Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 4, Issue. 5, May 2015, pg.905 – 913

RESEARCH ARTICLE

WPBS: A Workload Pattern Based Scheduler for Competent Task Assignments in Hadoop

Asst. Prof. Ranjana Nadagoudar[1], Shreerashmi B.S[2]

[1]Computer Science & VTU PG Centre Kalaburgi, India

[2]Computer Science & VTU PG Centre Kalaburgi, India

[1] ranjanapriya8@gmail.com; [2] sshreerashmi@gmail.com

Abstract— MapReduce has emerged as a popular paradigm for processing large datasets in parallel over a cluster. Hadoop is an open source implementation of Map Reduce, which is very attractive for parallel processing of a variety of different applications, e.g., web crawling, log processing, video and image analysis, recommendation systems, etc. Multiple users with various types of workloads share MapReduce cluster. When a group of jobs are simultaneously submitted to a MapReduce cluster, they compete for the pooled resources and the overall system performance in terms of job response times, might be degraded. Therefore, key challenge is the ability of proficient scheduling in such a shared MapReduce environment. However, we find that conventional scheduling algorithms supported by Hadoop cannot always assurance good average response times under diverse workloads. In this paper, we address this problem by introducing a novel Hadoop scheduler, which leverages the knowledge of workload patterns to reduce average job response times by dynamically tuning the resource shares among users and the scheduling algorithms for each user.

Keywords—MapReduce, Hadoop, Scheduling, Heterogeneous workloads

I. INTRODUCTION

Big data is the term for a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications. Usually, it requires parallel execution on hundreds or even thousands of servers. Google's services is typical example, which uses the MapReduce framework to process around 20 petabytes of data per day.

The MapReduce runtime consists of a single master process and a large number of slave processes. When a MapReduce application (or 'job') is submitted to the runtime, it is split into a large number of Map and Reduce tasks, which are executed by the slave nodes. The runtime is responsible for dispatching tasks to slave nodes and ensuring their completion. While MapReduce was originally used primarily for batch data processing, it is now also being used in shared, multi-user environments in which submitted jobs may have completely different priorities: from small, almost interactive, executions, to very long program that take hours to complete. This change makes task scheduling, which is responsible for selecting

tasks for execution across multiple jobs, even more relevant. Task selection and slave node assignment govern a job's opportunity to make progress, and thus influences job performance.

For a MapReduce cluster, job scheduling is performed on the master node, where map/reduce tasks are assigned to slave nodes through a heartbeat mechanism. Upon submission of a new job, Hadoop splits the job into map and reduce tasks Each map task generates intermediate results in key value pairs for a block of input data. Then, reduce tasks fetch these intermediate results through the copy/merge phase according to keys from every map task, maintaining the sort ordering, and conduct reduce functions after receiving all the intermediate results.. In the enterprise setting, users would benefit from sharing Hadoop clusters and consolidating diverse applications over the same datasets. Originally, Hadoop employed a simple FIFO scheduling policy. Designed with a primary goal of minimizing the make span of large, routinely executed batch workloads, the simple FIFO scheduler is quite efficient. However, job management using this policy is very inflexible: once long, production jobs are scheduled in the MapReduce cluster the later submitted short, interactive ad- hoc queries must wait until the earlier jobs finish, which can make their outcomes less relevant. Fair scheduler[7],[8] was proposed to improve the average job response times in shared Hadoop clusters by assigning to all jobs, on average, an equal share of resources over time. However, we notice that the Fair scheduler makes its scheduling decision without considering workload patterns of different users.

To address the above issues, we develop a novel Hadoop scheduler, called workload pattern size based scheduler for tasks assignment in Hadoop ie , which aims to improve overall performance by leveraging the present job size patterns to tune its scheduling schemes among multiple users and for a single user. Specifically, we first develop a lightweight information collector that tracks important statistic information of the recently finished jobs for each user. We then propose a self tuning scheduling policy which consists of the scheduling at two levels: the resource shares *across multiple users* are assigned based on the estimated job size of each user; and the job scheduling *for each individual user* is further adjusted to accommodate to that user's job size distribution. Experimental results in a real-world Hadoop cluster environment confirm the effectiveness and the robustness of our solution. We show that our scheduler improves the performance in terms of average job execution times.

