OMO: Optimize MapReduce Overlap with a Good Start (Reduce) and a Good Finish (Map)

Jiayin Wang* jane@cs.umb.edu
Yi Yao† yyao@ece.neu.edu
Ying Mao* yingmao@cs.umb.edu
Bo Sheng* shengbo@cs.umb.edu
Ningfang Mi† ningfang@ece.neu.edu

*Department of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125
†Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115

***Abstract*—MapReduce has become a popular data processing framework in the past few years. Scheduling algorithm is crucial to the performance of a MapReduce cluster, especially when the cluster is concurrently executing a batch of MapReduce jobs. However, the scheduling problem in MapReduce is different from the traditional job scheduling problem as the reduce phase usually starts before the map phase is finished to "shuffle" the intermediate data. This paper develops a new strategy, named OMO, which particularly aims to optimize the overlap between the map and reduce phases. Our solution includes two new techniques, lazy start of reduce tasks and batch finish of map tasks, which catch the characteristics of the overlap in a MapReduce process and achieve a good alignment of the two phases. We have implemented OMO on Hadoop system and evaluated the performance with extensive experiments. The results show that OMO's performance is superior in terms of total completion length (i.e., makespan) of a batch of jobs.**

I. Introduction

MapReduce [1] has become a popular data processing framework in the past few years. Its open source implementation Hadoop [2] and the corresponding eco-system have attracted a lot of attentions and been widely adopted in many fields. With the rise of cloud computing and the high demand of big data processing, we envision that more and more users will launch a MapReduce cluster to process a large volume of data in various applications. A typical MapReduce job includes many identical map tasks and much fewer reduce tasks. Map tasks process the raw data in parallel and generate intermediate data in a form of $< key, value >$. The reduce tasks compute the intermediate data and produce the final results.

This paper aims to develop an efficient scheduling scheme in a MapReduce cluster to improve the resource utilization and reduce the makespan (i.e., the total completion length) of a given set of jobs. Given a limited set of resources in a MapReduce cluster, scheduling algorithm is crucial to the performance, especially when concurrently executing a batch of MapReduce jobs. Without an appropriate management, the available resources may not be efficiently utilized, which leads to a prolonged finish time of the jobs. The scheduling in MapReduce, however, is quite different from the traditional job scheduling in the previous work. MapReduce is composed of 'map' phase and 'reduce' phase, where the intermediate output of 'map' serves as the input of 'reduce'. A typical MapReduce job consists of multiple map tasks and reduce tasks. Thus, the scheduling algorithm in MapReduce needs to handle both job-level and task-level resource management. In addition, a complicated dependency exists between map tasks and reduce tasks of the same job. First, reduce tasks need the output of map tasks, thus cannot be finished before the map phase is done. However, reduce tasks can start earlier before the completion of the map phase (for transferring/shuffling the intermediate data). These factors make the scheduling design extremely challenging yet the existing products have not thoroughly addressed these issues.

This paper develops a new strategy, named OMO, which particularly aims to optimize the overlap between map and reduce phases. We observe that this overlapping period plays an important role in the MapReduce process especially when the map phase generates a large volume data to be shuffled. A good alignment of the map and reduce phases can reduce the job execution time. Compared to the prior work, our solution considers more dynamic factors at the runtime and allocates the resources based on the prediction of the future task execution and resource availability. Specifically, our solution OMO includes two new techniques, lazy start of reduce tasks and batch finish of map tasks. The first technique attempts to find the best timing to start reduce tasks so that there is sufficient time for reduce tasks to shuffle the intermediate data while the resources are allocated to serve map tasks as much as possible. We introduce a novel predication model to estimate the resource availability in the future which further helps make scheduling decision. The second technique is to increase the execution priority of the tailing map tasks in order to finish them in a wave. Different from the prior work that prefers wave-like execution throughout the map phase, we only focus on the last batch of the map tasks. Both techniques catch the characteristics of the overlap in a MapReduce process and achieve a good alignment of the map and reduce phases.

