2012

Real-Time Scheduling in MapReduce Clusters

Chen He
University of Nebraska-Lincoln, che@cse.unl.edu

Ying Lu
University of Nebraska-Lincoln, ying@unl.edu

David Swanson
University of Nebraska-Lincoln, dswanson@cse.unl.edu

Real-Time Scheduling in MapReduce Clusters

Chen He Ying Lu David Swanson
Computer Science & Engineering Department, University of Nebraska-Lincoln
Lincoln NE, United States
{che, ylu, dswanson}@cse.unl.edu

Abstract **MapReduce has been widely used as a Big Data processing platform. As it gets popular, its scheduling becomes increasingly important. In particular, since many MapReduce applications require real-time data processing, scheduling real-time applications in MapReduce environments has become a significant problem. In this paper, we create a novel real-time scheduler for MapReduce, which overcomes the deficiencies of an existing scheduler. It avoids accepting jobs that will lead to deadline misses and improves the cluster utilization. We implement our scheduler in Hadoop system and experimental results show that our scheduler provides deadline guarantees for accepted jobs and achieves good cluster utilization.**

Keywords: real-time scheduling; MapReduce; cluster utilization

I. INTRODUCTION

MapReduce is a framework used by Google for processing huge amounts of data in a distributed environment [1] and Hadoop [2] is Apache's open source implementation of the MapReduce framework. Due to the simplicity of the programming model, MapReduce is widely used for many applications [9]. Event logs from Facebook's website are imported into a Hadoop cluster every hour, where they are used for a variety of applications, including analyzing usage patterns to improve site design, detecting spam, data mining and ad optimization [3]. The New York Times rents a Hadoop cluster from Amazon EC2 [9] to conduct large scale image conversions [9]. Hadoop is also used to store and process tweets, log files, and many other types of data generated across Twitter [9]. As MapReduce clusters get popular, their scheduling becomes increasingly important. Yahoo! developed the capacity scheduler to share a Hadoop cluster among multiple groups and users [10]. Facebook's fair scheduler enabled fair sharing in MapReduce [3]. In particular, since many MapReduce applications [9], including some of the aforementioned ones (e.g., online data analytics for spam detection and ad optimization), require real-time data processing, scheduling real-time applications in MapReduce environments has become a significant problem [11][12][13][18][19] [20].

Polo et al. [11] developed a soft real-time scheduler that allows performance-driven management of MapReduce jobs. Dong et al. [13] extended the work by Polo et al., where a two-level MapReduce scheduler was developed to schedule mixed soft real-time and non-real-time jobs according to their respective performance demands. Although taking MapReduce jobs' QoS into consideration, most existing approaches [11] [13][18][19][20] do not provide deadline guarantees for the jobs. Kc and Anyanwu [12] developed a Deadline Constraint scheduler, aiming to provide time guarantees for MapReduce jobs. However, the Deadline Constraint scheduler has several deficiencies, which may lead to not only resource underutilization but also deadline violations (please refer to Section III for detailed analysis).

This paper develops a novel Real-Time MapReduce (RTMR) scheduler to not only provide deadline guarantees for MapReduce applications but also ensure good utilization of MapReduce clusters. The remainder of this paper is organized as follows. Section 2 presents the background. In Section 3, we briefly describe the Deadline Constraint scheduler [12] and its deficiencies. Section 4 presents our new scheduling algorithm in detail. Evaluations of these two schedulers are provided in Section 5. Section 6 concludes the paper.

II. BACKGROUND

In this section, we briefly describe how a Hadoop cluster works since other MapReduce-style clusters work similarly. In later parts of this paper, we will thus use the terms "Hadoop cluster" and "MapReduce cluster" interchangeably. A Hadoop cluster is often composed of many commodity PCs, where one PC acts as the master node and others as slave/worker nodes. A Hadoop cluster uses Hadoop Distributed File System (HDFS) [14] to manage its data. It divides each file into small fixed-size (e.g., 128 MB) blocks and stores several (e.g., 3) copies of each block in local disks of cluster machines. A MapReduce [1] computation is composed of two stages, map and reduce, which take a set of input key/value pairs and produce a set of output key/value pairs. When a MapReduce job is submitted to the cluster, it is divided into M map tasks and R reduce tasks, where each map task will process one block of input data.

A Hadoop cluster uses worker nodes to execute map and reduce tasks. There are limitations on the number of map and reduce tasks that a worker node can accept and execute simultaneously (i.e., map and reduce slots). Periodically, a worker node sends a heartbeat signal to the master node. Upon receiving a heartbeat from a worker node that has empty map/reduce slots, the master node invokes the MapReduce scheduler to assign tasks to the worker node. A worker node that is assigned a map task reads the content of the corresponding input data block from a local or remote disk, parses input key/value pairs out of the block, and passes each pair to the user-defined map function. The map function generates intermediate key/value pairs, which are buffered in memory, and periodically written to the local disk and divided into R regions by the partitioning function. The locations of these intermediate data are passed back to the master node, which is responsible for forwarding these locations to reduce tasks. A reduce task uses remote procedure calls to read the intermediate data generated by the M map tasks of the job. Each reduce task is responsible for a region (partition) of intermediate data with certain keys. Thus, it has to retrieve its partition of data from all worker nodes that have executed the M map tasks. This process is called shuffle, which involves many-to-many communications among worker nodes. The reduce task then reads in the intermediate data and invokes the reduce function to produce the final output data (i.e., output key/value pairs) for its reduce partition [1]. Figure I illustrates Hadoop framework and computation.

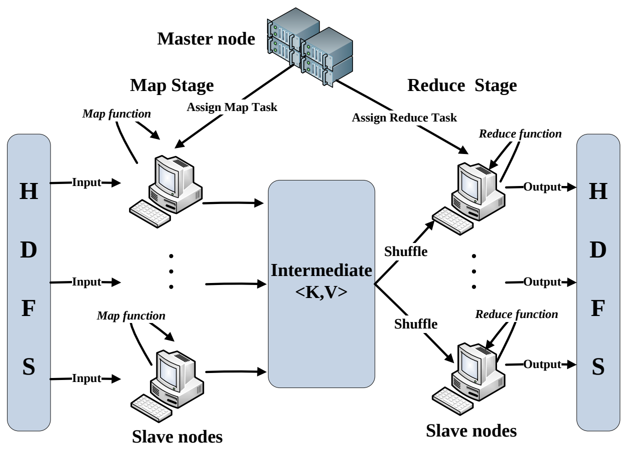

Fig 1. Hadoop Framework and Computation

III. Deadline Constraint Scheduler

The Deadline Constraint scheduler [12] aims to ensure deadlines for real-time MapReduce jobs. After a job is submitted, the scheduler first determines whether the job can be completed within the specified deadline or not using a schedulability test. It assumes that 1) a job's reduce stage does not start until the job's map tasks finish and 2) a job's reduce tasks all start execution simultaneously for the same amount of time that is known a priori. Based on these assumptions, it first calculates the latest start time s_r^{max} for a job's reduce stage, which is also the deadline for the job's map tasks. If the job arrives at time A, then the job has at most s_r^{max} - A amount of time to complete its map stage. Unlike for the reduce stage, the Deadline Constraint scheduler assumes that each job executes at a minimum degree of task parallelism for the map stage. That is, the scheduler only assigns the job the minimum number n_m^{min} of map slots that are required to meet its deadline. The scheduler, however, demands all n_m^{min} map slots to be available simultaneously at the job's arrival time.

