High-Responsive Scheduling with MapReduce Performance Prediction on Hadoop YARN

Yang Liu*, Yukun Zeng*, Xuefeng Piao†
School of Computer Science and Technology
Harbin Institute of Technology
Weihai, China
*Email: *{yang.liu.hitwh, yukun.zeng.hit}@gmail.com*
†hbpark@hit.edu.cn

***Abstract*—Hadoop is an open-source big data analysis platform that is widely used in both academia and industry. Decoupling of resource management and programming framework, the next generation of Hadoop, namely Hadoop YARN, is accommodated to various programming frameworks and capable of handling more kinds of workload, such as interactive analysis and stream processing. However, most existent schedulers in YARN are designed for batch processing and they do not value per-job response time, which results in low responsiveness of the Hadoop platform. This paper proposes a FSPY (Fair Sojourn Protocol in YARN) scheduler to improve responsiveness with guaranteeing fairness. FSPY relies on sizes of jobs, but which are unknown a priori. Consequently, we also present a job size prediction mechanism for MapReduce. Experimental results show that our scheduler outperforms Fair scheduler by 10x with respect to responsiveness under heavy workloads. Meanwhile, our prediction mechanism reaches an R2 prediction accuracy of 0.97.**

***Keywords*-YARN; scheduling; responsiveness; fairness; job size prediction**

I. Introduction

As state-of-the-art open-source big data processing platform, Apache Hadoop is now widely accepted not only in the academia, but also among real IT industries. The development of Hadoop goes through two phases: Hadoop 1.0 and Hadoop 2.0. In Hadoop 1.0, MapReduce [1] is the only supported programming framework and it is tightly coupled with the resource management scheme. In Hadoop 2.0, YARN [2] is introduced to independently manage computing resource of the Hadoop platform. It effectively decouples resource management and programming frameworks, which enables jobs of various programming frameworks to run simultaneously on a single cluster.

With the diversification of jobs on Hadoop, more and more concerns have been raised. One such concern is the system responsiveness which is important for both interactive analysis [3] and stream processing [4] applications. Under the scenario of interactive analysis, users have to wait for the results after job submission, thus the responsiveness will significantly influence user experience. Interactive analysis frameworks that have been ported onto YARN include Tez [5], Spark [6] and Drill [7]. Under the scenario of stream processing, large amount of stream data must be processed in real time. An example of stream data processing is real time financial fraud detection, in which a stream of transaction data is processed to detect fraudulent activities. One of the most famous stream processing frameworks is Storm [8], which has been ported onto YARN. Existing resource scheduling strategies in YARN are mainly designed for batch processing systems, which results in low responsiveness of the Hadoop platform. Native schedulers in YARN include:

- FIFO scheduler: The scheduler assigns higher priority to earlier-submitted jobs. It performs poor in terms of responsiveness, since small jobs submitted after a large job have to wait until the large one releases the resources.
- Fair scheduler: The scheduler assigns each job a concurrent fair share of cluster resources. It might also cause responsiveness problems when several large jobs are running in parallel.
- Capacity scheduler: The scheduler arranges jobs in several queues and allocates computing resources to each queue according to specified proportion. Inside a queue, both FIFO and Fair policies could be applied for resource scheduling.

To fulfill the growing demands of responsiveness in big data applications, it's necessary to develop new scheduling policies for Hadoop YARN. Related works show that size-based scheduling policies are commonly effective to improve the system responsiveness. As a typical size-based scheduler, Shortest Remaining Processing Time discipline (SRPT) [9] assigns the highest priority to jobs with the least remaining processing time. SRPT is optimal in terms of average response time in single server environment [10]. It is also efficient in multi-server environment [11] and has been applied to the scheduling of MapReduce jobs in [12], [13], [14].

Though SRPT can minimize the average response time,

†Xuefeng Piao is corresponding author

it penalizes large jobs in order to accelerate the completion of small ones, which can lead to starvation of large jobs. To solve this problem, another size-based scheduler Fair Sojourn Protocol (FSP) has been proposed. FSP assigns the highest priority to the job that would be completed first by processor sharing scheduler. According to the policy of processor sharing, jobs with smaller remaining size will finish earlier. Therefore, the FSP policy can also be stated that jobs with smaller virtual size will be assigned higher priority, where the virtual size refers to the job remaining size in the processor sharing scheduler. Assuming that job virtual sizes at each time point are known exactly, the sojourn time (i.e., the response time) of each job in FSP shall be no longer than that in processor sharing. In this sense, FSP effectively guarantees the fairness. Additionally, FSP is near optimal in responsiveness since the average response time of jobs under its scheduling is similar to that under SRPT.

Based on the principle of FSP, HFSP [15] has been presented to schedule MapReduce jobs in Hadoop 1.0. To obtain job virtual sizes at each time point, HFSP is equipped with two modules: a job size prediction module and a simulated cluster. The job size prediction module can only estimate the running phase size of a job. In other words, if a job is running in the map phase, the module cannot estimate its reduce-phase size or total size. The simulated cluster is designed to track job virtual sizes by simulating a Hadoop cluster running under Fair scheduler (a variant of processor sharing for Hadoop). However, this module is designed exclusively for resource management mechanism in Hadoop 1.0.

Applying the HFSP policy to YARN raises a number of challenges. Firstly, scheduling based on job phase sizes is not efficient on YARN. The resource management mechanism in Hadoop 1.0 is based on static slot partition, in which map slots and reduce slots are separated. Consequently, there is no resource contention between map tasks and reduce tasks. However, in YARN, the static slot partition is abandoned, and resource competitions exist between tasks of different job phases. As a result, scheduling based on job phase size can lead to flapping, i.e., jobs get priority alternately, which can bring down the scheduling performance. Secondly, the simulated cluster is not workable on YARN. In a simulated cluster, the scheduler usually need to foreknow the resource requests of a job whose virtual progress is ahead of its real progress. This is feasible in Hadoop 1.0, because job execution plans are accessible to the central scheduler. But in YARN, the job execution plan is managed by its own master process and not accessible to the central scheduler, which means the simulated cluster strategy of HFSP is no longer feasible in YARN. Moreover, the scheduling mechanism should be compatible with various programming frameworks on YARN.