The remainder of this paper is organized as follows. Section II presents the related work of this paper. In Section III, we formalize the problem considered in this paper. Sections IV describes the details of the solution. Section V provides the experimental evaluation of OMO. Finally, we summarize our work in Section VI.

II. Related Work

In Hadoop system, job scheduling is a significant direction. The default FIFO scheduler cannot work fairly in a shared cluster with multiple users and a variety of jobs. FAIR SCHEDULER [3] and Capacity Scheduler [4] are widely used to ensure each job to get a proper share of the available resources.

To improve the performance, Quincy [5] and Delay Scheduling [6] optimize data locality in the case of FAIR SCHEDULER. Coupling Scheduler in [7], [8] aims to mitigate the starvation of reduce slots in FAIR SCHEDULER and analyze the performance by modeling the fundamental scheduling characteristics for MapReduce. Another category of schedulers consider user-level goals while improving the performance. iShuffle [9] separates shuffling from the reduce phase and provides a platform service to manage and schedule data output from map phase. However, reduce slots will be occupied for shuffling from the beginning of the map phase and such resources are not efficiently used since a part of time is used to wait for the ends of map tasks. Recent work TuMM [10], [11] and FRESH [12] have developed dynamic slot configurations in Hadoop based on FIFO and FAIR SCHEDULER. A free task slot will be assigned as map slot or reduce slot according to the workload of the map and reduce phase for all jobs running concurrently in the cluster. Some other work focuses on heterogeneous environments. LATE scheduler [13] was proposed to stop unnecessary speculative executions in order to improve the performance in a heterogeneous Hadoop cluster. LsPS [14] uses the present heterogeneous job size patterns to tune the scheduling schemes.Some other work [15], [16] focused on the skew problem in MapReduce.

The Hadoop community recently released Next Generation MapReduce (YARN) [17]. In YARN, instead of fixed-size slots, each task specifies a resource request in the form of <*memory size, number of CPU cores*> and each slave node offers resource containers to process such requests. Some recent work [18], [19] proposed to use multiple scheduler to solve the scalability issue. Mesos [20] introduced a distributed two-level scheduling mechanism to share clusters and data efficiently between different platforms such as MapReduce, Dryad [21] and others. Our work can be integrated into these platforms as a single low-level scheduler.

III. Problem Formulation

In this paper, we consider a Hadoop cluster as the MapReduce service platform. Currently, there are two branches of Hadoop frameworks available, Hadoop [2] and Hadoop YARN [17]. Our solution and implementation are based on the first generation of Hadoop [2]. But the techniques we present can be easily extend to Hadoop YARN [17]. We will also compare the performance with Hadoop YARN in our evaluation (Section V).

In our problem setting, we consider that a Hadoop cluster consists of a master node and multiple slave nodes. Each node is configured with multiple *slots* which indicate its capacity of serving tasks. A slot can be set as a map slot or reduce slot to serve one map task or reduce task, respectively. We assume there are totally S slots and the cluster has received a batch of n jobs for processing. J represents the set of jobs, $J=\{J_1,J_2,...,J_n\}$. Each job J_i is configured with m_i map tasks and r_i reduce tasks. In a traditional Hadoop system, the cluster administrator has to specify the numbers of map slots and reduce slots in the cluster. A map/reduce slot is dedicated to serve map/reduce tasks throughout the lifetime of the cluster. In this work, however, we adopt a dynamic slot configuration that we have developed in our prior work [10], [12], where a slot can be set as a map slot or reduce slot during the job execution based on the scheduler's decision. Therefore, there is no need to configure the number of map slots and reduce slots before launching the cluster. The system will dynamically allocate slots to serve map and reduce tasks on-the-fly. We omit the details of its implementation in this paper because our focus is a different scheduling strategy based on the dynamic slot configuration. Essentially, our objective is to develop a scheduling algorithm that assigns tasks to available slots in order to minimize the makespan of the given set of MapReduce jobs. Table I lists the notations we use in the rest of this paper.