II. ALGORITHM DESCRIPTION

There is the dependency between map and reduce tasks in hadoop, taking into this consideration the Hadoop scheduling can be formulated as a two-stage multi-processor flow-shop problem. However, finding the optimal solution with the minimum response times (flow times) is NP-hard [14]. Therefore, in this section we propose new adaptive scheduling algorithm which leverages the knowledge of workload characteristics to dynamically adjust the scheduling schemes, aiming to improve efficiency in terms of job response times, especially under heavy-tailed workloads[6].

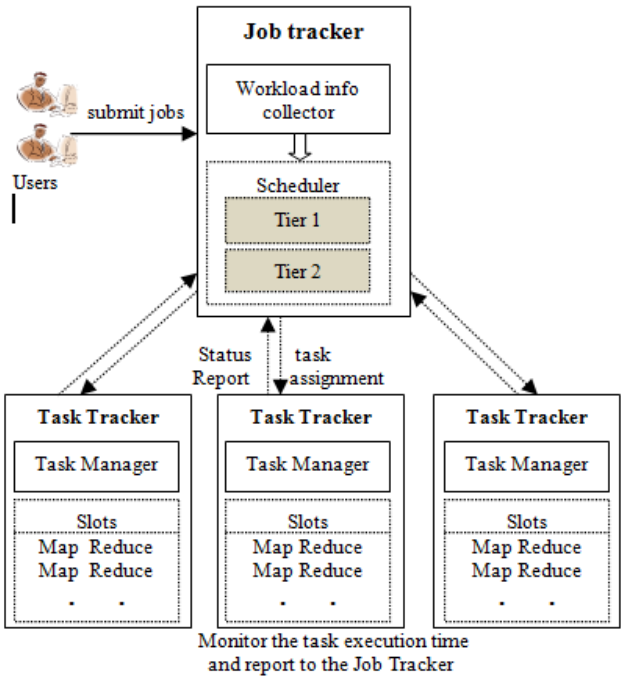

Fig. 1. The Architecture overview of workload pattern based scheduler

The architecture of Workload pattern based scheduler for competent Task Assignments in Hadoop is shown in Fig. 1.

The details of our designed WPBS scheduler are presented in Algorithms 1-3. Proposed scheduler consists of the following three components:

- Workload information collection: monitor the execution of each job and each task, and gather the workload information.
- Scheduling among multiple users: allocate (both map and reduce) slots for users according to their workload characteristics, i.e., scheduling at Tier 1.
- Scheduling for each individual user: tune the scheduling schemes for jobs from each individual user based on that user's job size distribution, i.e., scheduling at Tier 2.

In this section, we present the detailed implementation of the above three components. WPBS appropriately allocates slots for Hadoop users and guides each user to select the right scheduling algorithm for their own job queue, even under highly variable and heavy-tailed workloads. In the remainder of this section, we describe the detailed implementation of the above three components. Table I lists some notations used in the rest of this paper.

TABLE I

U / u_i	number of users / i-th user, i ϵ [1,U].
J_i / $job_{i,j}$	set of all users i's jobs / j-th job of user i.
$t_{i,j}^m$ / $t_{i,j}^r$	average map / reduce task exe.time of $job_{i,j}$.
t_i^{-m} / t_i^{-r}	average map/reduce task exe. time of jobs from u_i.
$n_{i,j}^m$ / $n_{i,j}^r$	number of map/reduce tasks in $job_{i,j}$.
$S_{i,j}$	Size of $job_{i,j}$, i.e., total exe .time of tasks.
$\overline{S}_i / \overline{S}_i^*$	average size of completed/ current jobs from u_i.
CV_i / CV_i^*	CV of completed/current job sizes of u_i.
SU_i / $SJ_{i,j}$	the slot share of u_i / the slot share of $job_{i,j}$.

AS_i	the slot share that u_i actually received.
T_w	time window for collecting historic informations.
W	window size for collecting historic informations.

A. Workload Information Collection.

In this subsection, we first introduce a light-weighted history information collector in WPBS that gathers the important statistical information of jobs and users. Basically, we collect and update the information of each job's map and reduce tasks separately by the same means. To avoid redundant description, we use the general term *task* to represent both types of tasks and the term *phase size* to represent size of either map phase or reduce phase of each job.