Upon a job's submission, the constraint scheduler carries out the schedulability test. The job is rejected if n_m^{min} number of map slots are not available at that time. The job is also rejected if the number of reduce slots available at s_r^{max} is smaller than the total number of reduce tasks specified for the job.

The Deadline Constraint scheduler, however, has some limitations and deficiencies, which may lead to resource underutilization and deadline violations. First, because the scheduler assumes that all reduce tasks of a job start to run simultaneously, it cannot accept a job with more reduce tasks than the cluster's total number of reduce slots. Second, by checking the aforementioned two conditions in the schedulability test, the scheduler only considers a single scenario where the job's deadline might be satisfied. Those conditions are, however, unnecessary for meeting a job's deadline. Many jobs that do not pass the test can nevertheless be accepted and completed by their deadlines. For instance, even if the system does not have n_m^{min} number of map slots available upon the job's arrival, the job can still finish its map stage on time and meet the job's deadline if we have more resources available at a later time point. Furthermore, the constraint scheduler does not consider the case where slots become available and utilized at different time points. Due to these reasons, the Deadline Constraint scheduler rejects tasks unnecessarily and cannot well utilize system resources.

Last but not the least, the schedulability test conditions checked by the scheduler are insufficient to ensure the deadline constraint. As a result, accepted jobs may actually miss their deadlines, violating the scheduler's real-time property. The cause for the deadline violation is that the scheduler only checks if a certain number of reduce slots are available at a particular time point s_r^{max} . Instead, the job requires the specified number of reduce slots available for the whole time interval [s_r^{max} , D], where D is the job's deadline.

IV. RTMR Scheduler

In this paper, we develop a new Real-Time MapReduce (RTMR) scheduler for heterogeneous clusters. RTMR scheduler not only provides deadline guarantees to accepted jobs but also well utilizes system resources. We have made the following three assumptions when designing RTMR scheduler:

- The input data is available in Hadoop Distributed File System (HDFS) before a job starts.
- No preemption is allowed. The proposed scheduler orders the job queue according to job deadlines. However, once a job starts to execute its first map task, the job will not be preempted. That is, even if a new coming job B has an earlier deadline than a currently running job A, our scheduler makes no attempt to execute B's tasks before A's tasks.
- A MapReduce job contains two stages: map and reduce stages. Similar to [11][12][13], we assume that a job's reduce stage does not start until the job's map tasks have all finished.

RTMR scheduler is composed of three components. The first and most important one is the admission controller, which makes decisions on whether to accept or reject a job. The second component is the job dispatcher, which assigns tasks to execute on worker nodes. The last component is the feedback controller. Since a job may finish at a different time than estimated, a feedback controller is designed to keep the admission controller up-to-date.

A. Definitions

Before describing the algorithm, we first present the parameters and data structures used in RTMR scheduler.

- J=(A, D, M, R, δ): A MapReduce job J is specified by the tuple (A, D, M, R, δ), where A is the job arrival time, D is the relative deadline, M and R respectively specify the number of map and reduce tasks for the job, and δ is the input data size of the job. For a MapReduce job, each map task processes a unique part, δ_i^m , of the job's input data, where $\sum_{i=1}^{M} \delta_i^m = \delta$.
- η : the estimated maximum ratio between a job's intermediate data size δ^r and input data size δ. That is, the input data size δ^r for the job's reduce stage is at most $\eta * \delta$. For a MapReduce job, each one of the R reduce tasks processes a unique part, δ_i^r , of the job's intermediate data, where $\sum_{i=1}^{R} \delta_i^r = \delta^r$.
- c_m: the estimated time of retrieving and processing a unit of data in a map task.
- $c_m^{\max}$: the estimated longest time of retrieving and processing a unit of data in a map task. The time to retrieve data for a map task varies depending on where the input data is located (i.e., in memory, local disk, or remote disk). In addition, for a heterogeneous cluster, the task execution time differs on different nodes. $c_m^{\max}$ gives the worst-case estimation.
- c_r: the estimated time of retrieving and processing a unit of data in a reduce task.
- $c_r^{\max}$: the estimated longest time of retrieving and processing a unit of data in a reduce task.
- $J.T^m = \left[t_1^m, t_2^m, \ldots t_l^m\right]$: For each accepted job J, we maintain a sorted vector T^m to record the *estimated* available time of the cluster's map slots, after the scheduled execution of J and J's predecessors. In the vector, l denotes the total number of map slots in the MapReduce cluster.
- $J.T^r = \left[t_1^r, t_2^r, \ldots t_q^r\right]$: For each accepted job J, we maintain a sorted vector T^r to record the *estimated* available time of the cluster's reduce slots, after the scheduled execution of J and J's predecessors. In the vector, q denotes the total number of reduce slots in the MapReduce cluster.
- $J.V^m = [v_1^m, v_2^m, \ldots v_l^m]$: For each accepted job J, we use a sorted vector V^m to represent the *actual* available time of the cluster's map slots after considering the actual execution of J and J's predecessors.
- $J.V^r = [v_1^r, v_2^r, \ldots v_q^r]$: For each accepted job J, we use a sorted vector V^r to represent the *actual* available time of the cluster's reduce slots after considering the actual execution of J and J's predecessors.
- Δ: The threshold that we set for triggering the feedback controller. That is, if the difference of a job's actual and estimated finish times is larger than Δ, RTMR scheduler will invoke the feedback controller to keep the admission controller up-to-date.
- ε_i^m : the execution time of the ith map task of job J.
- ε_i^r : the execution time of the ith reduce task of job J.

RTMR scheduler uses historical job execution data to estimate some of the aforementioned parameters: η, $c_m^{\max}$, and $c_r^{\max}$. After executing a job J, we could update ratio η through the following equation:

$$\eta = \max(\eta, \frac{\delta^r}{\delta})$$

Similarly, we update the values of $c_m^{\max}$ and $c_r^{\max}$ as follows:

$$c_m^{\max} = \max(c_m^{\max}, \frac{\varepsilon_1^m}{\delta_1^m}, \frac{\varepsilon_2^m}{\delta_2^m}, \ldots \frac{\varepsilon_M^m}{\delta_M^m})$$

$$c_r^{\max} = \max(c_r^{\max}, \frac{\varepsilon_1^r}{\delta_1^r}, \frac{\varepsilon_2^r}{\delta_2^r}, \ldots \frac{\varepsilon_R^r}{\delta_R^r})$$

In a heterogeneous environment, worker nodes have different data retrieving and processing power. In order to avoid deadline miss, we follow the same mechanism as adopted by the Deadline Constraint scheduler [12] where the longest time of running a map/reduce task is used in the execution time estimation.

B. Admission Controller

In this paper, we assume, for both Deadline Constraint and RTMR schedulers, that jobs are put in a priority queue following EDF (earliest deadline first) order. Our admission control mechanism is, however, applicable beyond EDF, in general, to any policy (e.g., FIFO) that defines an order in which jobs should be given resources. When a new MapReduce job arrives, the admission controller determines if it is feasible to schedule the new job without compromising the guarantees for previously admitted jobs.