TABLE I: Notations

$n/K/S$	# of jobs / # of active jobs / # of slots in the cluster
$J_i/m_i/r_i$	i-th job / number of its map tasks / number of its reduce tasks
F_o/A_o	observed slot release frequency / observed # of *available slots*
F_e/A_e	estimated slot release frequency / estimated # of *available slots*
T_m/T_s	execution time of a map task / execution time of the shuffling phase
T_w/R_t	length of a historical window / # of slots released in the t-window
f_t	slot release frequency in the t-th window, $f_t = R_t/T_w$
a_t	# of available slots in t-th window
m'_i	# of pending map tasks of job J_i
α	gap from the end of map phase to the end of shuffling phase
d/B	size of data generated by one map task / network bandwidth

IV. Our Solution : OMO

In this section, we present our solution OMO which aims to reduce the execution time of MapReduce jobs. We develop two new techniques in our solution, *lazy start of reduce tasks* and *batch finish of map tasks*. In the rest of this section, we first describe a monitor module that serves as a building block for both techniques. And then, we introduce these two techniques individually and present a complete algorithm that integrates both of them. The entire solution is mainly developed as a new Hadoop scheduler. The implementation details will be introduced in Section V.

A. Slot release frequency

Both of our new techniques rely on an important parameter which is the estimated frequency of slot release in the system. For a Hadoop scheduler making decisions of resource allocation, this parameter represents the system resource availability in the future. We find that it is a critical factor for the system performance, but neglected by all the prior work. While the details will be discussed in the following subsection, we first present the basic method to estimate the slot release frequency.

We define two parameters F_o and F_e to represent the *observed* slot release frequency and *estimated* slot release frequency respectively, i.e., F_o or F_e slots released per time unit. F_o is a measurement value obtained by monitoring the job execution and F_e is the estimation of the future release frequency that will be used by the scheduler. In addition, we introduce *available slots* to describe the slots that could be possibly released in the near future. Available slots include all the slots serving map tasks and the slots serving a job's reduce tasks if the job's map phase has be finished. In other words, the *available slots* exclude the slots serving the reduce tasks of a job with unfinished map tasks in which case the release time of the slots is undetermined. In our solution, we suppose that for a particular circumstance, the slot release frequency is proportional to the number of *available slots*.

Specifically, we monitor a historic window to measure the number of released slots and the number of available slots in the window indicated by R_t and a_t (for the t-th window), respectively. Assume that the window size is T_w seconds. The slot release frequency in this window will be $f_t = \frac{R_t}{T_w}$ and the ratio between slot release frequency and the number of available slots is $\frac{f_t}{a_t} = \frac{R_t}{a_t \cdot T_w}$. In our design intuition, this ratio is supposed to be consistent over a certain period. For each window t, we thus record the (f_t, a_t) pairs and derive the average value of the slot release frequency F_o and the average number of available slots denoted by A_o. We use the common method of exponential moving average (EMA) to catch the dynamics during the execution,

$$\begin{aligned} F_o(t) &= \alpha \cdot f(t) + (1-\alpha) \cdot F_o(t-1), \\ A_o(t) &= \alpha \cdot a(t) + (1-\alpha) \cdot A_o(t-1), \end{aligned}$$

where $F_o(t)$ or $A_o(t)$ is the value of F_o or A_o after the t-th window and $F_o(t-1)$ or $A_o(t-1)$ indicates the old value of F_o or A_o after the $(t-1)$-th window.

When estimating F_e for a future time, we first need to determine the number of available slots (A_e) at that point. Then, based on the assumption that the slot releasing frequency is proportional to the number of available slots, we can calculate F_e as $F_e = \frac{F_o \cdot A_e}{A_o}$. This component of estimating the slot release frequency will be used by both of our new techniques which will be presented later in this section. We will introduce more details, such as how to obtain the value of A_e, in the algorithm descriptions.

B. Lazy start of reduce tasks

The goal of our first technique is to optimize the start time of the reduce phase of MapReduce jobs. We first show how a traditional Hadoop system controls the overlapping period and give the motivation of our design. Then, we will introduce the intuitions of our solution and the details of the algorithm.

