter that holds the maximum number of J 's unique blocks.

The task scheduler of JoSS
Input : *J* and input-data description
Output:task-scheduling decision
Procedure:

```
1.  Calculate a hash value for J's executable code and J's input-data type;
2.  Let H be a set of hash values previously generated by JoSS;
3.  If the hash value is not in H{
4.     Append all map tasks of J to the end of MQFIFO;
5.     Append all reduce tasks of J to the end of RQFIFO; }
6.  else {
7.     if J is a small RH job {//Use policy A.
8.        Let cenw be a datacenter having the least unprocessed tasks among cen1,cen2,…cenk;
9.     Append all map tasks of J to the end of MQw,0;
10.    Append all reduce tasks of J to the end of RQw,0;}
11. else {
12.     Let Lc be a set of all unique input blocks of J held by cenc where c=1,2,…k;
13.     Let α = m; /*m is the number of map tasks of J.*/
14.     while α > 0{/*i.e., not all map tasks of J are scheduled.*/
15.     Let Ld is the first largest set among L1,L2,…..,Lk;
16.     Let |Ld| be the size of Ld;
17.     Let cend be the related datacenter;
18.     If J is a small MH job {//Use policy B
19.        Append |Ld | map tasks of J to the end of MQd,0;}
20.    else {/*i.e., J is a large job, so use policy C.*/
21.        Let ρ be the total number of map-task queues in cend;
22.        Generate a new map-task queue MQd,p+1;
23.        Append |Ld| map tasks of J to the end of MQd,p+1;}
24.        for c=1 to k{
25.        Delete a block from Lc if the block is in Ld;}
26.        α = α-|Ld|;}
27.        Let cenc be a datacenter holding the largest number of unique input blocks of J;
28. If J is a small MH job{//Use policy B.
29.      Append all reduce tasks of J to the end of RQe,0;}
30. else { /*i.e., J is large job, so use policy C.*/
31.      Let q be the total number of reduce-task queue in cenc;
32.      Generate a new reduce-task queue RQe,q+1;
33. Append all reduce tasks of J to the end of RQe,q+1; }}}
```

Fig 2.The algorithm of task scheduler.

For example, Fig. 1 illustrates the locations of all blocks of a job Y over three datacenters (Note that the input file of Y is fragmented into six blocks, and each block has two replicas.). Since *cen2* holds the largest number of Y's unique blocks (i.e., four), policy B will schedule four map tasks of Y to *cen2* to process B1 , B2 , B3 , and B5 by appending the four map tasks to the end of *MQ2,0*. After that, *cen1* still holds one unscheduled block of Y (i.e., B4) and *cen3* still holds two unscheduled blocks of Y (i.e., B4 and B6). Hence, policy B will schedule the remaining two map tasks of Y to *cen3* to process B4 and B6 by inserting the two map tasks to the end of *MQ3,0* . Finally, due to the fact that *cen2* holds the maximum number of unique blocks of Y, policy B schedules all reduce tasks of Y to *cen2* by appending them to the end of *RQ2,0*.

- *Policy C*

This policy is designed for a large job. If J is a large job to a virtual MapReduce cluster, using one datacenter of the cluster to run all map tasks of J might need several rounds to finish these map tasks, implying that job turnaround time will prolong. To prevent this from happening, it is better not to use a single datacenter to run all these map tasks.

Hence, as long as J is a large job, JoSS utilizes policy C, which in fact uses the same strategy of policy B to schedule all tasks of J . However, in policy C, all the map tasks scheduled to *cenc* will not be put into *MQc,0* since *MQc,0* is reserved for only small jobs. Instead, these map tasks will be put into a new map-task queue created for *cenc*. Similarly, the reduce tasks of the large job scheduled to *cenc* will be put into a new reduce-task queue created for *cenc* , rather than *RQc,0* . The purpose is to separate large jobs and small jobs into different queues and allow JoSS to avoid job starvation.