In WPBS, the workload information that needs to be collected for each user u_i includes average task execution time t_i^{-m}(or t_i^{-r}), average job size $\overline{s}_i$, and the coefficient of variation CV_i of job sizes. The Welford's one-pass algorithm is used to on-line update these statistics.

$$t_i^{-m} = t_i^{-m} + (t_{i,j}^{-m} - t_i^{-m}) / j , \quad (1)$$

$$t_i^{-r} = t_i^{-r} + (t_{i,j}^{-r} - t_i^{-r}) / j , \quad (2)$$

$$s_{i,j} = t_{i,j}^{m} * n_{i,j}^{m} + t_{i,j}^{r} * n_{i,j}^{r} , \quad (3)$$

$$\overline{S}_i = \overline{S}_i + (s_{i,j} - \overline{S}_i) / j , \quad (4)$$

$$v_i = v_i + (s_{i,j} - \overline{S}_i)^2 \cdot (j - 1)/j , \quad (5)$$

$$CV_i = \sqrt{v_i / j} / \overline{S}_i , \quad (6)$$

where t_i^{-m} (resp. t_i^{-r}) represents the measured average map (resp. reduce) task execution time of $job_{i,j}$, $n_{i,j}^{m}$ (resp. $n_{i,j}^{r}$) indicates the number of map (resp. reduce) tasks of $job_{i,j}$, and $s_{i,j}$ denotes the size of the j-th completed job of user u_i (i.e., $job_{i,j}$). The average job size $\overline{S_i}$ (resp. map and reduce task execution time, t_i^{-m} and t_i^{-r} of user u_i is set to be zero initially and updated once a new job's size information is collected.

Upon each job's completion, WPBS collects its task execution time and updates the workload statistics for the corresponding user using the above equations, i.e., Eq.(1-6). The statistical information will then be utilized by WPBS to tune the schemes for scheduling the jobs arriving in the next time period.

The overview of WPBS is presented in Algorithm 1.

Algorithm 1 Overview of the WPBS

1. When a new job from user i is submitted
   a. Estimate job size and avg. job size $\overline{s}_i^*$ of user i using Eq.7;
   b. Adjust slot shares among all active users, see Alg. 2;
   c. Tune the job scheduling scheme for user i, see Alg. 3;
2. When a task of job j from user i is finished
   a. Update the estimated average task execution time $\overline{t}_{i,j}^*$:
3. When the j-th job from user i is finished
   a. Measure avg. map/reduce task execution time $t_{i,j}^{m}$ / $t_{i,j}^{r}$ and map/reduce task number $n_{i,j}^{m}$ / $n_{i,j}^{r}$;
   b. Update history info. of user i, i.e. using Eq.(1-6);
4. When a free slot is available

a. Sort users in decreasing order of deficits ASi - SUi;
b. Assign the slot to the first user in the sorted list;
c. Increase num. of actual received slots ASi* by 1;
d. Choose a job from user ui_ to get service based on the current scheduling scheme.

B. Scheduling Among Multiple User.

In this subsection, we present our algorithm (i.e., Algorithm 2) for scheduling among multiple users, i.e., assigning slots to all users. Although there are two types of slots, i.e., map slots and reduce slots, in a Hadoop system, we use the same strategy to allocate them to different users. For simplicity, we present a general algorithm in the rest of this subsection which can be applied to both types of slots. Note that a job could get different slot assignments for its map and reduce tasks if they have different workload characteristics.

Specifically, WPBS adaptively adjusts the slot shares among all active users such that the share ratio is inversely proportional to the ratio of their average job phase sizes. Consequently, WPBS implicitly gives higher priority to users with smaller jobs. We expect that WPBS can avoid small jobs waiting behind large ones and thus improve the overall performance.

One critical issue that needs to be addressed is how to correctly measure the phase sizes of the jobs that are currently running or waiting for the service. In Hadoop systems, it is not easy to obtain the exact execution times of job's tasks before they are finished. In this paper, we instead estimate a job's phase size, as the production of its task number and the average task execution time. In particular, the total task number of $job_{i,j}$, i.e., $n_{i,j}$, can be obtained immediately when the job is submitted, and the execution times of tasks from the same job can be assumed to be close to each other. Therefore, if $job_{i,j}$ is running, we use the average execution time of the finished tasks of $job_{i,j}$,e.g. $\overline{t}^{*}_{i,j}$, *to* represent its overall average task execution time $\overline{t}_{i,j}$ For those jobs from user u_i that are still waiting for service or currently running but have no finished tasks, we use the average task execution times $\overline{t}_i$ of jobs from the same user to approximate their average task execution times $\overline{t}_{i,j}$.