1) Motivation: One important feature of MapReduce jobs is the overlap between the map and reduce phase. The reduce phase usually starts before the map phase is finished, i.e., some reduce tasks may be concurrently running with map tasks of the same job. The benefit of this design is to allow the reduce tasks to *shuffle* (i.e., prepare) the intermediate data (partially) created by map tasks before the entire map phase is done to save the execution time of the reduce tasks. In Hadoop, a system parameter *slowstart* can be configured to indicate when to start the reduce tasks. Specifically, slowstart is a fractional value representing the threshold for the map phase's progress exceeding which reduce tasks will be allowed to execute.

Apparently, setting slowstart to 1 yields the worst performance because all data shuffling happens after the map phase is finished. However, if we start reduce tasks too early, it will not only affect the reduce progress of other jobs, but also the execution of map tasks because those occupied slots could otherwise serve map tasks.

In practice, it is extremely hard for a user to specify the value of slowstart before launching the cluster. And the pre-configured value will not be the optimal for various job workloads. In this paper, we develop a new technique, *lazy start of reduce tasks*, to improve the performance. The basic idea is to postpone the start of reduce phase as much as possible until data shuffling will incur additional delay in the process. Ideally, a perfect alignment of the map and reduce phases is that the last reduce task finishes the data shuffling right after the last map task is completed. However, simply using the slowstart threshold is difficult to achieve the best performance because it depends on not only the progress of the map phase but also other factors such as the map task execution time and shuffling time. In the rest of this subsection, we present our solution that determines the start time of reduce tasks during the execution of the job. We first introduce an algorithm for single job execution to illustrate our design intuition, and then extend it for multiple job execution.

2) Single Job: The original design of the slowstart parameter in Hadoop indicates that the progress of the map phase is certainly important for deciding when to start the first reduce task. The best start time of the reduce phase, however, also depends on the following factors. (1) Shuffling time: This is determined by the size of intermediate data generated by map tasks and the network bandwidth. Intuitively, a job generating more intermediate data after the map phase needs a longer shuffling time in its reduce phase. Thus, we should start the reduce tasks earlier. The intermediate data size is generally proportional to the progress of the map phase. Thus, by monitoring the finished map tasks, we can obtain a good estimation of the final data size. (2) Map task executing time: The benefit of starting reduce tasks before the end of the map phase is to overlap the shuffling in the reduce phase with the execution of the rest of map tasks (the last a few waves of map tasks). Therefore, given a certain shuffling time, if each map task of a job requires longer time to finish, then we prefer to start the reduce phase later. (3) Frequency of slot release: How frequently a slot is released and becomes available in the cluster is also an important factor. For both map and reduce phases, the tailing tasks have the critical impact on the overlapping period. Once we specify a target start time for the last reduce task, we can use the slot release frequency to derive when we should start the reduce phase.

Our solution monitors the above three parameters to decide the start time of the reduce phase. Before introducing the detailed algorithm, we first present and prove a design principal.

Principle 1: Once the reduce phase of a job is started, all reduce tasks of the job should be consecutively executed in order to minimize the job execution time.

Proof: We prove this principle by contradiction. Assume that the best arrangement does not follow this principle, i.e., some map tasks are launched after the first reduce task is started, and before the last few reduce tasks are started. We identify the last such map tasks, e.g., task B in Fig. 1, and the first reduce task, e.g., task A in Fig. 1. Then, we form another arrangement by switching these two tasks and show that the performance is no worse than the original arrangement. After the switch, the finish time of the map phase becomes earlier because task A occupies a slot at a later time point and that slot could serve map tasks before task A is started. Meanwhile, the shuffling performance keeps the same, i.e., the gap from the end of the map phase to the end of the shuffling has no change because the bottleneck of the shuffling is the last reduce task, i.e., task C in Fig. 1. Therefore, if we consider the end of the shuffling phase as the performance indicator, the new arrangement after the switch is no worse than the original solution. We can keep applying the same switch on new arrangements and get a solution where reduce tasks are consecutively executed with no interruption of map tasks. ■

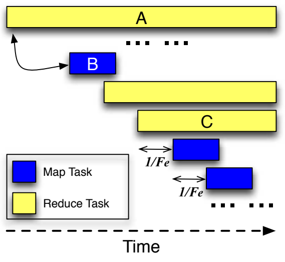

Fig. 1: Illustration of the proof.