Algorithms I, II, and III show the pseudo code of the admission control. RTMR scheduler first checks if the new job J's deadline can be satisfied or not, i.e., to check if e ≤ A + D, where e is the estimated finish time of the job (Algorithm I lines 1-9). To estimate J's finish time, we start with identifying J's proceeding job J_p if J were inserted in the priority queue. If J were at the head of the queue, J_p is the job that has been started latest by the dispatcher. If J is the first job submitted to the cluster, it does not have a proceeding job. Since T_p^m and T_p^r record the estimated available time of the cluster's map and reduce slots after the scheduled execution of J_p and J_p's predecessors, we can estimate job J's finish time based on these vectors. If the new job J's deadline can be satisfied, RTMR scheduler then checks whether accepting J will violate the deadline of any previously admitted job (Algorithm I lines 10-21). Since only jobs that succeed job J in the priority queue will be delayed, RTMR scheduler re-estimates their finish times. If any of them will miss deadline as a result of J's acceptance, RTMR scheduler rejects job J. Finally, once the admission controller decides to accept job J, the priority queue and the T^m and T^r vectors of J and J's successors will be updated to reflect the change (Algorithm I lines 22-23).

ALGORITHM I. ADMISSION CONTROLLER

```
AC(J = (A, D, M, R, δ), Priority-Q)
  // Identifying J's proceeding job J_p if J were inserted in the queue
  1:  J_p = getPredecessor(J, Priority-Q)
  2:  $T_p^m$ = J_p.$T^m$ ($T_p^m$ = [0,0, …0] if J_p = nil)
  3:  $T_p^r$ = J_p.$T^r$ ($T_p^r$ = [0,0, …0] if J_p = nil)
  // invoke Algorithms II and III to do the calculation
  4:  $J.T^m$ = Cal$T^m$(J, $T_p^m$) . $T^m$
  5:  $J.T^r$ = Cal$T^r$(J, $T_p^m$, $T_p^r$).$T^r$
  6:  e = Cal$T^r$(J, $T_p^m$, $T_p^r$).e
  7:  if e > A + D then
  8:      return false
  9:  end if
  10: J_p = J
  11: J_s = getSuccessor(J_p, Priority-Q)
  12: while (J_s != nil) do
          // invoke Algorithms II and III to do the calculation
  13:         $T_s^m$ = Cal$T^m$( J_s, J_p.$T^m$ ) . $T^m$
  14:         $T_s^r$ = Cal$T^r$( J_s, J_p.$T^m$, J_p.$T^r$ ).$T^r$
  15:         e_s = Cal$T^r$( J_s, J_p.$T^m$, J_p.$T^r$ ).e
  16:         if e_s > J_s.A + J_s.D then
  17:             return false
  18:         end if
  19:         J_p = J_s
  20:         J_s = getSuccessor(J_p, Priority-Q)
  21: end while
  22: Proiority-Q.insert(J)
  23: record $J.T^m$, $J.T^r$, $T_s^m$ and $T_s^r$ computed above as the new $T^m$ & $T^r$ vectors for J and J's successors
  24: return true
```

ALGORITHM II. CACULATION OF T^m AND e^m

```
Cal$T^m$ (J = (A, D, M, R, δ), $T^m = \left[t_1^m, t_2^m, \ldots t_l^m\right]$)
  // This algorithm estimates $e^m$, job J's map stage finish time and $T^m$, the available time of map slots after the scheduled execution of J and J's predecessors
  1:  $\tilde{\varepsilon}^m = c_m^{\max} * \max(\delta_i^m, i = 1, 2, \ldots M)$
  2:  for k =1 to M do
  3:      pick the smallest value in vector $T^m$, i.e., $t_1^m$
  4:      $t_1^m$ = max($t_1^m$, current Time)
  5:      $t_1^m$ += $\tilde{\varepsilon}^m$
  6:      $e^m$ = $t_1^m$
  7:      sort items in $T^m$ to keep $T^m$ a sorted vector
  8:  end for
  9:  return $T^m$, $e^m$
```

ALGORITHM III. CACULATION OF T^r AND e

```
Cal$T^r$ (J = (A, D, M, R, δ), $T^m = \left[t_1^m, \ldots t_l^m\right], T^r = \left[t_1^r, \ldots t_q^r\right]$)
  // This algorithm estimates e, job J's finish time and $T^r$, the available time of reduce slots after the scheduled execution of J and J's predecessors
  // invoke Algorithm II to estimate J's map stage finish time
  1:  $e^m$ = Cal$T^m$(J, $T^m$).$e^m$
  2:  $\tilde{\varepsilon}^r = c_r^{\max} * \max(\delta_i^r, i = 1, 2, \ldots R)$
  3:  for k = 1 to R do
  4:      pick the smallest value in vector $T^r$, i.e., $t_1^r$
  5:      $t_1^r$ = max($t_1^r$, $e^m$)
  6:      $t_1^r$ += $\tilde{\varepsilon}^r$
  7:      e = $t_1^r$
  8:      sort items in $T^r$ to keep $T^r$ a sorted vector
  9:  end for
  10: return $T^r$, e
```

C. Dispatcher

As mentioned in Section II, a Hadoop cluster uses worker nodes to execute map and reduce tasks. Each worker node has a fixed number of map slots and reduce slots, which limit the number of map tasks and reduce tasks that a worker node can execute simultaneously. Periodically, a worker node sends a heartbeat signal to the master node. Upon receiving a heartbeat from a worker node with empty map/reduce slots, the master node invokes the scheduler to assign tasks. RTMR scheduler's dispatcher fulfills this role, allocating tasks to execute on worker nodes. Algorithm IV shows the pseudo code of the dispatcher.

When jobs are inserted into the priority queue, their map stages can start and their map tasks are ready to run. Therefore, it is straightforward to dispatch map tasks following the job order/priority. No modification is needed here and RTMR scheduler dispatches map tasks following the same approach as the default Hadoop system (lines 4-5).

However, since a job's map stage finish time depends on not only the job's map stage start time but also the number of map tasks the job has, when there are multiple jobs concurrently running in the cluster, which jobs can finish their map stages and start their reduce stages earlier is not determined by the job priority alone. Although jobs start their map stages following the job order/priority, it is highly likely that jobs will not finish their map stages in that order. As a result, the reduce tasks of a lower-priority job could become ready earlier than those of a higher-priority job. Thus, if ready reduce tasks are assigned to execute on worker nodes without any constraint, the proper execution of higher-priority jobs may be interfered by the execution of lower-priority jobs, leading to deadline violations. One simple method to avoid such interferences is to strictly enforce that jobs start their reduce stages following the job order. That is, a job cannot start the reduce stage until all proceeding jobs have finished their map stages. However, this straightforward method puts a strong constraint on job parallelism and causes inefficient utilization of system resources. Therefore, we instead design a reservation-based dispatcher, which simply ensures that a lower-priority job does not occupy slots that belong to higher-priority jobs. That is, the dispatcher reserves slots that are needed by higher-priority jobs to avoid potential interferences. Upon receiving a heartbeat from a worker node with empty reduce slots, the dispatcher assigns a reduce task to the worker node only if enough reduce slots have been left unused for higher-priority jobs (lines 6-21).

We have proved that all jobs accepted by the admission controller can be successfully dispatched and completed by their deadlines in normal scenarios when there is neither a node failure nor a task re-execution (please refer to the Appendix for the proof).