Algorithm 2 Tier 1: Allocate slots for each user

Input: historic information of each active user;
Output: slot share SUi of each active user;
for each user u_i do
  Update that user's slot share SU_i using Eq.8;
  for j-th job of user i, i.e., $job_{i,j}$ do
    if currently scheduling based on submission times then
      if $job_{i,j}$ has the earliest submission time in J_i then
        $SJ_{i,j} = SU_i$;
      else
        $SJ_{i,j} = 0$;
    else
      $SJ_{i,j} = SU_i / |J_i|$.

Therefore, user u_i's average map/reduce phase size of jobs is calculated as follows,

$$\overline{S}_i^{*} = \frac{1}{|J_i|} \cdot \sum_{j=1}^{|J_i|} n_{i,j}^{m} \cdot \bar{t}_{i,j}^{m} \qquad (7)$$

where J_i represents the set of jobs from user u_i that are currently running or waiting for service.

Once a new job arrives, WPBS updates the average size of that job's owner and then adaptively adjusts the map and the reduce slot shares (SU_i) among all active users using Eq.8.

$$SU_i = SU_i \,.\, (\alpha \,.\, U \,.\, \frac{\frac{1}{S_i^*}}{\sum_{i=1}^{U} \frac{1}{S_i^*}} + 1 - \alpha), \qquad (8)$$

$$\forall i,\ SU_i > 0, \qquad (9)$$

$$\sum_{i=1}^{U} SU_i = \sum_{i=1}^{U} SU_i^* \qquad (10)$$

where SU_i represents the slot shares for user u_i under the *Fair* scheme, i.e., equally dispatching the slots among users, U indicates the number of active users, and α is a tuning parameter within the range from 0 to 1. Parameter α in Eq.8 is used to control how aggressively proposed scheduler biases towards the users with smaller job phase sizes: when α is close to 0, our scheduler increases the degree of fairness among all users, performing similar as *Fair*; and when α is increased to 1, WPBS gives the strong bias towards the users with small jobs in order to improve the efficiency. In the remainder of the paper, we set α to 1 if there is no explicit specification. We remark that through setting parameter α, one can tune WPBS to meet different predefined targets, e.g., fairness or efficiency. We also remark that it is guaranteed no active users gets starved for slots, see Eq.9, and all available slots in the system are fully distributed to active users, see Eq.10.

The new slot shares (i.e., SU_i) will then be used to determine which user can receive the slot that just became available for redistribution. Specifically, WPBS sorts all active users in a non-increasing order of their deficits, i.e., the gap between the expected assigned slots (SU_i) and the actual received slots (AS_i), and then dispatches that particular slot to the user with the largest deficit. Additionally, some users might have high deficits but their actual demands on map/reduce slots are less than the expected shares. In such a case, WPBS re-dispatches the extra slots to those users who have less deficits but need more slots for serving their jobs.

C. Scheduling for A Single User.

The second design principle used in WPBS is to dynamically tune the scheduling scheme for jobs from an individual user by leveraging the knowledge of job size distribution.

Our algorithm considers the CV of total job sizes, i.e., map size plus reduce size, of each user to determine which scheme should be used for jobs from the same user. In order to accurately estimate CV′ of each user's current job sizes, we combine the history information of recently finished jobs and the estimated size distribution of current jobs that are running or waiting in the system. The past CV_i of user u_i can be provided by the information collector. The current CV_i' can be calculated using Eq.(3- 6) but replacing average task execution times $\overline{t}_{i,j}$ with average task execution times $\overline{t}_i$ of jobs from the same user. If both CV values of a user are smaller than 1, then the WPBS scheme schedules the current jobs from that user in the order of their submission times. Otherwise the user level scheduler fairly assigns slots among jobs. It is also possible that the two values are conflicting, i.e., $CV_i > 1$ and $CV_i^* < 1$ or vise versa, which means the user's workload pattern changes. Under such case, the fair scheme is adopted to assign slots to that user's jobs. Meanwhile, the history information is reset by starting a new collection window. see the pseudo-code in Algorithm 3.