To improve responsiveness for Hadoop YARN, we preserve some core ideas in FSP but make significant improvements on job size estimation, job virtual size calculation and YARN support. In our work,

- We present a new mechanism to predict MapReduce job sizes on YARN.
- We propose a light-weight job virtual size calculation approach that is well-adapted to YARN.
- We present a scheduling framework based on above works to improve responsiveness and guarantee fairness of Hadoop YARN.

The rest of this paper is organized as follows. Section II briefly reviews Hadoop YARN and defines our system model. Section III presents FSPY scheduler including job size prediction module, job virtual size calculation and scheduling framework. Section IV shows experimental results of our job size prediction module and FSPY scheduler. Finally, we conclude our work in Section V.

II. Definition

A. Jobs and Tasks

A job is a program instance that is submitted to a Hadoop system. We assume that every job is finite, which means that every job will be finished in a certain time period with enough resources. Jobs that are submitted but not finished yet are called pending jobs. If job J was submitted at time $Release(J)$ and finished at time $Finish(J)$, then the *response time* of *J* can be calculated as follows:

$$Response(J) = Finish(J) - Release(J) \qquad (1)$$

A *(*task) is a sub process in a job. Tasks of a job can be executed in parallel. The output of a task generally is a part of either the intermediate results or the final results of its host job. There might also be dependence relationships between tasks in a job. Tasks in a MapReduce job can be categorized into two subsets: map tasks and reduce tasks. Reduce tasks relies on the output of map tasks in the same job, thus the map tasks are generally launched before the reduce tasks. Therefore, the whole MapReduce job execution process can be divided into the map and the reduce phases. To improve the overall efficiency, there is often some overlapping between the map and the reduce phases in a job.

B. Resource

The execution of a job requires various types of computing resources. Currently, only two kinds of resource, namely memory and CPU, are scheduled in YARN. The basic unit of resource allocation is called a container. A container contains sufficient computing resources to run an application process (either an AM process or a task process). A container is not a set of concrete hardware, but a logical partition of resources.

When a job is submitted, the central *Resource Manager* (RM) of the cluster will first assign a container to the job to launch its *Application Master* (AM). After the AM is launched, it will schedule the execution of tasks in the job. The AM requests resources from the RM for tasks that are ready to run. The request of an AM contains information about number and resource capacity of its needed containers. In Hadoop, vcore is the basic unit of CPU while memory is measured in MB. Note that the RM scheduler makes no modifications to the resource quantity of requested containers and only determines when to allocate them. After some of the containers are allocated, the AM will further determine how to assign them to specific tasks. After a certain task is completed, its container will be recollected and then rescheduled by the RM.

C. Job size and job virtual size

We define *task size* as the product of task container resource quantity and its execution time, the *remaining size* of a task as the product of container resource quantity and remaining execution time. Like container resource, the size is also described by a tuple containing both memory and CPU resource quantities. A tasks execution time might differ when running in containers with different hardware and data locality. However, in a certain cluster, the execution time of a task is statistically predictable [16].

Job size equals to the sum of all task sizes in the job. Likewise, the remaining size of a job is sum of all task remaining sizes in the job. Now that the task size is predictable, the job size shall also be predictable. Suppose there is a virtual cluster that has the same resources with the real cluster but uses Fair scheduling policy. Every job submitted to the real cluster is at the meantime submitted to the virtual cluster. Then, we define *job virtual size* as the virtual remaining size of a job in the virtual cluster.

Suppose there is a virtual cluster that has the same resources with the real cluster but uses Fair scheduling policy. Every job submitted to the real cluster is at the meantime submitted to the virtual cluster. Then, we define *job virtual size* as the virtual remaining size of a job in the virtual cluster.

D. Parallelism bound and average parallelism bound

Parallelism bound is defined as the most possible resources that a job can get at a certain moment. It is also a tuple containing both memory and CPU resource quantities, which can be calculated as follows:

$$PB_{i,t} = min(HR_{i,t} + RR_{i,t}, CR_t) \tag{2}$$

where $PB_{i,t}$ denotes the parallelism bound of J_i. $HR_{i,t}$ is the quantity of resources holded by J_i. $RR_{i,t}$ represents the quantity of resources that are requested but not acquired by J_i. CR_i represents the total resource quantity of the cluster at time t.

Even if we ignore the finiteness of the cluster resources, the quantity of resources required by a job might be changing throughout its lifespan. Therefore, the parallelism bound of a job could also be changing all the time. Hence, we introduce *average parallelism bound (APB)* to denote the average resource quantities occupied by the job during its execution under no external resource contention. The *APB* of a job could be calculated as follows:

$$APB_i = \frac{S_i}{minE_i} \tag{3}$$

where APB_i denotes the average parallelism bound of J_i. S_i is the size of job J_i. $minE_i$ represents the ideal execution time of J_i, i.e., the execution time of J_i without external resource competitions.

$$minE_i = e_i^m \times ceil\left(\frac{R_i^m \times N_i^m}{CR}\right) + e_i^r \times ceil\left(\frac{R_i^r \times N_i^r}{CR}\right) \tag{4}$$

where $R_i^m(R_i^r)$ denotes the container resource requirements of one map (reduce) task in J_i. $e_i^m(e_i^r)$ represents the average execution time of map (reduce) tasks in J_i. $N_i^m(N_i^r)$ is the count of all map (reduce) tasks in J_i. CR stands for the total resource quantity in the cluster. In this paper, we use APB to supplement the calculation of jobs virtual sizes.

E. Fairness

For MapReduce jobs and many other batch processing or interactive analysis applications, response time is a major concern. Hence, we follow the definition of fairness in [17] that a protocol P is fair if no job completes later under P than under Fair scheduler. According to this definition, Fair scheduler is fair in nature, and FSP is also proved to be fair.