ALGORITHM IV. DISPATCHER

```
DP(J=(A, D, M, R, δ), Priority-Q,i,Ra)
 1:  m: available map slots on node i
 2:  r: available reduce slots on node i
 3:  Ra: the number of available reduce slots in the cluster, which is
     counted upon calling this algorithm
     // dispatch map tasks:
 4:  if (m>0) then
 5:      follow the same approach as the default Hadoop system to
         dispatch map tasks
     // dispatch reduce tasks:
 6:  if r > 0 then
 7:      reservedSlot: the number of reduce slots reserved for high-
         priority jobs
 8:      reservedSlot = 0
 9:      for J from Priority-Q do
10:          if reservedSlot > Ra then
11:              break for
12:          end if
13:          T = findAReadyReduceTask(J)
14:          if T != nil then
15:              assign T to node i
16:              break for
17:          else if J has not reached its reduce stage then
18:              reservedSlot += J.R
19:          end if
20:      end for
21:  end if
```

D. Feedback Controller

A feedback controller is developed to keep the admission controller up-to-date. As described in Section B, the admission controller makes decisions based on information maintained in job records, i.e., $J.T^m$ and $J.T^r$ vectors. These vectors record the estimated available time of the cluster's map and reduce slots after the scheduled execution of job J and its predecessors. However, these jobs' actual execution may be different from the estimate. For instance, due to the pessimistic estimation where we use $c_m^{\max}$ and $c_r^{\max}$ as the estimated cost of retrieving and processing a unit of data in a map and a reduce task and η as the estimated ratio between a job's intermediate data size and input data size, it is highly likely that a job finishes earlier than that estimated by the admission controller. In addition, node failures or speculative re-execution of slow tasks can result in a job finish time later than expected. To reduce false negatives (i.e., rejecting jobs that can meet their deadlines) and deal with unexpected events (such as node failures), a feedback controller is invoked to update all waiting jobs' T^m and T^r vectors if the difference between a job's actual and estimated finish times is larger than a certain threshold Δ. The feedback controller is also triggered if a job misses its deadline due to unexpected events. As a result of the update, the admission controller makes decisions based on more accurate estimates. Algorithms V and VI show the pseudo code of the feedback controller.

To avoid high algorithm overhead, we do not keep track of $J.V^m$ and $J.V^r$, the actual available time of the cluster's map and reduce slots after considering the actual execution of job J and J's predecessors. Tracking these vectors is not an easy task. First, it requires identifying the correct execution slot and updating it after each task's execution. Second, as mentioned in Section C, to well utilize system resources, we develop a reservation-based reduce task dispatcher, which allows out of order execution of jobs' reduce stages and out of order completion of jobs. Thus, a job may finish its execution before some of its predecessors and after some of its successors. Due to these cases, simply taking snapshots of the cluster when a job J's tasks finish will not give the correct $J.V^m$ and $J.V^r$ vectors. In addition, there is a more critical problem: due to out of order job completion, if some of J's predecessors are still executing, the actual values of $J.V^m$ and $J.V^r$ are unknown when job J finishes and when the feedback controller is triggered. Thus, instead of tracking these vectors, we derive U^m and U^r vectors as updated estimates of $J.V^m$ and $J.V^r$. This estimation is carried out only when the feedback controller (Algorithm V) invokes the slot available time update (Algorithm VI). To derive U^m and U^r, like deriving $J.T^m$ and $J.T^r$, we still assume all J's predecessors finish and make the slots available at T_p^m and T_p^r. Then the actual execution of job J's map and reduce tasks are considered following a non-decreasing order of task finish time and it is assumed that the earlier an execution slot becomes available, i.e., the earlier an execution slot starts to run a task, the earlier it finishes the task execution (Algorithm VI lines 7-21). These assumptions may not hold in the actual execution and thus U^m and U^r are only updated estimates of $J.V^m$ and $J.V^r$. However, as long as $U^m \geq J.V^m$ and $U^r \geq J.V^r$, the feedback controller still works correctly and preserves RTMR scheduler's real-time property.

ALGORITHM V. FEEDBACK CONTROLLER

```
FC(J=(A, D, M, R, δ), Priority-Q)
 1: Δ: threshold to trigger the update
 2: $\tilde{e}$ : job J's actual finish time
 3: $J_p$ = getPredecessor(J, Priority-Q)
 4: $T_p^m$ = $J_p.T^m$ ($T_p^m$ = [0,0, …0] if $J_p$ = nil)
 5: $T_p^r$ = $J_p.T^r$ ($T_p^r$ = [0,0, …0] if $J_p$ = nil)
    // invoke Algorithm III to do the calculation
 6: e = Cal$T^r$(J, $T_p^m$, $T_p^r$).e
 7: if | e-$\tilde{e}$ | ≥ Δ or $\tilde{e}$ > (A+D) then
 8:     build $\tilde{E}^m$, the sorted vector containing the actual
        finish time of job J's map tasks
 9:     build $\tilde{E}^r$, the sorted vector containing the actual finish
        time of job J's reduce tasks
        // invoke Algorithm VI to calculate the updated estimates
 10:    $J.T^m$ = SATU(J, $T_p^m$, $T_p^r$, $\tilde{E}^m$, $\tilde{E}^r$).$U^m$
 11:    $J.T^r$ = SATU(J, $T_p^m$, $T_p^r$, $\tilde{E}^m$, $\tilde{E}^r$).$U^r$
 12:    $J_p$ = J
 13:    $J_s$ = getSuccessor($J_p$, Priority-Q)
 14:    while $J_s$ != nil do
            // invoke Algorithms II and III to do the calculation
 15:        $J_s.T^m$ = Cal$T^m$($J_s$, $J_p.T^m$).$T^m$
 16:        $J_s.T^r$ = Cal$T^r$($J_s$, $J_p.T^m$, $J_p.T^r$).$T^r$
 17:        $J_p$ = $J_s$
 18:        $J_s$ = getSuccessor($J_p$, Priority-Q)
 19:    end while
 20: else return
 21: end if
```

ALGORITHM VI. SLOT AVAILABLE TIME UPDATE

```
SATU (J=(A, D, M, R, δ), $T_p^m$, $T_p^r$, $\tilde{E}^m$, $\tilde{E}^r$)
 1: $T_p^m$ : map slot available time in J's predecessor's record
 2: $T_p^r$ : reduce slot available time in J's predecessor's record
 3: $\tilde{E}^m$ : sorted vector containing the actual finish time of job
    J's map tasks
 4: $\tilde{E}^r$ : sorted vector containing the actual finish time of job
    J's reduce tasks
 5: $U^m$ = $T_p^m$
 6: $U^r$ = $T_p^r$
 7: while $\tilde{E}^m$ is not empty do
 8:     remove the item currently located at the beginning of
        vector $\tilde{E}^m$, say it is $\tilde{e}_i^m$
 9:     $u_1^m$ = $\tilde{e}_i^m$ (where $u_1^m$ is the first and smallest item in
        vector $U^m$)
 10:    sort items in $U^m$ to keep $U^m$ a sorted vector
 11: end while
 12: while $\tilde{E}^r$ is not empty do
 13:    remove the item currently located at the beginning of
        vector $\tilde{E}^r$, say it is $\tilde{e}_i^r$
 14:    $u_1^r$ = $\tilde{e}_i^r$ (where $u_1^r$ is the first and smallest item in
        vector $U^r$)
 15:    sort items in $U^r$ to keep $U^r$ a sorted vector
 21: end while
 22: return $U^m$, $U^r$
```

We have proved the correctness of the feedback controller by showing that $U^m \geq J.V^m$ and $U^r \geq J.V^r$. Therefore, after updating job J's vectors T^m and T^r with U^m and U^r in Algorithm V (lines 10-11), the condition $J.T^m \geq J.V^m$ and $J.T^r \geq J.V^r$ (i.e., the estimated slot available time is greater or equal to the actual available time) still holds for job J (please refer to the Appendix for the proof). Since the derivation of $J_s.T^m$ and $J_s.T^r$ are based on $J.T^m$ and $J.T^r$ (see Algorithm V),

$J\ .\ T^m \geq J\ .\ V^m$ and $J\ .\ T^r \geq J\ .\ V^r$ also ensures that $J_s . T^m \geq J_s . V^m$ and $J_s . T^r \geq J_s . V^r$ for all succeeding jobs J_s .