Algorithm 3 Tier 2: Tune job scheduling for each user

Input: historic information of each active user;
Output: UseF IFO vector;
for each user u_i **do**
  if user u_i is active, i.e., $|J_i| > 1$ **then**
    calculate the CV_1^* of current jobs;
    if CV_1^* and $CV_i < 1$ **then**
      schedule current jobs based on their submission times;
    if $CV_1^* > 1 \| CV_i > 1$ **then**
      equally allocate slots among current jobs;
      clear history information and restart collection.

III. RELATED WORK

Task assigning is one of the important processes in Hadoop. FIFO is the default scheduler for almost all Hadoop applications. The Job queue is processed in First in First out fashion. With the scheduler, the main drawback is that only after finishing the previous job, next jobs in the job queue will be assigned. The scheduler implementation is simple and efficient. Fair scheduler[1] is introduced by Facebook. Here, fair sharing of resources is possible. Main advantage of the scheduler is that whenever slot becomes free, shorter jobs can be assigned. Main drawback is that tasks will be allocated to all the slots in the cluster with maximum slot capacity. Joel Wolf et. al. proposed a slot allocation scheduler called FLEX[2] that could optimize towards a given scheduling metric, e.g., average response time, make span, etc., by sorting jobs with generic schemes before allocating slots. Capacity scheduler is introduced by Yahoo. Capacity scheduler is very effective for large scale applications. Other than job pool, here multiple job queues are allocated. There is a guaranteed capacity for each queue. When the queue capacity exceeds the job will be allocated to other queues. Here, higher priority jobs can access resources fastly than lower priority jobs. In the job queue, Capacity scheduler follows FIFO scheduling with priority. But there is a limit on percentage of running tasks per user. Another major direction of improving the Hadoop scheduling policy is considering the deadline or SLA of jobs. A deadline based scheduler was proposed in[3], which utilizes earliest deadline first policy to sort jobs and the lagrange optimization method to find out the jobs' minimum map and reduce slots number requirements to meet their deadline. Jorda Polo et al. estimated the task execution time of a job from the average execution time of already finished tasks of that job, and calculate the slots number a job needs from its deadline and estimated task execution time. We partly adopt the method to help estimate the job size of each user in our proposed WPBS scheduler. Longest Approximate Time to end (LATE) scheduler [4] is mainly focusing on speculative execution of tasks. In speculative execution, when a task execution is performing slowing, on the back end an equivalent task is also running. If that back end task complete fastly performance will be improved. Due to speculative execution response time will be improved. In a heterogeneous environment LATE scheduler is robust. Jaideep Dhok et al. [5] applied the Bayes pattern classification method to classify and sort jobs and dynamically decided the slots numbers a node can provide according to its load limitation instead of assigning fixed number of slots to each node.

IV. RESULTS AND DISCUSSION

We implement and evaluate the WPBS scheduler in Amazon EC2, a cloud platform that provides pools of computing resources to developers for flexibly configuring and scaling their compute capacity on demand.

1) Experimental Setting: In our experiments, we lease a m1.large instance as master node to perform *heartbeat* and *jobtracker* routines for job scheduling. Additionally, we use another 11 m1.large instances to launch slave nodes, each of which is configured with two map slots and two reduce slots. As the Hadoop project provides an API to support pluggable schedulers, we implement WPBS in Hadoop by extending the *TaskScheduler* interface and then change *mapred.jobtracker.taskScheduler* in the Hadoop configuration file to plug our scheduler into the Hadoop system. In addition, the randomtextwriter program and RandomWriter application are also used.

2) Performance Evaluation: We conduct experiments with the mixed MapReduce applications, aiming to evaluate WPBS performance in a diverse environment of both CPU-bound applications and IO-bound applications. In our experiments, there are four different users each of which submits a set of jobs for one type of applications above. Different distributions are introduced in both inter-arrival times and job sizes for each user, see Table II.