III. FSPY SCHEDULING ALGORITHM

FSPY is a size-based scheduler designed for Hadoop YARN. Inspired by FSP, FSPY assigns higher priority to jobs that has smaller virtual size. To track job virtual sizes at each time point, our scheduler uses a new mechanism for job size prediction and a couple of light-weight algorithms for job virtual size calculation. In addition, FSPY adopts a new scheduling framework to integrate these technologies.

A. MapReduce Job Size Prediction

The programming frameworks on YARN are such different from each other that no one model is capable to predict job sizes for all these frameworks. In this work, we do the prediction for the widely used MapReduce framework. Our prediction method is inspired by [18], where the job size prediction models are built by job feature analysis and weighted linear regression. Considering the characteristics of YARN, we present a new job features analysis mechanism and a new prediction model.

First of all, we introduce our job size prediction models. The size of a MapReduce job is the sum of all its map and reduce task sizes, as in (5).

$$S = R_m \times SE_m + R_r \times SE_r \tag{5}$$

where S denotes the job size. $R_m(R_r)$ denotes the resource quantity of a map (reduce) task container. $SE_m(SE_r)$ denotes total execution time of all map (reduce) tasks in the job. R_m and R_r can be obtained from the job's configuration file directly, while SE_m and SE_r should be estimated.

As stated in [18], the per-task execution time of a MapReduce job can be estimated by a couple of linear models. We make several extensions of the previous models. Firstly, per-task execution time is the prediction object of previous models, while total task execution time in each job phase is the prediction object of our models. This adjustment makes job parallelism considered in the prediction, which can enhance the prediction precision. Secondly, the execution times of map and reduce function (i.e., MFE and RFE) occupy main parts of the execution time of corresponding tasks. The length of these periods can be obtained by our job features analysis mechanism. Hence, they are involved as parameters in our models. Moreover, in general, some overlap exists between map and reduce phases of a job, which makes some early launched reduce tasks have to wait until all map tasks are finished. Therefore, the total execution time of map tasks is involved as parameter in our prediction model for total execution time of reduce tasks. Our prediction models for SE_m and SE_r are stated respectively in (6) and (7).

$$\begin{aligned} SE_m = {} & \alpha_0 + \alpha_1 \times N_m + \alpha_2 \times MI + \alpha_3 \times MFE + \alpha_4 \\ & \times MO + \alpha_5 \times MOR + \alpha_6 \times MOR \times logMOR \end{aligned} \tag{6}$$

$$\begin{aligned} SE_m = {} & \alpha_0 + \alpha_1 \times N_m + \alpha_2 \times MI + \alpha_3 \times MFE + \alpha_4 \\ & \times MO + \alpha_5 \times MOR + \alpha_6 \times MOR \times logMOR \end{aligned} \tag{7}$$

where $N_m(N_r)$ denotes the number of map (reduce) tasks in the job. MI denotes the total input quantity of map tasks in the job. $MFE(RFE)$ denotes the total execution time of map (reduce) function in the job. $MO(RO)$ denotes the total output quantity of map (reduce) function in the job. MOR denotes the total number of map output records in the job. RIR denotes the total number of reduce input records in the job. $\alpha_0, \alpha_1, ..., \alpha_6$ and $\beta_0, \beta_1, ..., \beta_7$ are the coefficients. Note that MFE is measured when all the map tasks have their input data in local. In practice, the input data of some map tasks is not stored in local and should be downloaded from HDFS during execution of the map function. With the number of map tasks increasing, the percentage of local map tasks will rise to a certain level and become stable [19]. Therefore, we use $\alpha_2 \times MI + \alpha_3 \times MFE$ to denote the real total execution time of map function in the job.

Next, we discuss how to apply these models to practice. In prediction models for SE_m and SE_r, some parameters (e.g. MFE, RFE) cannot be obtained directly. To estimate these parameters, we present a new job feature analysis mechanism with probing job. A probing job is a job that executes the same map and reduce functions with the original job but only processes a small fraction of the total input. During the execution of a probing job, job features of our interests (e.g. MFE, RFE) are collected by the counter mechanism in the MapReduce framework. The counter mechanism provides an approach to track a variety of operations in a MapReduce job. After job features are collected from the probing job, we can calculate corresponding parameters of the original job as follows.

$$P_o = \frac{P_p}{f_p} \tag{8}$$

where P_o denotes a parameter of the original job. P_p denotes the corresponding parameter collected from the probing job. f_p denotes the sampling fraction of input data.

However, the implementation of job probing is faced with many challenges. First of all, probing jobs should consume as less resource and time as possible without reducing the prediction precision. Secondly, the input data of a job may be skewed, which can affect probed parameters [20]. To alleviate the effect of data skew, the input data of probing jobs should be sampled randomly. However, a completely random sampling may be time consuming. Next, the map function execution time can be affected by data locality. In current MapReduce framework, the reading and processing of map function input data are executed in parallel. A map function with input data in local disk generally executes faster than that with input data on other servers. As our prediction models depend on the map function execution time with data in local, the sampled data should be downloaded to local disk before starting the map function. This means that we need to reconstruct the existing MapReduce framework.

To save resource and time consumed in the probing process, a probing job is set as a tiny job with just one map task and one reduce task in our implementations. Furthermore, we make the probing job run in uber mode in which all processes of a job share just one container. In contrast to running the probing job in normal mode, this setting can reduce the consumed resource and time significantly. If the original job is already in uber mode, we directly estimate its size using historic average task execution time. The uber mode is a marker of the tiny job in MapReduce. In size-based scheduling, the job size prediction precision for tiny jobs has far less influences than that for large jobs. Thus, it is rational to use simpler prediction methods for tiny jobs. Besides, as tiny jobs generally account for a large proportion of the total job set [21], this strategy can save lots of resources and time.