V. EVALUATION

Our implementation of RTMR scheduler and Deadline Constraint scheduler [12] are all based on Hadoop 0.21[1]. These two schedulers are implemented and compared experimentally in terms of real-time property and cluster utilization. To test the effects of feedback control, we run RTMR scheduler twice, with and without the feedback controller enabled. In addition, since the cluster utilization is determined by not only the scheduling algorithm but also the workload volume, we run the default Hadoop FIFO scheduler, which accepts all jobs to execute in the cluster, collecting its resultant cluster utilization to reflect the workload volume. If a real-time scheduler achieves a cluster utilization close to that achieved by the default Hadoop FIFO scheduler, we think that the resource cost of providing the real-time property is not high.

For the RTMR scheduler, the admission controller is implemented in the *JobQueueJobInProgressListener* class which makes the admission control decision and maintains the MapReduce job queue. The dispatcher is in the *RTMRTaskScheduler* class which extends from the *TaskScheduler* class and is in charge of dispatching map and reduces tasks. The feedback controller is also in the *JobQueueJobInProgressListener* class, where we set the threshold Δ to be a typical map task execution time.

Similarly, Deadline Constraint scheduler's admission controller is in *JobQueueJobInProgressListener* class and its dispatcher, called *DCTaskScheduler*, extends from the *TaskScheduler* class.

A heterogeneous Hadoop cluster that contains one master node and 30 worker nodes is used as the testbed. The 30 worker nodes are configured as one rack and they are of two types. 20 of them are 2 dual-core CPU nodes and 10 of them are 2 single-core CPU nodes. Table I gives the detailed hardware information of the cluster. We make the number of map slots in a worker node equal to the number of CPU cores. Because each node has only one Ethernet card, we configure one reduce slot per worker node to avoid bandwidth competition between multiple reduce tasks on a single node. *Loadgen*, a test example in Hadoop source code for evaluating Hadoop schedulers [16][17], is used as the test application.

[1] Kc and Anyanwu [12] implemented Constraint scheduler in Hadoop 0.20.2. We instead choose Hadoop 0.21 because it is the closest version to 0.20.2 but with improved features necessary for small and medium size clusters. Since Hadoop 0.23/2.x is mainly designed for large clusters, it is not adopted for our experiments.

TABLE I. EXPERIMENTAL ENVIRONMENT

Nodes	**Quantity**	**Hardware and Hadoop Configuration**
Master node	1	2 single-core 2.2GHz Opteron-248 CPUs, 8GB RAM, 1Gbps Ethernet
Type I worker nodes	20	2 dual-core 2.2GHz Opteron-275 CPUs, 4GB RAM, 1 Gbps Ethernet, 4 map and 1 reduce slots per node
Type II worker nodes	10	2 single-core 2.2GHz Opteron-64 CPUs, 4GB RAM, 1 Gbps Ethernet, 2 map and 1 reduce slots per node

We first create a submission schedule (workload I) that is similar to the one used by Zaharia et al. [17]. Zaharia et al. [17] generated a submission schedule for 100 jobs by sampling job inter-arrival times and input sizes from the distribution seen at Facebook over a week in October 2009. By sampling job inter-arrival times at random from the Facebook trace, they found that the distribution of inter-arrival times was roughly exponential with a mean of 14 seconds. They also generated job input sizes based on the Facebook workload, by looking at the distribution of the number of map tasks per job at Facebook and creating datasets with the corresponding sizes (i.e., each map task requires a 128 MB input block). To make it possible to compare jobs in the same bin within and across experiments, job sizes were quantized into nine bins, listed in Table II [17]. Our workload I has similar job sizes and job inter-arrival times. In particular, our job size distribution follows the first six bins of the benchmark shown in Table II, which reflect about 89% of the jobs at the Facebook production cluster. Because our testbed is limited in size, we exclude those jobs with more than 300 map tasks. Like the schedule in [17], the distribution of inter-arrival times is exponential with a mean of 14 seconds, making our workload totally 21 minutes long.

The submission schedule used by Zaharia et al. [17], however, does not specify the number of reduce tasks and the deadline for a job. To generate workload I, we create two intervals in each job bin (see Table II), one for reduce task number and one for deadline. Two random numbers from the two intervals are picked as the number of reduce tasks and the deadline for a job. Because the Deadline Constraint scheduler cannot accept a job with more reduce tasks than the cluster's total number of reduce slots, for workload I, we fix the maximum number of reduce tasks per job to be 30, the total number of reduce slots in the cluster.

TABLE II. DISTRIBUTION OF JOB SIZES (in Terms of Number of Map Tasks) at Facebook [17]

Bin	**#Maps**	**%Jobs at**	**#Maps in Benchmark**	**# of jobs in**

		Facebook		Benchmark
1	1	39%	1	38
2	2	16%	2	16
3	3-20	14%	10	14
4	21-60	9%	50	8
5	61-150	6%	100	6
6	151-300	6%	200	6
7	301-500	4%	400	4
8	501-1500	4%	800	4
9	>1501	3%	4800	4

TABLE III. WORKLOAD I'S CONFIGURATION(in Terms of Number of Map, Reduce Tasks and Deadline)

Bin	#Maps	#Reduces	Deadline (second)
1	1	[1,5]	[200,300]
2	2	[1,5]	[200,300]
3	10	[5,10]	[300,400]
4	50	[10,20]	[500,800]
5	100	[20,30]	[1000,1500]
6	200	30	[2000,2500]

Since most jobs in the Facebook workload are small, in particular, some of them having only 1 map task, we create workload II to include more jobs with higher parallelism. That is, in workload II, we let the number of map tasks per job follow normal distribution with an average of 100. Again, because of the moderate size of our cluster, we do not include the three jobs that have more than 300 map tasks. Table IV shows the detailed information of workload II. To test how RTMR scheduler works with large jobs, we also create some jobs with more reduce tasks than the cluster's total number of reduce slots in workload II. However, since we already know that Deadline Constraint scheduler cannot accept such jobs, they are not included in workload II when Deadline Constraint scheduler is tested.

For performance evaluation of the real-time schedulers, the following three metrics, i.e. *job accept ratio, job success ratio*, and *cluster utilization* are used:

$$AcceptR = \frac{\#accepted\_jobs}{\#jobs\_in\_a\_workload}$$

$$SuccessR = \frac{\#successful\_jobs}{\#accepted\_jobs}$$

$$Util = \frac{slot\_time\_used\_by\_successful\_jobs}{available\_slot\_time\_during\_workload\_exe}$$

TABLE IV. WORKLOAD II'S CONFIGURATION (in Terms of Number of Map, Reduce Tasks and Deadline)

Bin	No. Job	#Maps	#Reduces	Deadline (second)
1	9	[1,10]	[1,5]	[200,300]
2	24	[10,50]	[5,10]	[300,500]
3	25	[50,100]	[15,30]	[1000,1500]
4	18	[100,200]	[25,50]	[1500,2500]
5	13	[200,300]	[35,70]	[2500,3500]

The following equation is used to calculate the cluster utilization achieved by default Hadoop FIFO scheduler:

$$Util = \frac{slot\_time\_used\_by\_all\_jobs}{available\_slot\_time\_during\_workload\_exe}$$

Here, *successful_jobs* denotes those jobs that finish before their deadlines and *slot_time_used_by_successful_jobs* refers to the total map and reduce slot time used to execute them. Since Hadoop FIFO scheduler does not consider job deadlines and provides no real-time guarantees, it accepts all jobs and its cluster utilization is calculated using *slot_time_used_by_all_jobs* instead.

available_slot_time_during_workload_exe refers to the total usable time of cluster map and reduce slots during the execution of a workload, i.e., the product of the number of slots and the turnaround execution time of all accepted jobs in a workload.