C. Job Driven Scheduling Scheme(JoSS) and JoSS-T

JoSS consists of three components: input-data classifier, task scheduler, and task assigner. The input-data classifier is designed to classify input data uploaded by a user into one of the two types: web document and non-web document. A web document refers to a file consisting of a lot of tags enclosed in angle brackets. By simply inspecting the first several sentences of a document, the input-data classifier can easily know if it is a web document or not. After the classification, the input- data classifier records the type of the input data in JoSS.

Whenever receiving a MapReduce job from a user, the task scheduler determines the type of the job and then schedules the job based on one of policies A, B, and C.

Fig.2 illustrates the algorithm of the task scheduler. Upon receiving J , the task scheduler retrieves J 's input- data type classified by the input-data classifier and checks whether JoSS has executed J on such input-data type or not by calculating the corresponding hash value and comparing the value with H, where H is a set of hash values previously generated and recorded by JoSS.

If the hash value is not in H (see line 4), it means that JoSS does not know J 's average filtering-percentage value and J's job classification. To obtain the above information, the task scheduler simply appends J 's all map tasks and J 's all reduce tasks to two queues, denoted by *MQFIFO* and *RQFIFO* , respectively. This allows the task assigner to use the Hadoop FIFO algorithm [2] to assign these tasks to idle VPSs. Once J is completed, JoSS records the corresponding hash value and average filtering-percentage value.

However, if the hash value is in H (see line 7), it means that JoSS knows the average filtering-percentage value of J . Then the task scheduler schedules J as follows: If J is a small RH job, the abovementioned policy A is used to schedule the tasks of J (please see lines 9 to 12). Otherwise, it means that J is either a small MH job or a large job, and the task scheduler uses lines 14 to 37 to schedule J . Recall that policies B and C are used to schedule a small MH job and a large job, respectively. If J is a small MH job, the task scheduler directly inserts J 's map tasks to the permanent map-task queue of the determined datacenter (see line 22), and also inserts J 's reduce tasks to the permanent reduce-task queue of the determined datacenter (see line 33). In other words, no additional queue will be created for any small jobs. The purpose is not to increase the queue management overhead of JoSS.

Task-driven Task Assigner(TTA)
Input: an idle slot of VPSc,l
Output: a task assigned to VPSc,l
Procedure:

```
1.  Let Imap and Ired be two indexes with the
    same initial value 0;
2.  while VPSc,l has an idle slot{
3.    Let Nmap be the total number of map-
    task queues in cenc;
4.    Let Nred be the total number of reduce-
    task queues in cenc;
5.    if the slot is a map slot{
6.      if MQFIFO is not empty{
7.        Use FIFO to assign a map task from
    MQFIFO to VPSc,l
8.        Remove the task from MQFIFO;}
9.      else{
10.       Imap = Imap mod (Nmap +1);
11.       Assign the first task from MQc,Imap
    to VPSc,l;
12.       Remove the task from MQc,Imap;
13.       Imap ++;}}
14. else{/*i.e., the idle slot is a reduce slot;*/
15.     if RQFIFO is not empty {
16.       Assign the first reduce task from
    RQFIFO to VPSc,l;
17.       Remove the task from RQFIFO;}
18.     else {
19.       Ired=Ired mod (Nred+1);
20.       Assign the first reduce task from
    RQc,Ired to VPSc,l;
21.       Remove the task from RQc,Ired;
22.       Ired++; }}}
```

Fig. 3. The algorithm of task-driven task assigner (TTA)

In another case, if J is a large job, the task scheduler addi- tionally generates a new map-task queue and a new reduce- task queue to respectively put J 's map tasks and J 's reduce tasks (see lines 24 to 26 and lines 35 to 37). This will allow the task assigner to properly assign small jobs and large jobs to VPSs.

Fig. 3 illustrates how TTA works. Whenever *VPSc* has an idle map slot, TTA preferentially assigns a map task from *MQFIFO* to *VPSc* based on the Hadoop FIFO algorithm (see lines 7 to 8). The goal is to preferentially execute all newly submitted jobs one by one and obtain their filtering-percentage values to determine their job classifications. However, if *MQFIFO* is empty, TTA assigns one of the first map tasks from all the other map-task queues of cenc in a round-robin fashion (see lines 10 to 13) such that tasks can be assigned quickly and job starvation can be avoided.