We present a semi-random sampling algorithm for the probing job, as shown in Algorithm 1. The input parameters

of this algorithm include splits (input data splits of the original job), mapperCount (map tasks count of the original job), reducerCount (reduce tasks count of the original job), and sampledSplitsUpperBound (maximum number of splits that can be sampled in this routine). The output is the dataset sampled from the splits. Firstly, we set the sampling fraction to be the reciprocal of the larger number between the counts of map and reduce tasks in the original job. This setting can prevent either the map or reduce task in a probing job from crashing under a too heavy workload. Then, the number of splits to be sampled and the sampling frequency of records in each split are calculated. To reduce consumed time, we do the sampling on a number of splits (not more than sampledSplitsUpperBound) selected randomly from the original splits. Finally, data is sampled from the selected splits according to the sampling frequency.

Algorithm 1: Data Sampling

Input: splits, mapperCount, reducerCount, sampledSplitsUpperBound
Output: sampledData

```
sampleFrac = 1/ max(mapperCount, reducerCount);
sampleSplitNum = min(mapperCount, sampledSplitsUpperBound);
splitSampleFreq = sampleFrac / (sampleSplitNum / mapperCount);
sampledData = an empty collection;
for j = 0;j ≤ mapperCount;j ++ do
    j = random(0, N_i^m);
    temSplit = splits[i];
    splits[i] = splits[j];
    splits[j] = temSplit;
for j = 0;j ≤ sampleSplitNum;j ++ do
    sampleBytes = splits[i].length × splitSampleFreq;
    Read sampleBytes of data from splits[i] and append to sampledData
return sampledData;
```

Moreover, we develop a new version of MapReduce framework, named probing MapReduce, specially for the probing job. In the probing MapReduce framework, the procedure of map task is adjusted and some new counters are added. The map task execution procedure contains the data sampling routine. After the sampling routine is finished, the map function starts to process the data sampled. Native counters in the MapReduce framework cannot collect enough features, we add some new counters to the framework. The new added counters include the counter of map function execution time and the counter of reduce function execution time.

The coefficients in our models can vary in different clusters or different period. Thus, we doesnt give any fixed values here. These coefficients can be learned by linear regression on history traces. We use generalized least squares [22] method to do the regression analysis. When a job running under normal mode is finished successfully, its probed parameters (e.g. MFE, RFE, MO) and actual parameters (e.g. SE_m, SE_r) will be added to the data set for linear regression analysis. After that, the updated models will be used to predict the sizes of subsequent jobs.

B. Job Virtual Size Calculation

Job virtual size is the most important parameter in our scheduling. The procedure we designed to calculate job virtual size is composed of two algorithms: the Virtual Resource Allocation algorithm (VRA) and the Virtual Size Calculation algorithm (VSC). When a job is submitted or has a virtual size of 0, VSC should be run to update virtual size of each job, and then VRA should be run to update the allocated virtual resource of each job.

According to the principle of Fair scheduler, computing resources should be equally allocated to current jobs. However, in practice, the required resource quantity varies a great deal among different jobs. The fair share may be redundant for some jobs but inadequate for some others. To avoid wastes of resource, the idle resource of some jobs should be reallocated to others whose requirements are not satisfied. Therefore, the actual allocations are not equal among jobs. To calculate the allocation of each job under Fair scheduler, we must consider the differences of jobs requirements. As the requirement of a job usually fluctuates during the job's life, it is hard to find out jobs requirements at each time point. Instead, we use average parallelism bound (APB) to represent the mean required resource quantity all over the job's life. Furthermore, we calculate jobs fair share in accordance with the dominant resource fairness principle [23].

For simplicity of description, jobs scheduled by the virtual scheduler is called *virtual jobs*. Jobs that has been submitted but not finished are called *pending virtual jobs*. All pending virtual jobs reside in the *virtual queue*, where jobs are ordered by their APBs. Note that the APB contains two kinds of resources, namely CPU and memory. Ordered by quantities of different resources in the APBs, the order of jobs is commonly different. According to the common practice in Hadoop [24], we order the virtual queue by the memory quantities of the APBs.

Algorithm 2 describes the procedure of VRA. The input parameters of this procedure include the virtual queue and the total resource quantity of the cluster. In general, APBs of jobs at the head of the virtual queue would be less than the fair share, while that of jobs at the tail of the queue would be more than the fair share. Virtual resources are allocated to the jobs in order. The unused resources in the fair share of a job will be allocated to its subsequent jobs in

the virtual queue. After the execution of this procedure, the virtual resource quantities of each job in the virtual queue are updated.

Algorithm 2: VIRTUAL RESOURCE ALLOCATION

```
Input: virtualQueue, clusterResource
Output: virtualQueue
remainingResource = clusterResource;
for j = 0;j ≤ virtualQueue.length;j ++ do
    job = virtualQueue[i];
    fairShare = remainingResource/(virtualQueue.length − i);
    if fairShare > job.APB then
        job.virtualResource = job.APB;
    else
        feedPercentage = fariShare/job.APB;
        job.virtualResource = job.APB × feedPercentage;
    remainingResourceC = job.virtualResource;
return virtualQueue;
```

The VSC algorithm depends on the last running result of the VRA algorithm. Algorithm 3 describes the procedure of VSC. The input parameters of VSC include virtual queue and lastUpdateTime, i.e., the last time when VSC was executed. During the interval of two adjacent update, the virtual size variance of a job is the product of its allocated virtual resources and the interval. Thus, current virtual size of the job is the result of its last virtual size subtracting the variance. One thing to note here is that, for jobs not in uber mode, the resources of their AM containers are eliminated from the resources used to calculate the virtual size. As the AM process is in charge of job management rather than doing specific processing works, its execution time is almost equal to the execution time of its host job and can be affected significantly by its competition circumstances and the scheduling policy. Thus, the size of an AM process should not be counted to the job size. After the execution of VSC, virtual sizes of each job in the virtual queue are updated.