The relative performance improvement against FIFO is also plotted in the figure. We first observe that all the users experience worst performance under FIFO when the workload is a complex mixture of demands from multiple users. The performance degradation comes mainly from the fact that the extremely large jobs from user 4, i.e., Sort, take over all the resources in the cluster and stuck all the following small jobs from other users. We also observe that, under the FIFO policy, all the users tend to have similar average job execution time despite of their different job size patterns. On the other hand, *Fair* could improve the performance by allowing jobs from all users to get resource shares. Therefore, small jobs gain more benefits under *Fair* policy compared with FIFO by avoiding waiting for large ones. As a result, the average execution times of the first three users, which in average submit small jobs, are improved by a factor of 2. We further observe that better performance is achieve under WPBS such that the overall performance is improved by a factor of 3.5 and 1.8 over FIFO and *Fair*, respectively. We interpret it as an outcome of setting suitable scheduling algorithms for each user based on their corresponding workload features. The largest performance improvements come from the first two users. WPBS significantly reduces the average job execution time of user 1 by assigning more resources to it. For user 2, when WPBS detects that this user keeps submitting jobs with similar sizes, it turns to schedule the jobs from this user based on their submission times because Fair scheduling now cannot achieve any benefits due to the uniform job size distribution. Moreover, compared to FIFO, the performance of user 4 is not sacrificed although the WPBS policy discriminately gives it less resources due to its large jobs.

TABLE II Experimental setting for 4 users in Amazon EC2

User	job type	Avg. input Size	Input size pattern	job arrival pattern	Average Inter-arrival time	Submission number
1	WC	100MB	Exponential	Bursty	20sec	150
2	Pi Est.	-	-	Uniform	30sec	100
3	GREP	200MB	Bursty	Exponential	100sec	30
4	Sort	10GB	Uniform	Exponential	600sec	5

The experimental results are shown in Figure 2.

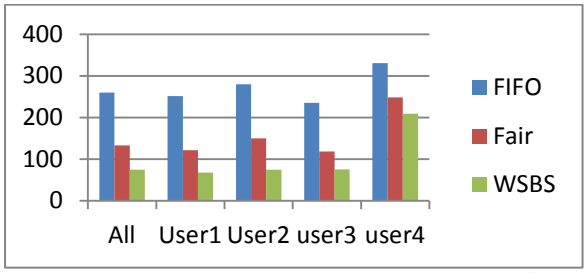

Fig. 2. Average job execution time, i.e., the duration between job submission and completion, for all users and schedulers. The relative improvements over FIFO are plotted on the bars of *Fair* and WPBS.

CONCLUSION

Task assignment in in Hadoop is an interesting problem because efficient task assignment can significantly reduce runtime, or improve hardware utilization. To improve the average job response time of Hadoop systems we have proposed WPBS an adaptive scheduling technique for improving the efficiency of Hadoop systems that process diverse MapReduce jobs. MapReduce workloads of modern enterprise clients have showed the diversity of job sizes, ranging from seconds to hours and having varying distributions as well. Our new wpbs scheme online captures the present job size patterns of each user and leverages this knowledge to dynamically adjust the slot shares among all active users and to further on-the-fly tune the scheme for scheduling jobs for a single user. Experiments in Amazon EC2 have shown that LsPS consistently improves the performance in terms of job execution times under a variety of system workloads.

ACKNOWLEDGEMENT

I would like to thank my guide Ranjana Nadagoudar for assisting me in this paper.

REFERENCES

[1] M. Zaharia, D. Borthakur, J. S. Sarma et al., "Job scheduling for multi-user mapreduce clusters," University of California, Berkeley, Tech. Rep., Apr. 2009

[2] J. Wolf, D. Rajan, K. Hildrum, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin, "Flex: A slot allocation scheduling optimizer for MapReduce workloads," in Middleware 2010. Springer, 2010, pp. 1–20.

[3] A. Verma, Ludmila Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for MapReduce environments," in ICAC'11, 2011, pp. 235–244M.

[4] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, "A methodology for understanding mapreduce performance under diverse workloads," University of California, Berkeley, Tech. Rep., 2010.

[5] J. Dhok and V. Varma, "Using pattern classification for task assignment in MapReduce," in ISEC'10, 2010.

[6] Ningfang Mi, "LsPS: A Job Size-Based Scheduler for Efficient Assignments in Hadoop", *IEEE Transactions on Cloud Computing*, , no. 1, pp. 1, PrePrints PrePrints, doi:10.1109/TCC.2014.2338291.

[7] .ApacheHadoop[Online]Available:http://hadoop.apache.org/

[8] Apache Hadoop Users. [Online]. Available: http://wiki.apache org/hadoop/PoweredBy.

[9] H. Chang, M. Kodialam, R. R. Kompella et al., "Schedulingin mapreduce-like systems for fast completion time," in INFOCOM'11.

[10] J. Polo, D. Carrera, Y. Becerra et al., "Performance driven task coscheduling for MapReduce environments," in NOMS'10, 2010.