Similarly, whenever *VPSc* has an idle reduce slot, TTA preferentially assigns a reduce task from *RQFIFO* to *VPSc* . Only when *RQFIFO* is empty, TTA assigns one of the first reduce tasks from other reduce-task queues of *cenc* to *VPSc*;' (see lines 19 to 22).

VI. CONCLUSION

In this paper, we have introduced JoSS for scheduling Map- Reduce jobs in a virtual MapReduce cluster consisting of a set of VPSs rented from a VPS provider. Different from current MapReduce scheduling algorithms, JoSS takes both the map-data locality and reduce-data locality of a virtual MapReduce cluster into consideration. JoSS classifies jobs into three job types, i.e., small map-heavy job, small reduce-heavy job, and large job, and introduced appropriate policies to schedule each type of job. In addition, JoSS-T is further introduced to respectively achieve a fast task assignment and improve the VPS-locality.

VII . REFERENCES

[1] Ming-Chang Lee, Jia-Chun Lin, and Ramin Yahyapour ,“Hybrid Job-Driven Scheduling for Virtual MapReduce Clusters”, IEEE ,May 2016.

[2] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” Commun. ACM, vol. 51, no. 1, pp. 107–113,2008.

[3] 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 Proc. 5th Eur. Conf.Comput. Syst., Apr. 2010, pp. 265–278.

[4] C. He, Y. Lu, and D. Swanson, “Matchmaking: A new mapreduce scheduling technique,” in Proc. IEEE 3rd Int. Conf. Cloud Comput.Technol. Sci., Nov. 2011, pp. 40–47.

[5] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, “BAR: An efficientdata locality driven task scheduling algorithm for cloud computing,” in Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput., May 2011, pp. 295–304.

[6] C. Tian, H. Zhou, Y. He, and L. Zha, “A dynamic mapreduce scheduler for heterogeneous workloads,” in Proc. IEEE 8th Int.Conf. Grid Cooperative Comput., 2009, pp. 218–224.

[7] Z. Guo, G. Fox, and M. Zhou, “Investigation of data locality in mapreduce,” in Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud GridComput., May 2012, pp. 419–426.

[8] M. Ehsan, and R. Sion, “LiPS: A cost-efficient data and task co-scheduler for MapReduce,” in Proc. IEEE 27th Int. Symp.Parallel Distrib. Process. Workshops PhD Forum, May 2013,pp. 2230–2233.

[9] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguade, M. Steinder,and I. Whalley, “Performance-driven task co-scheduling for mapreduce environments,” in Proc. IEEE Netw. Oper. Manage. Symp.,2010, pp. 373–380.

[10] B. Palanisamy, A. Singh, L. Liu, and B. Jain, “Purlieus: Localityawareresource allocation for MapReduce in a cloud,” in Proc. Int.Conf. High Perform. Comput., Netw., Storage Anal., Nov. 2011, pp. 58.

[11] J. Park, D. Lee, B. Kim, J. Huh, and S. Maeng, “Locality-aware dynamic VM reconfiguration on MapReduce clouds,” in Proc. 21stInt. Symp. High-Perform. Parallel Distrib. Comput., Jun. 2012,pp. 27–36.

[12] X. Bu, J. Rao, and C.-Z. Xu, “Interference and locality-aware task scheduling for MapReduce applications in virtual clusters,” inProc. 22nd Int. Symp. High-Perform. Parallel Distrib. Comput., Jun.2013, pp. 227–238.

[13] S.-Y. Ko, I. Hoque, B. Cho, and I. Gupta, “Making cloud intermediate data fault-tolerant,” in Proc. ACM Symp. Cloud Comput.,2010, pp. 181–192.

[14] T. White, Hadoop: The Definitive Guide. Sebastopol, CA, USA:O’Reilly Media, Jun. 5, 2009.