Tables V and VI show how schedulers perform with workload I and II respectively. As we can see, although compared to RTMR scheduler Deadline Constraint scheduler accepts more jobs, it fails to provide deadline guarantees to all accepted jobs, with job success ratio of 85.7% and 22.5% respectively. Since not all accepted jobs are successful while more jobs are accepted, which prolong the workload's execution in the cluster, Deadline Constraint scheduler leads to much lower cluster utilizations of only 5.7% and 0.7% respectively. In contrast, RTMR scheduler maintains good cluster utilization of 15.5% and 64.6%, in comparison to 21.3% and 69.7% achieved by default Hadoop FIFO scheduler. Deadline Constraint scheduler's very poor performance with workload II experimentally demonstrates its deficiencies in handling real-time MapReduce jobs with high parallelism. From the data, we can also conclude that RTMR scheduler performs better when we enable the feedback controller to keep the admission controller up-to-date, which results in better *job accept ratio* and *cluster utilization*.

VI. CONCLUSION

This paper develops, implements, and experimentally evaluates a novel Real-Time MapReduce (RTMR) scheduler for cluster-based scheduling of real-time MapReduce applications. RTMR scheduler overcomes the deficiencies of an existing algorithm and achieves good cluster utilization and 100% job success ratio, ensuring the real-time property for all admitted MapReduce jobs.

TABLE V. SCHEDULER PERFORMANCE WITH WORKLOAD I

Metrics	Deadline Constraint	RTMR	RTMR w/o Feedback	Hadoop FIFO
Accept Ratio	71.6%	56.8%	46.6%	n/a

Success Ratio	85.7%	100%	100%	n/a
Cluster Utilization	5.7%	15.5%	11.6%	21.3%

TABLE VI. SCHEDULER PERFORMANCE WITH WORKLOAD II

Metrics	**Deadline Constraint**	**RTMR**	**RTMR w/o Feedback**	**Hadoop FIFO**
Accept Ratio	49.4%	24.7%	15.7%	n/a
Success Ratio	22.5%	100%	100%	n/a
Cluster Utilization	0.7%	64.6%	49.8%	69.7%

REFERENCES

[1] Dean, J. and Ghemawat, S. 2008. “MapReduce: Simplified Data Processing on Large Clusters”. Commun. ACM, 51(1):107–113.

[2] Apache Hadoop http://hadoop.apache.org.

[3] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Job scheduling for multi-user mapreduce clusters,” EECS Department, University of California, Berkeley, Tech. Rep., Apr 2009.

[4] American Express. https://www.americanexpress.com/

[5] The Compact Muon Solenoid Experiment. Available: http://cms.web.cern.ch/cms/index.html

[6] The Large Hadron Collider. Available: http://lhc.web.cern.ch/lhc

[7] Attebury, G.; Baranovski, A.; Bloom, K.; Bockelman, B.; Kcira, D.; Letts, J.; Levshina, T.; Lundestedt, C.; Martin, T.; Maier, W.; Haifeng Pi; Rana, A.; Sfiligoi, I.; Sim, A.; Thomas, M.; Wuerthwein, F.; “Hadoop distributed file system for the Grid”. Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE. pp. 1056 – 1061.

[8] Open Science Grid. Available: http://www.opensciencegrid.org

[9] Hadoop Users

http://wiki.apache.org/hadoop/PoweredBy#F

[10] Capacity Scheduler

http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html

[11] Jorda Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for mapreduce environments. In Network Operations and Management Symposium (NOMS), 2010 IEEE, pages 373 –380, 19-23 2010.

[12] K. Kc and K. Anyanwu, “Scheduling hadoop jobs to meet deadlines,” in 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2010, pp. 388 – 392.

[13] Xicheng Dong, Ying Wang, Huaming Liao “Scheduling Mixed Real-time and Non-real-time Applications in MapReduce Environment”. In the proceeding of 17th International Conference on Parallel and Distributed Systems. 2011, pp. 9 – 16.

[14] Xuan Lin, Ying Lu, J. Deogun, and S. Goddard. Real-time divisible load scheduling for cluster computing. In Real Time and Embedded Technology and Applications Symposium, 2007. RTAS ’07. 13th IEEE pages 303 –314, 3-6 2007.

[15] HDFS http://hadoop.apache.org/common/docs/current/hdfs design.html

[16] Chen He, Ying Lu, David Swanson. “Matchmaking : A New MapReduce Scheduling Technique”. In the proceeding of 2011 CloudCom, Athens, Greece, 2011, pp. 40 – 47.

[17] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma and Khaled Elmeleegy, Scott Shenker, and Ion Stoica, “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling”. In the proceedings of the 5th European conference on Computer systems, 2010. pp 265-278.

[18] Zhuo Tang, Junqing Zhou, Kenli Li, and Ruixuan Li "A MapReduce task scheduling algorithm for deadline constraints.", Cluster Computing, Vol. 15, 2012.

[19] Eunji Hwang, and Kyong Hoon Kim. "Minimizing Cost of Virtual Machines for Deadline-Constrained MapReduce Applications in the Cloud." Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on. IEEE, 2012.

[20] Micheal Mattess, Rodrigo N. Calheiros, and Rajkumar Buyya. "Scaling MapReduce Applications across Hybrid Clouds to Meet Soft Deadlines." Technical Report CLOUDS-TR-2012-5, Cloud Computing and Distributed Systems Laboratory, the University of Melbourne, August 15, 2012.

APPENDIX

This Appendix proves the correctness of our real-time MapReduce (RTMR) scheduling algorithm.

First, the correctness of admission control and dispatch algorithms is proved. That is, we prove that all jobs accepted by the admission controller can be successfully dispatched and completed by their deadlines in normal scenarios when there is neither a node failure nor a task re-execution. Several vector operators used in the proof are defined below.

Definition-1: > & ≥

For two sorted vectors V^A *and* V^B, *where*

$$V^A = (v_1^A, v_2^A, v_3^A, ...v_n^A), v_i^A \le v_j^A, 1 \le i < j \le n$$

$$\text{and}\, V^B = (v_1^B, v_2^B, v_3^B, ...v_n^B), v_k^B \le v_l^B, 1 \le k < l \le n$$

$V^A > V^B$ *if and only if* $v_i^A > v_i^B, i = 1,2,...n$;

$V^A \ge V^B$ *if and only if* $v_i^A \ge v_i^B, i = 1,2,...n$.

Definition-2: ⊕

For a sorted vector V^A

$$V^A = (v_1^A, v_2^A, v_3^A, ...v_n^A), v_i^A \le v_j^A, 1 \le i < j \le n$$

and a vector V^B

$$V^B = (v_1^B, v_2^B, v_3^B, ...v_m^B)$$

$V^A \oplus V^B$ generates an n dimensional vector V^C as follows: first, let $V^C = V^A$; second, from V^B , remove the item currently located at the beginning of the vector, say it is v_i^B ; third, change v_1^C to be equal to $v_1^C + v_i^B$ and resort V^C to keep it a sorted vector; forth, repeat the second and third steps until there is no element left in V^B .