C. FSPY Scheduling Framework

We have introduced the techniques replied on by FSPY. To integrate these techniques into FSPY, some problems must be addressed. The first is how to allocate resources among probing and original jobs. To obtain job sizes quickly, probing jobs should be allocated enough resources. However, if too much resources are allocated to probing jobs, the progresses of original jobs might be slowed down. Thus, a proper resource allocation scheme should be established among these jobs. Next, the probing process for job features can be time consuming (usually from a few seconds to tens

Algorithm 3: VIRTUAL SIZE CALCULATION

```
Input: virtualQueue, lastUpdateTime
Output: virtualQueue
interval = lastUpdateTime C currentTime;
for job in virtualQueue do
    if job.virtualResource > 0 then
        if job is in uber mode then
            job.virtualSizeC = job.virtualResource × interval
        else
            effectiveResource = job.virtualResourceCjob.amResource
            job.virtualSizeC = effectiveResource × interval
        if job.virtualSize <= 0 then
            remove job from virtualQueue
        else
            continue
    else
        continue
return virtualQueue;
```

of seconds). If a job is allocated no resources before its size is predicted, its finish time might be delayed, which can affect the responsiveness and fairness of the scheduler. Thus, how to allocate resources to jobs without size information is a problem.

We present a new scheduling framework to address the problems stated above. In our scheduling framework, cluster resources are allocated to two job queues, named *probing queue* and *production queue*, according to a specified fraction. Probing jobs are placed in the probing queue, while original jobs are placed in the production queue. The resource fraction for each queue can be configured in accordance with specific conditions. Furthermore, the idle resources in one queue can be allocated to the other queue according to the policy of Capacity scheduling. Jobs in the probing queue are scheduled by FIFO. The scheduling policy in the production queue is more sophisticated and will be introduced in the follows. Unless otherwise specified, jobs in the following part of this paper refer to the original jobs rather than the probing jobs.

To address the problem of probing latency, we schedule jobs by the fair policy until their sizes are predicted. Consequently, jobs in the production queue are further arranged into two sub queues: *size-based queue* and *fair queue*. Jobs with size information are placed in the size-based queue and are ordered by their virtual sizes, while jobs without size information are placed in the fair queue and are ordered by the quantities of their allocated resources. Now that both the job virtual size and the allocated resources involve two kinds

of resources in YARN, we use memory seconds and memory quantities instead respectively. Moreover, virtual jobs are scheduled in the virtual queue, which has as much virtual resources as the real resources in the cluster. Jobs in the virtual queue are ordered by the memory quantities of their APBs.

As jobs in the production queue are scheduled in two sub queues, how to allocate resources between these two sub queues is a key problem. According to the fair principle, jobs in the fair queue should hold as much resources as they would hold under the virtual Fair scheduler. In other words, their allocated real resources should be as much as their allocated virtual resources. If the fair queue holds less resources than the allocated virtual resources of its jobs, the scheduler will allocate new available resources to the fair queue. Otherwise, new available resources should be allocated to the size-based queue. However, the resources might be rejected by the selected queue for certain reasons, such as the requirement of the queue is fulfilled or the resources dont meet its requirements. Then, the scheduler will try allocate the resources to the other queue to enhance the resource utilization of the cluster.

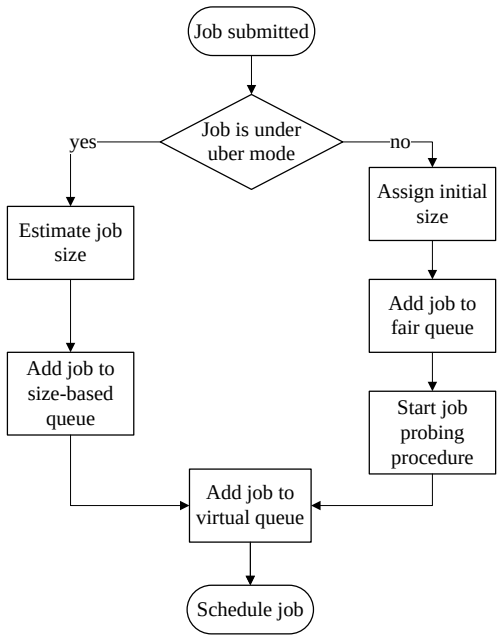


Figure 1. SSD Architecture

The procedure to handle a newly-submitted job is given in Figure 1. If the job is running in uber mode, its size is estimated directly as stated above, and then it is put into the size-based queue. If the job is in normal mode, we assign the job an initial size and put it into the fair queue. Then, we start the job probing procedure to predict its job size. Meanwhile, both jobs in uber and normal modes are put into the virtual queue. The initial size of a job should be large enough, otherwise the job may be taken out from the virtual queue by VSC algorithm before it is actually finished in the virtual Fair scheduler. In addition to job size, APB is also necessary in the virtual size calculation. The APB of an uber job is equal to the resource quantity of its AM container, while that of a normal job cannot be calculated until the job size is predicted. Similar to the operation of job size, we assign an initial APB to each normal job before its size is predicted. Now that a job usually runs in the map phase during its probing process, we calculate its initial APB as follows:

$$APB_{init} = min(R_{am} + R_m \times N_m, RC) \tag{9}$$

where APB_{init} denotes the jobs initial APB. R_{am} denotes the resource quantity in the jobs AM container. R_m denotes the resource quantity in a map container of the job. N_m denotes the number of map tasks in the job. RC denotes the total resource quantity in the system.

After the job probing process is finished, job size and APB are obtainted. As the current job virtual size is calculated based on initial job size, it should be updated with the newly-predicted job size. Before calculating the new virtual size, VSC algorithm should be run to update virtual sizes of jobs in the virtual queue. Then, new virtual size of the job is calculated as follows:

$$VS_{new} = S_{pred} - (S_{init} - VS_{cur}) \tag{10}$$

where VS_{new} is the new virtual size of the job. VS_{cur} is the current virtual size of the job. S_{prob} is the predicted job size. S_{init} denotes the initial job size. After updating the virtual size, the job is moved from the fair queue to the size-based queue. Finally, the allocated virtual resources of each job in the virtual queue are updated by the VRA algorithm.