Based on the above principal, we analyze the gap from the end of the map phase to the end of the shuffling phase and derive the best start time of the reduce phase to minimize this gap. Assume the running job has m map and r reduce tasks. When a slot becomes available, our scheduler needs to assign it to a new task. When the reduce phase has not started, there are just two options, to serve a map task or to serve a reduce task which starts the reduce phase.

Let α be the time gap from the end of the map phase to the end of the shuffling stage (see Fig. 2) and assume that variable x represents the number of pending map tasks. We first derive α as a function x and then decide the time to start the reduce phase. Additionally, we use T_m to indicate the average execution time of a map task, and T_s to represent the estimated shuffling time of the last reduce task.

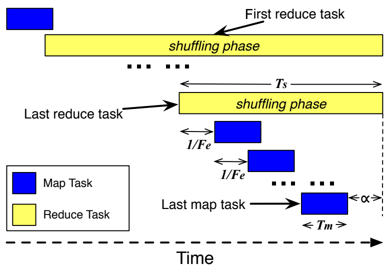

Fig. 2: Lazy Start of Reduce Tasks: illustrating the alignment of map phase and shuffling phase.

Since all reduce tasks are executed consecutively and reduce task slots will not be released until the end of a job, after the last reduce task is assigned, the number of available slots becomes $S - r$. Therefore, the estimated frequency of slot release is decreased to $F_e = \frac{F_o \cdot (S-r)}{S}$. Assuming the slots to be released at a constant rate with an interval of $\frac{1}{F_e}$ between any two consecutive releases, the execution time for the remaining map tasks can be expressed as $\frac{x}{F_e} + T_m$ (see Fig. 2). Therefore, we express α as a function of x:

$$\alpha = T_s - (\frac{x}{F_e} + T_m) = T_s - (\frac{x \cdot S}{F_o \cdot (S-r)} + T_m), \qquad (1)$$

where F_o is a measured value as described in Section IV-B and T_m records the average execution of a map task. Both F_o and T_m are updated once a task is finished. To estimate T_s, we measure the average size of the intermediate data generated by a map task (indicated by d) and the network bandwidth in the cluster (indicated by B). T_s can be expressed as: $T_s = \frac{d \cdot m}{B \cdot r}$

During the execution of a job, our scheduler forms α as the function of x and then calculates the values with different x whenever a slot is released. When the actual number of the pending map tasks m' satisfies the following equation, the reduce phase will be started, i.e., the first reduce task will be assigned to the current available slot:

$$m' = \arg \min_{x \in [m', m]} \alpha.$$

3) Multiple Jobs: Now, we extend our design for serving multiple MapReduce jobs. Our scheduler is built upon FAIR SCHEDULER which evenly distributes slots to all the active jobs. Specifically, if there are S slots in the cluster and K jobs running concurrently, each job can occupy $\frac{S}{K}$ slots and the effective slot release frequency for each job is $\frac{F_o}{K}$.

Following Eq. (1), we calculate the gap α for each job J_i,

$$\alpha = T_s(i) - (\frac{x_i}{F_e} + T_m(i)),$$

where x_i is the variable representing the number of the pending map tasks of J_i and parameters $T_s(i)$ and $T_m(i)$ are also specific to J_i. The estimated slot release frequency F_e can be estimated as $\frac{F_o \cdot A_e}{A_o \cdot K}$, where F_o and A_o are common parameters for all the jobs. With multiple jobs running, A_o may not be the same as S as in the case of single job execution. When calculating α for J_i, we will use the measured value of A_o. However, the following equation still holds $A_e = A_o - r_i$, where r_i is the number of the reduce tasks in J_i. Therefore,

$$\alpha = T_s(i) - (\frac{x_i \cdot A_o \cdot K}{F_o \cdot (A_o - r_i)} + T_m(i)). \qquad (2)$$

Finally, the reduce phase should be started when the number of pending map tasks m'_i satisfies the following equation:

$$m'_i = \arg \min_{x \in [m'_i, m_i]} \alpha.$$

It is possible that multiple jobs satisfy the above equation, in which case our scheduler will allocate the slot to the job that has occupied the fewest slots among all the candidates.