Definition-3 maximum of a vector and a value

For a sorted vector V^A

$$V^A = (v_1^A, v_2^A, v_3^A, ...v_n^A), v_i^A \le v_j^A, 1 \le i < j \le n$$

and a value a , $MAX(V^A, a)$ generates an n dimensional vector as follows:

$$MAX(V^A, a) = (\max(v_1^A, a), \max(v_2^A, a), ..., \max(v_n^A, a))$$

It can be easily proved that the following properties hold for the aforementioned operators:

1) If $V^A \ge V^B$ and $V^B \ge V^C$, then $V^A \ge V^C$

2) If $V^A \ge V^B$ and $V^C \ge V^D$,

then $V^A \oplus V^C \ge V^B \oplus V^D$

3) If $V^A \ge V^B$ and $a \ge b$,

then $MAX(V^A, a) \ge MAX(V^B, b)$

The admission controller generates $J.T^m$ and $J.T^r$ vectors, which record the estimated slot available time after the scheduled execution of job J and J's predecessors, while $J.V^m$ and $J.V^r$ respectively represent the actual available time of the cluster's map and reduce slots after considering these jobs' actual execution. To guarantee that an accepted job J_i does not miss its deadline in normal scenarios, we prove $\forall i, J_i.T^m \ge J_i.V^m$ and $J_i.T^r \ge J_i.V^r$ when there is neither a node failure nor a task re-execution.

Proof-1:

Admission control algorithm ensures $\forall i, J_i.T^m \ge J_i.V^m$

For the first job J$_1$ admitted to the cluster, since it does not have a proceeding job, when the admission controller calculates $J_1.T^m$, we have $T_p^m = [0,0,...0]$ (see Algorithm 1), which equals V_0^m , the initial available time of the cluster's map slots. According to Algorithm 2, $J_1.T^m$ is calculated as follows:

$J_1.T^m = MAX(T_p^m, J_1.A) \oplus \max \varepsilon_1^m = MAX(V_0^m, J_1.A) \oplus \max \varepsilon_1^m$ where $\max \varepsilon_1^m$ is a vector composed of M items with equal value of $\tilde{\varepsilon}_1^m = c_m^{\max} * \max(J_1.\delta_i^m, i = 1,2,...M)$. In addition, we have:

$$J_1.V^m = MAX(V_0^m, J_1.A) \oplus \Sigma_1^m$$

where Σ_1^m is the vector composed of the actual execution time of J$_1$'s map tasks. Since $\tilde{\varepsilon}_1^m$ is a pessimistic estimation of a map task's execution time, we have:

$$\max \varepsilon_1^m \ge \Sigma_1^m$$

According to the property of vector operator "⊕", we conclude from the above three equations and inequality that:

$$J_1.T^m \ge J_1.V^m$$

Assuming $J_k.T^m \ge J_k.V^m$, we can show that $J_{k+1}.T^m \ge J_{k+1}.V^m$ following a similar proof procedure. According to mathematical induction, we conclude $J_i.T^m \ge J_i.V^m$ for all accepted job J_i .

Proof-2:

Admission control algorithm ensures $\forall i, J_i.T^r \ge J_i.V^r$

For the first job J$_1$ admitted to the cluster, since it does not have a proceeding job, when the admission controller calculates $J_1.T^r$, we have $T_p^r = [0,0,...0]$ (see Algorithm 1), which equals V_0^r , the initial available time of the cluster's reduce slots. According to Algorithm 3, $J_1.T^r$ is calculated as follows:

$J_1.T^r = MAX(T_p^r, J_1.e^m) \oplus \max \varepsilon_1^r = MAX(V_0^r, J_1.e^m) \oplus \max \varepsilon_1^r$ where $J_1.e^m$ is the estimated finish time of J$_1$'s map stage and $\max \varepsilon_1^r$ is a vector composed of R items with equal value of $\tilde{\varepsilon}_1^r = c_r^{\max} * \max(J_1.\delta_i^r, i = 1,2,...R)$. In addition, we have:

$$J_1.V^r = MAX(V_0^r, J_1.\tilde{e}^m) \oplus \Sigma_1^r$$

where $J_1.\tilde{e}^m$ is the actual finish time of J$_1$'s map stage and Σ_1^r is the vector composed of the actual execution time of J$_1$'s reduce tasks. Since as shown in ***Proof-1*** $J_1.T^m \ge J_1.V^m$, it implies the following relation for the largest items (i.e., $J_1.e^m$ and $J_1.\tilde{e}^m$) of the two vectors:

$$J_1.e^m \ge J_1.\tilde{e}^m$$

And since $\tilde{\varepsilon}_1^r$ is a pessimistic estimation of a reduce task's execution time, we have:

$$\max \varepsilon_1^r \geq \Sigma_1^r$$

According to the properties of "MAX" and "⊕" operators, we conclude from the above four equations and inequalities that:

$$J_1.T^r \geq J_1.V^r$$

Assuming $J_k.T^r \geq J_k.V^r$, we can show that $J_{k+1}.T^r \geq J_{k+1}.V^r$ following a similar proof procedure. According to mathematical induction, we conclude $J_i.T^r \geq J_i.V^r$ for all accepted job J_i.

In the following part of this section, we prove the correctness of the feedback controller by showing that $U^m \geq J.V^m$ and $U^r \geq J.V^r$. Therefore, after updating job J's vectors T^m and T^r with U^m and U^r in Algorithm 5 (lines 10-11), the condition $J.T^m \geq J.V^m$ and $J.T^r \geq J.V^r$ (i.e., the estimated slot available time is greater or equal to the actual available time) still holds for job J.

Proof-3: **Algorithm 6 ensures $U^m \geq J.V^m$**

We first prove by induction that $U^m \geq V^{m,i}$ holds after the ith iteration (where i=1, …, M) of the first while loop (i.e., lines 7-11) of Algorithm 6. Here, $V^{m,i}$ represents how $J.V^m$ looks like after considering the actual execution of the ith map task of job J.

Step 1: $U^m \geq V^{m,i}$ is true after the first iteration of the while loop, i.e. $U^m \geq V^{m,i}$ is true for i=1.

As we have shown in ***Proof-1***, the admission control algorithm ensures $J.T^m \geq J.V^m$, therefore after executing line 5 of Algorithm 6 (i.e., $U^m = T_p^m$) we have $U^m \geq J_p.V^m$ and thus $U^m \geq V^{m,i}$ holds before entering the while loop (i.e., $U^m \geq V^{m,i}$ is true for i=0).

Upon the completion of the first map task of job J at time point $\tilde{e}_1^m$, $V^{m,i} = [v_1^m, v_2^m, \ldots v_l^m]$, the sorted vector representing the actual available time of the cluster's map slots, first gets updated to be $[v_1^m, \ldots, v_{j-1}^m, \tilde{e}_1^m, v_{j+1}^m, \ldots v_l^m]$. Here, it is assumed that the map slot corresponding to the current jth position of vector $V^{m,i}$ has been used to execute the task and thus gets updated to $\tilde{e}_1^m$. Since it takes some time to execute a task, we have the new available time greater than the old available time of the slot, i.e., $\tilde{e}_1^m > v_j^m$. We thus know that $v_1^m \leq v_2^m \ldots \leq v_{j-1}^m \leq v_j^m < \tilde{e}_1^m$ holds, which means that for the first j items of vector $V^{m,i} = [v_1^m, \ldots, v_{j-1}^m, \tilde{e}_1^m, v_{j+1}^m, \ldots v_l^m]$, we have $v_1^m \leq v_2^m \ldots \leq v_{j-1}^m < \tilde{e}_1^m$. Then, we sort the vector and get $V^{m,i} = [\tilde{v}_1^m, \ldots, \tilde{v}_n^m, \tilde{e}_1^m, \tilde{v}_{n+1}^m, \ldots \tilde{v}_{l-1}^m]$, where *n ≥ j-1*. In addition, we know for *1≤ p ≤j-1*, $\tilde{v}_p^m = v_p^m$ and for *j-1 < p ≤ n* and *n+1 ≤ p ≤ l-1*, $\tilde{v}_p^m = v_{p+1}^m$.