When a job is finished, our handling procedure varies by which queue the job belongs to. If the job is in the fair queue, it means that this job is scheduled under fair scheduling principle throughout its lifespan. Then, the corresponding virtual job should be finished at the same time with the real one. As a result, the job should be removed from both the fair queue and the virtual queue. After the job is removed from the virtual queue, the VSC and VRA procedures should be executed in sequence to update the virtual sizes and allocated virtual resources of jobs in the virtual queue. Else, if the job is in the size-based queue, it should only be removed from the size-based queue. Then, the parameters of the job (both real parameters and the probed parameters) are collected to train the prediction models.

IV. Experiments

FSPY has been implemented and experimentally assessed. In this section, we provide the experiment details and analyze the results. Our experiments assess the performance of both our job size prediction mechanism and FSPY sched-

uler as a whole. To evaluate scheduling performance, we compare both fairness and responsiveness of FSPY to the Fair scheduler, which is commonly used in current Hadoop platform. Similar to the relevant works in [15], [25], the FIFO scheduler will not be compared with FSPY in our experiments, as it is demonstrated to be far less efficient than other schedulers.

A. Experimental Setup

We use three physical servers to build an experimental cluster. A physical server with 8GB RAM and 2 CPU cores is set as the master node of our cluster. The other two servers (each with 32GB RAM and 32 CPU cores) are virtualized to a resource pool by XenCenter. Then, we create 8 virtual servers (each with 8GB virtual RAM and 8 virtual cores) from the resource pool and set them as slave nodes. We deploy the Hadoop platform on our cluster and set the HDFS block size to 64MB.

We use SWIM [26] to synthesize the workload used in our experiments. SWIM can capture rich workload characteristics observed in traces, and synthesize representative workloads for execution. By default, SWIM provides two formatted workload traces, namely FB-2009 and FB-2010, both of which are sampled and converted from real workload trace files of FaceBooks datacenter. Comparing to FB-2009, jobs in FB-2010 are larger and submitted in a denser manner. Since the capacity of our experimental cluster is limited, we select FB-2009 to synthesize workloads in our experiments.

B. Job Size Prediction Validation

Figure 2 compares the predicted and actual per-job map execution times in our experiments. As the figure shows, most jobs are distributed intensively around the diagonal. We also notice that a portion of jobs are located under the diagonal in the lower left corner of the plot, which means that for tiny jobs (per-job map execution time less than 30s), our predicted values are generally smaller than the actual values. Statistically, the R2 accuracy of our prediction reaches 0.942. The whole results suggest that our method is effective in per-job map execution time prediction, especially for large jobs.

Figure 3 shows the predicted vs. actual total per-job reduce execution times of jobs in our experiments. Apparently, the distribution of points in this figure is less intensive than that in Figure 2, which means the prediction accuracy for reduce tasks is lower than that for map tasks. Nonetheless, the R2 accuracy of per-job reduce execution time prediction is still up to 0.918. Thus the overall prediction accuracy of reduce task execution time is also quite good.

Figure 4 compares the predicted and actual job sizes in our verification. The R2 accuracy reaches a high value of 0.970, which exceeds the prediction accuracies of both map and reduce task execution time. We notice that, generally, the predicted map-task execution times of small jobs are lower than the actual values, while the predicted reduce tasks execution times of small jobs are higher than the actual values. The higher accuracy of job size prediction might result from the offset of prediction deviation of the map and reduce tasks execution time.

C. Validation of FSPY Performance

Responsiveness and fairness are the core aspects of scheduler performance we are concerned with in this paper. Therefore, these performance aspects of FSPY should be the evaluation focus in our experiments. According to the common practice in scheduling literature, we use job response time as the metric of responsiveness. In our experiments, the average value and empirical cumulative distribution function (ECDF) of job response times under FSPY are compared with that under the Fair scheduler.

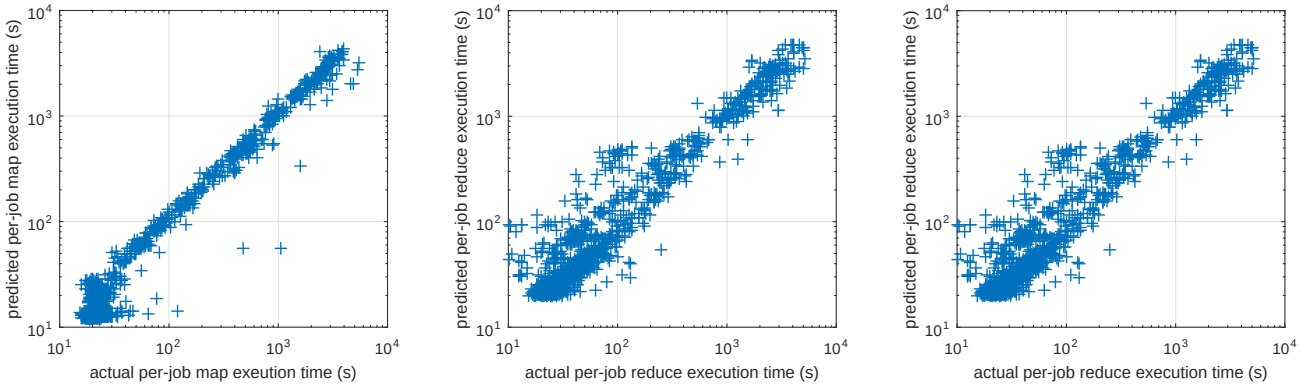
While for the evaluation of fairness, there are different metrics with different focuses. One of the commonly used metrics of fairness is job slowdown, i.e., the ratio between job's response time under a specific scheduler and its ideal response time in the system without sharing resources with other jobs. We refer to the commonly used job slowdown as *common slowdown* in this paper. Besides, we also defined another metric, namely *fair slowdown*, to measure the fairness. Now that the fairness definition in this paper is based on the performance of Fair scheduler, the fair slowdown is defined as the ratio between job's response time under a specific scheduler and that under the Fair scheduler. In our experiments, both the common slowdown and fair slowdown are used to evaluate the fairness of FSPY.