The details of our algorithm are shown in Algorithm 1. Function **LazyStartReduce()** is supposed to return the index of the job that should start its reduce phase. If there is no candidate, the function will return "-1". The variable res

records the set of candidate job indexes. Specifically, lines 1–6 set the number of active jobs (K). Lines 5–14 enumerate all the running jobs that have not started their reduce phases, and use Eq. (2) to determine if they are candidate jobs to start the reduce phase. Eventually, when there are multiple candidates in res, the algorithm return the index of the job with the minimum occupied slots (lines 15–19).

Algorithm 1 Function LazyStartReduce()

```
1:  $K = 0, res = \{\}$
2:  for $i = 1$ to $n$ do
3:      if $J_i$ is running then
4:          $K \leftarrow K + 1$
5:  for $i = 1$ to $n$ do
6:      if $J_i$ is running and has not started reduce phase then
7:          $\alpha_{OPT} = T_s(i) - (\frac{m'_i \cdot A_o \cdot K}{F_o \cdot (A_o - r_i)} + T_m(i))$
8:          selected = true
9:          for $x = m'_i + 1$ to $m_i$ do
10:             $\alpha = T_s(i) - (\frac{x_i \cdot A_o \cdot K}{F_o \cdot (A_o - r_i)} + T_m(i))$
11:             if $\alpha < \alpha_{OPT}$ then
12:                 selected=false; break;
13:         if selected == true then
14:             $res = res + \{i\}$
15: if $res$ is empty then
16:     return $-1$
17: else
18:     sort all the job indexes in $res$ in the ascending order of
        the number of occupied slots
19:     return the first index in the sorted list
```

C. Batch finish of map tasks

Our second technique aims to improve the performance of the map phase by arranging the tailing map tasks to be finished in a batch. In this subsection, we first show how the alignment of map tasks affects the execution time of a MapReduce job, and then present our algorithm to improve the performance.

1) Motivation: In the design of a Hadoop system, the map tasks are expected to finish in waves to achieve the good performance. The misalignment of map tasks, especially the tailing map tasks may significantly degrade the overall job execution time. With a misalignment, the last few pending map tasks will incur an additional round of execution in the map phase, and the reduce tasks that have been started have to wait for the finish of these map tasks causing poor utilization of their occupied slots.

In practice, however, map tasks are barely aligned as waves because the number of map tasks may not be a multiple of the number of the allocated slots. In a traditional Hadoop system, the number of map slots in the cluster is a system parameter, and unknown to the user who submits the job. In our solution with dynamic slot configuration, the same problem remains and with no reserved slots for map or reduce tasks, the number of slots assigned to map tasks is even more uncertain. In addition, when multiple jobs are concurrently running, the misalignment of map tasks is more serious because of the heterogeneous execution times of map and reduce tasks and various scheduling policies.

Therefore, in our solution, we aim to arrange the tailing map tasks in a batch to address this issue. Our intuition is to let the Hadoop scheduler increase the priority of the tailing map tasks when assigning tasks to available slots, which may violate its original policy. The decision depends on the number of pending map tasks and the estimation of the slot release frequency in the future. Basically, given the number of the pending map tasks of a job, if the scheduler finds that the cluster will release a sufficient number of slots in a short time window, it will reserve those future slots to serve the pending map tasks. The benefit is that the target job's map phase can be finished more quickly and the slots occupied by its reduce tasks will become available sooner. The downside is a possible delay incurred to other active jobs because those reserved future slots could otherwise serve them.

2) Algorithm Design: First of all, the candidate jobs for batch finish of map tasks must have started their reduce phase. Otherwise, if we apply this technique to the jobs that have not started their reduce phases, then the result is equivalent to launching their reduce phases after the map phases without any overlap. Second, for each candidate job, our scheduler analyzes the benefit and penalty of finishing the pending map tasks in a batch and then chooses the job which yields the most reward to apply this technique.