After the first iteration of the while loop, $U^m = [u_1^m, u_2^m, \ldots u_l^m]$ changes to be a new sorted vector $U^m = [u_2^m, \ldots, u_k^m, \tilde{e}_1^m, u_{k+1}^m, \ldots, u_l^m]$.

Before entering the while loop, $U^m = [u_1^m, u_2^m, \ldots u_l^m]$, $V^{m,i} = [v_1^m, v_2^m, \ldots v_l^m]$, and $U^m \geq V^{m,i}$ holds. Thus, we have for *1≤ p ≤ l*, $u_p^m \geq v_p^m$. After the aforementioned updates, we have $U^m = [u_2^m, \ldots, u_k^m, \tilde{e}_1^m, u_{k+1}^m, \ldots, u_l^m]$ and $V^{m,i} = [\tilde{v}_1^m, \ldots, \tilde{v}_n^m, \tilde{e}_1^m, \tilde{v}_{n+1}^m, \ldots \tilde{v}_{l-1}^m]$, and

1) ***For the first k-1 items*** of the two vectors**,** i.e., when *1 ≤p ≤ k-1*, $u_{p+1}^m \geq \tilde{v}_p^m$ holds. The reasoning is as follows: $\tilde{v}_p^m$ equals either v_p^m or v_{p+1}^m. When $\tilde{v}_p^m = v_p^m$, because $u_{p+1}^m \geq u_p^m$, $u_p^m \geq v_p^m$, and $v_p^m = \tilde{v}_p^m$, we have $u_{p+1}^m \geq \tilde{v}_p^m$; and when $\tilde{v}_p^m = v_{p+1}^m$, because $u_{p+1}^m \geq v_{p+1}^m$ and $v_{p+1}^m = \tilde{v}_p^m$, we too have $u_{p+1}^m \geq \tilde{v}_p^m$.

2) ***The kth item*** of U^m is always greater or equal to that of $V^{m,i}$. The reasoning is as follows: because when *1 ≤p ≤ k-1*, $u_{p+1}^m \geq \tilde{v}_p^m$ and both $V^{m,i} = [\tilde{v}_1^m, \ldots, \tilde{v}_n^m, \tilde{e}_1^m, \tilde{v}_{n+1}^m, \ldots \tilde{v}_{l-1}^m]$ and $U^m = [u_2^m, \ldots, u_k^m, \tilde{e}_1^m, u_{k+1}^m, \ldots, u_l^m]$ are sorted vectors, $\tilde{e}_1^m$'s position in U^m must be earlier than that in $V^{m,i}$, i.e., *k ≤ n+1*. If *k = n+1*, the kth items of vectors U^m and $V^{m,i}$ all equal to $\tilde{e}_1^m$. If *k < n+1*, the kth items of vectors U^m and $V^{m,i}$ are $\tilde{e}_1^m$ and $\tilde{v}_k^m$. Since $V^{m,i} = [\tilde{v}_1^m, \ldots, \tilde{v}_n^m, \tilde{e}_1^m, \tilde{v}_{n+1}^m, \ldots \tilde{v}_{l-1}^m]$ is a sorted vector, i.e., $\tilde{v}_1^m \leq \ldots \leq \tilde{v}_k^m \leq \ldots \leq \tilde{v}_n^m \leq \tilde{e}_1^m$, we have $\tilde{e}_1^m \geq \tilde{v}_k^m$. That is, the kth item of U^m is always greater or equal to that of $V^{m,i}$.

3) ***For all items from the (k+1)th to the nth positions***, i.e., when *k+1 ≤ p ≤ n*, we have $u_{p+1}^{m} \geq \tilde{v}_{p}^{m}$ since $u_{p+1}^{m} \geq \tilde{e}_{1}^{m}$ and $\tilde{e}_{1}^{m} \geq \tilde{v}_{p}^{m}$.

4) ***The (n+1)th item*** of U^{m} is always greater or equal to that of $V^{m,i}$. The reasoning is as follows: we know that *k ≤ n+1*. If *k = n+1*, the kth items of vectors U^{m} and $V^{m,i}$ are equal since they both equal to $\tilde{e}_{1}^{m}$. If *k < n+1*, the (n+1)th items of U^{m} and $V^{m,i}$, are u_{n+1}^{m} and $\tilde{e}_{1}^{m}$ respectively. Since $U^{m} = [u_{2}^{m}, \ldots, u_{k}^{m}, \tilde{e}_{1}^{m}, u_{k+1}^{m}, \ldots, u_{l}^{m}]$ is a sorted vector, i.e., $u_{2}^{m} \leq \ldots \leq u_{k}^{m} \leq \tilde{e}_{1}^{m} \leq \ldots \leq u_{n+1}^{m} \leq \ldots$

$\leq u_{l}^{m}$, we have $u_{n+1}^{m} \geq \tilde{e}_{1}^{m}$, the (n+1)th item of U^{m} is greater or equal to that of $V^{m,i}$.

5) ***For the last l-(n+1) items*** of the two vectors, i.e., when *n+1 ≤ p ≤ l-1*, we have $u_{p+1}^{m} \geq \tilde{v}_{p}^{m}$ since $u_{p+1}^{m} \geq v_{p+1}^{m}$ and $\tilde{v}_{p}^{m} = v_{p+1}^{m}$.

In summary, $U^{m} \geq V^{m,i}$ holds after the first iteration of the while loop, i.e., $U^{m} \geq V^{m,i}$ is true for i=1.

Step 2: Assume $U^{m} \geq V^{m,i}$ holds after the qth iteration of the while loop, i.e., $U^{m} \geq V^{m,i}$ is true for i=q.

Step 3: Following a procedure similar to Step 1, we can prove that $U^{m} \geq V^{m,i}$ also holds after the (q+1)th iteration of the while loop, i.e., $U^{m} \geq V^{m,i}$ is true for i=q+1.

According to mathematical induction, we conclude $U^{m} \geq V^{m,i}$ holds after the ith iteration, for i=1, …, M, of the first while loop (i.e., lines 7-11) of Algorithm 6.

Since the values of both vectors (i.e., U^{m} and $V^{m,i}$) do not change after the first while loop, we have proved that Algorithm 6 ensures $U^{m} \geq V^{m,i}$ for i=M, that is, $U^{m} \geq J.V^{m}$.

Proof-4: **Algorithm 6 ensures** $U^{r} \geq J.V^{r}$

Similar to the procedure of ***Proof-3***, we can prove Algorithm 6 ensures $U^{r} \geq J.V^{r}$.

According to ***Proof-3*** and ***Proof-4***, we conclude that after updating $J.T^{m}$ and $J.T^{r}$ with U^{m} and U^{r} by invoking Algorithm 6 in Algorithm 5, the condition $J.T^{m} \geq J.V^{m}$ and $J.T^{r} \geq J.V^{r}$ (i.e., the estimated slot available time is greater or equal to the actual available time) still holds for job J.