As the scheduler performances are affected by the resource competition pressures (RCP) in the system, we should compare the performances of schedulers respectively under different resource competition pressures. In this paper, the RCP of a workload is defined as follows:

$$RCP = \alpha \times log\left(\frac{\sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} I_i \times CR}\right) + \beta \qquad (11)$$

where S_i denotes job size of J_i. I_i is the time interval between submissions of J_{i-1} and J_i. n denotes the number of jobs involved in the workload. CR denotes the total resource quantity in the system. α and β are coefficients to make the result approximate to an integer.

To get workload with different RCPs, we extract a series of continuous segments from FB-2009 and run the workload built from each segment separately. The FB-2009 is composed of 5894 independent jobs. Each segment contains 200 jobs and adjacent segments have an overlap of 50 jobs. In consequence, we get 37 segments from the FB-2009. Each segment is used to synthesize an executable workload. And then, each synthesized workload is run under the FSPY and Fair scheduler respectively. After the execution, the RCP of each workload, the job response times under the FSPY and Fair scheduler are calculated.



(a) Predicted vs. actual values of the total map tasks execution time

(b) Predicted vs. actual values of the total reduce tasks execution time

(c) Predicted vs. actual values of the total reduce tasks execution time

Figure 2. Two Subfigures

The workloads in our experiments are classified by RCP into 7 groups. Figure 5 shows the number of trace segments (workloads) in each RCP group. The group with RCP of 5 has more workloads than all the others. As the RCP rises or falls from 5, the number of workloads in the group decreases. The number of workloads with RCP that is greater than or equal to 6 is up to 19, accounting for more than 50% of the total.

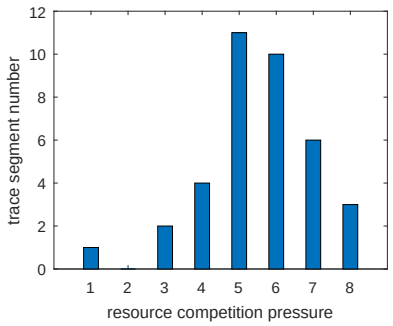
1) Responsiveness: The comparisons of job average response times under the FSPY and Fair scheduler are given in Figure 6. As shown in the figure, when RCP is less than or equal to 3, the average response time of jobs under the FSPY scheduler and that under Fair scheduler are almost equal. The reason lies in the fact that when the resources are abundant to the workload, every job can get resources timely and the scheduler will have few effects on the performance. When RCP is greater than or equal to 6, the average response time of jobs under the FSPY scheduler is dramatically less than that under the Fair scheduler. Furthermore, as the RCP rises from 6, the gap between FSPY and Fair scheduler widens sharply. Therefore, FSPY is more efficient than the Fair scheduler in terms of job response time, especially when the workload is under a high RCP.

Figure 7 shows the ECDF of per-job response times in the group with RCP of 6. The two curves in this figure almost coincide when job response times are less than 15s because that the tiny jobs are generally given the highest priority under both FSPY and the Fair scheduler. In the middle area of the figure, there is a significant gap between the curves of FSPY and Fair scheduler. The ECDF under FSPY exceeds 0.8 at the response time of 40s, which means more than 80% of jobs under FSPY have a response time below 40s, while the percentage under the Fair scheduler is less than 50%. It indicates middle-size jobs generally have a shorter response time under FSPY than under the Fair scheduler. Moreover, when job response time exceeds 64s, the two curves coincided again, which means that large jobs have similar response times under these two schedulers.

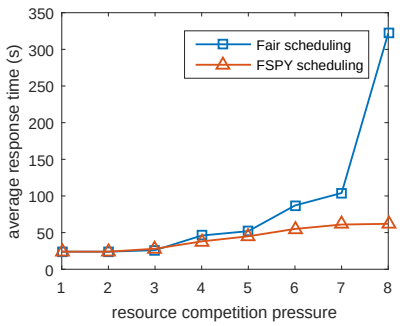
2) Fairness: We use the common slowdown and fair slowdown as metrics to evaluate the fairness of FSPY. Figure 8 shows the ECDF of job common slowdown in the group with RCP of 10. On the whole, the job common slowdown under FSPY is less than that under the Fair scheduler. The percentage of jobs under FSPY with a common slowdown less than 4 exceeds 95% while it is less than 65% under the Fair scheduler. Moreover, the maximum of job common slowdown under FSPY (about 5) is significantly less than that under the Fair scheduler (about 10).

Figure 9 shows the ECDF of fair slowdown in the group with RCP of 6. As shown in this figure, jobs with a fair slowdown less than or equal to 1 account for more than 75% of the total. This indicates that more than 75% of jobs under FSPY have a response time no longer than that they would have under the Fair scheduler. Besides, more than 98% of jobs under FSPY have a response time less than 1.5. And the maximum of fair slowdown under FSPY in the experiment approximates 1.7. There are still some jobs whose fair slowdown exceeds 1 under FSPY, which can be attributed to a couple of factors. One is the job size prediction error, which can be reduced but never eliminated. Another is that the job size itself can be fluctuant to certain extent from one run to another.

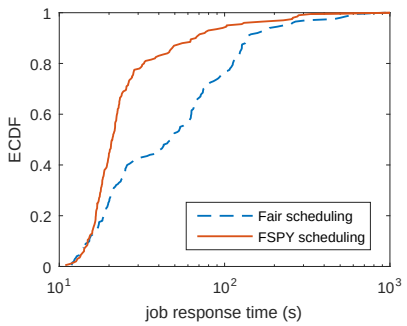
Figure 10 shows the comparisons of job response times under FSPY and the Fair scheduler. In the figure, most points are below the quadrant diagonal, indicating that most jobs have a shorter response time under FSPY than under the Fair scheduler. The points above the diagonal are distributed mainly at the bottom-left corner, which reconfirmed the poor precision in the size prediction of small jobs. Although, for small jobs, delay caused by the poor job size prediction precision is relatively short (generally a few seconds), thus can be tolerated in the scheduling. In summary, FSPY, on



(a) Trace segment numbers in each RCP group.