Specifically, we examine each job that has started its reduce phase and determine if the batch finish of its map tasks is appropriate. We first analyze the performance under regular FAIR SCHEDULER and then compare to the case if we finish all the pending map tasks in a batch. For each job J_i, recall that m'_i be the number of its pending map tasks and r_i be the number of reduce tasks. Given the slot release frequency F_o, a slot will be allocated to job J_i every $\frac{K}{F_o}$, where K is the number of active jobs in the cluster. Under FAIR SCHEDULER, J_i will finish its map phase in t_{fair} time units,

$$t_{fair} = \frac{K \cdot m'_i}{F_o} + T_m(i). \tag{3}$$

Meanwhile, other jobs get $\frac{F_o \cdot (K-1)}{K}$ slots per time unit, thus the total number of slots that other jobs obtain is

$$s = \frac{F_o \cdot (K-1)}{K} \cdot t_{fair} = (K-1) \cdot m'_i + \frac{F_o \cdot (K-1) \cdot T_m(i)}{K}.$$

Now if we decide to increase the priority of J_i's pending map tasks and finish them in batch, then the map phase will be finished in $m'_i \cdot \frac{1}{F_o} + T_m$ time unites. After that, J_i's reduce slots become available and the slot release frequency will be increased to $F_e = \frac{F_o \cdot (A_o + r_i)}{A_o}$. To contribute s slots to other jobs, the time required is

$$t_{batch} = \frac{s}{F_e} = \frac{s \cdot A_o}{F_o \cdot (A_o + r_i)}. \tag{4}$$

If $t_{batch} < t_{fair}$, then the batch finish of map tasks becomes superior because it achieves the same scenario, i.e., J_i's map phase is finished and all the other jobs get s slots, in a shorter time period. The details are illustrated in Algorithm 2. Function **BatchFinishMap** returns the index of the job that should apply the batch finish to its pending map tasks. Variable c represents the index of the candidate job. If there is no such candidate, the function will return "-1". The algorithm mainly includes a loop (lines 5–9) that enumerates every active job and calculates t_{fair} and t_{batch} to determine if it is worthwhile to apply the technique. Variable max is defined to temporarily record the current maximum difference between t_{fair} and t_{batch}. Eventually, the index of the job with the maximum benefit is returned.

Algorithm 2 Function BatchFinishMap ()

```
1: $K = 0, max = 0, c = -1$
2: for $i = 1$ to $n$ do
3:     if $J_i$ is running then
4:         $K \leftarrow K + 1$
5: for $i = 1$ to $n$ do
6:     if $J_i$ is running and has started reduce phase then
7:         Calculate $t_{fair}$ and $t_{batch}$ as in Eq. (3) and Eq. (4)
8:         if $t_{batch} < t_{fair}$ and $t_{fair} - t_{batch} > max$ then
9:             $max = t_{fair} - t_{batch}$, $c = i$
10: return $c$
```

D. Combination of the two techniques

Finally, our scheduler integrates our two techniques introduced above into the baseline fair scheduler for the execution of all the jobs. The challenge in the design is that there could be conflict between these two techniques. For example, when a slot is released, the first technique may decide to use this slot to start a job's reduce phase, i.e., assigning a reduce task to it, while the second technique may prefer to reserve the slot as well as the following consecutive slots to finish another job's pending tasks in a batch. In our solution, we adopt a simple strategy to solve the issue: when there is a conflict, we give the technique of lazy start of reduce tasks a higher priority. The intuition is that when a new job starts its reduce phase, the decision of batch finish of map tasks could be affected because there is a new candidate for applying the technique.

V. Performance Evaluation

In this section, we evaluate the performance of OMO and compare it with other alternative schemes.