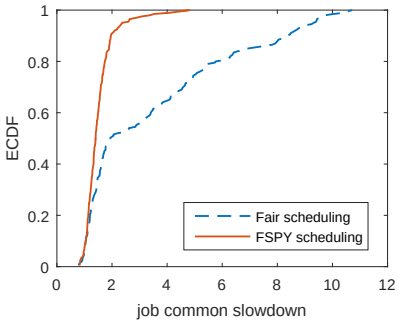


(b) The average response time of jobs in each RCP group.

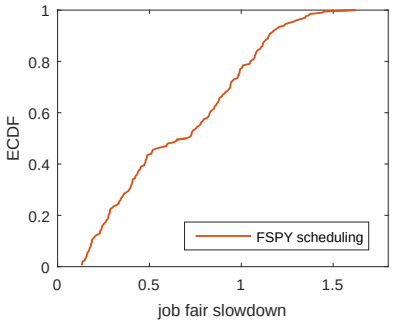


(c) ECDF of job response times under the RCP of 6.

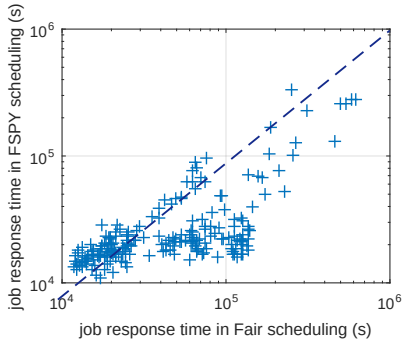
Figure 3. Two Subfigures



(a) ECDF of job common slowdown under the RCP of 6.



(b) ECDF of job fair slowdown under the RCP of 6.



(c) Job response times under FSPY vs. that under the Fair scheduler (RCP = 6).

Figure 4. Two Subfigures

the whole, can guarantee the fairness of resource scheduling in YARN.

V. Conclusion

In this paper, we present a new scheduler FSPY to improve the responsiveness and guarantee the fairness of the Hadoop YARN platform. The scheduler is designed based on the idea of size-based scheduling. As the job size is not known a priori on YARN, we design a job size prediction module to predict the whole job sizes of MapReduce jobs in a resource and time economic manner. Moreover, to guarantee the fairness of scheduling, FSPY is equipped with a novel job virtual size calculation algorithm.

We evaluate the performance of both the scheduler and our job size prediction module through a series of realistic experiments. Results show that the average job response time under FSPY is significantly shorter than that under Fair scheduler, especially when the system is under heavy workloads. Meanwhile, the scheduler guarantees the fairness between jobs as the job slowdown is restricted to a narrow scope. Our job size prediction method also achieves a satisfying precision.

In this work, we only focus on MapReduce programming framework. In a future work, we will provide job size prediction models for other programming models such as Tez and Spark.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, and S. Seth, "Apache hadoop yarn: Yet another resource negotiator," *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, 2013.

[3] J. Heer and S. Kandel, "Interactive analysis of big data," *XRDS: Crossroads, The ACM Magazine for Students*, vol. 19, no. 1, pp. 50–54, 2012.

[4] P. Zikopoulos, C. Eaton *et al.*, *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[5] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1357–1369.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[7] M. Hausenblas and J. Nadeau, "Apache drill: interactive ad-hoc analysis at scale," *Big Data*, vol. 1, no. 2, pp. 100–104, 2013.

[8] M. H. Iqbal and T. R. Soomro, "Big data analysis: Apache storm perspective," 2015.

[9] L. E. Schrage and L. W. Miller, "The queue m/g/1 with the shortest remaining processing time discipline," *Operations Research*, vol. 14, no. 4, pp. 670–684, 1966.

[10] N. Bansal and M. Harchol-Balter, *Analysis of SRPT scheduling: Investigating unfairness*. ACM, 2001, vol. 29, no. 1.

[11] M. Harchol-Balter, M. Crovella, and S. Park, "The case for srpt scheduling in web servers," Citeseer, Tech. Rep., 1998.

[12] Z. Guo, M. Pierce, G. Fox, and M. Zhou, "Automatic task re-organization in mapreduce," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 2011, pp. 335–343.

[13] M. Lin, L. Zhang, A. Wierman, and J. Tan, "Joint optimization of overlapping phases in mapreduce," *Performance Evaluation*, vol. 70, no. 10, pp. 720–735, 2013.

[14] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011, pp. 289–298.

[15] M. Pastorelli, D. Carra, M. Dell'Amico, and P. Michiardi, "Hfsp: Bringing size-based scheduling to hadoop," 2015.

[16] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 87–92.

[17] E. J. Friedman and S. G. Henderson, "Fairness and efficiency in web server protocols," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1. ACM, 2003, pp. 229–237.

[18] G. Song, Z. Meng, F. Huet, F. Magoules, L. Yu, and X. Lin, "A hadoop mapreduce performance prediction method," in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 2013, pp. 820–825.

[19] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 419–426.

[20] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Handing data skew in mapreduce," in *Proceedings of the 1st International Conference on Cloud Computing and Services Science*, vol. 146, 2011, pp. 574–583.

[21] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[22] N. Orsini, R. Bellocco, S. Greenland *et al.*, "Generalized least squares for trend estimation of summarized dose-response data," *Stata Journal*, vol. 6, no. 1, p. 40, 2006.

[23] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, vol. 11, 2011, pp. 24–24.

[24] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[25] Y. Li, C. Lin, F. Ren, and Y. Geng, "H-pfsp: Efficient hybrid parallel pfsp protected scheduling for mapreduce system," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*. IEEE, 2013, pp. 1099–1106.

[26] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE, 2011, pp. 390–399.