A. Testbed setup and workloads

1) Implementation: We implemented our new scheduler OMO on Hadoop version 0.20.2 by adding a set of new components to support our solution. First, we create four new modules into *JobTracker*: the Task Monitor (**TM**), the Cluster Monitor (**CM**), the Execution Predictor (**EP**) and the Slot Assigner (**SA**). **TM** is responsible for recording the size of the intermediate data created by each map task, the execution progress of each task, the execution time of each completed task and the numbers of the finished and pending map/reduce tasks of each job. According to the statistics from **TM** and the number of concurrent jobs in the cluster from *JobInProgress*, **CM** collects the number of released slots in the whole cluster in real-time and updates the slot release frequency dynamically. It is also responsible for collecting the total intermediate data output by the map phase of each job. Based on above statistics, **EP** is responsible to predict the overall slot frequency of the cluster, the best time point to apply the algorithm of the batch finish of map tasks, the remaining execution time of the map phase and the execution time of the shuffling of each job. Furthermore, the role of **SA** is to assign a map or reduce task to every released slot with the information received from **EP**.

2) Hadoop cluster: All the experiments are conducted on NSF CloudLab platform at the University of Utah [22]. Each server has 8 ARMv8 cores at 2.4GHz, 64 GB memory and 120 GB storage. We create two Hadoop clusters with 20 and 40 slave nodes. Each slave node is configured with 4 slots. OMO and other schedulers for Hadoop are compared on the 20 slave nodes platform. And we use other cluster to evaluate the scalability of OMO. Additionally, we also launch another YARN cluster (v2.6.0) with 20 slave nodes (node managers) for performance comparison. Instead of specifying the number of slots, each node declares 8 CPU cores and 40 GB memory as the resource capacity.

3) Workloads: Our workloads for evaluation consider general Hadoop benchmarks with large datasets as the input. In particular, we use six datasets in our experiments including 10GB/20GB wiki category links data, 10GB/20GB Netflix movie rating data, and 10GB/20GB synthetic data. The wiki data includes wiki page categories information, the movie rating data is the user rating information and the synthetic data is generated by the tool TeraGen in Hadoop. We choose the following six Hadoop benchmarks from Purdue MapReduce Benchmarks Suite [23] to evaluate the performance: (1) $Terasort$: Sort (key,value) tuples on the key with the synthetic data as input. (2) $Sequence\ Count$: Count all unique sets of three consecutive words per document with a list of Wikipedia documents as input. (3) $Word\ Count$: Count the occurrences of each word with a list of Wikipedia documents as input. (4) $Inverted\ Index$: Generate word to document indexing with a list of Wikipedia documents as input. (5) $Classification$: Classify the movies based on their ratings with the Netflix movie rating data as input. (6) $Histogram\ Movies$: Generate a histogram of the number of movies in each user rating with the Netflix movie rating data as input.

Table II shows the details of all six benchmarks in our tests, including the benchmark's name, input data type/size, intermediate data size, and the number of map/reduce tasks.

B. Evaluation

In this subsection, we present the performance of OMO and compare it to other solutions. We mainly compare OMO to the following alternative schedulers in the prior work: (1) Fair scheduler: We use the Hadoop's default slot configuration, i.e, each slave has 2 map slots and 2 reduce slots. The slowstart

TABLE II: Benchmark Characteristics

Benchmark	Input Data	Input Size	Shuffle Size	map, reduce #
Terasort	Synthetic	20 GB 10 GB	20 GB 10 GB	80, 2 40, 1
SeqCount	Wikipedia	20 GB 10 GB	17.5 GB 8.8 GB	80, 2 40, 1
WordCount	Wikipedia	20 GB 10 GB	3.9 GB 2 GB	80, 2 40, 1
InvertedIndex	Wikipedia	20 GB 10 GB	3.45 GB 1.7 GB	80, 2 40, 1
HistMovies	Netflix	20 GB 10 GB	22 KB 11 KB	80, 2 40, 1
Classification	Netflix	20 GB 10 GB	6 MB 3 MB	80, 2 40, 1

is set from the default value 0.05 to 1, represented as *Fair-0.05*, *Fair-0.2*, *Fair-0.4*, *Fair-0.6*, *Fair-0.8* and *Fair-1*. (2) FRESH [12]: Our previous work FRESH also adopts dynamic slot configuration. The slowstart is set to 1.

1) Performance: We compare our solutions with other schedulers in a Hadoop cluster with 20 slave nodes. We first show the makespan performance of *Lazy Start* and *Batch Finish* individually. Then we show the performance with the combination of these two techniques.

In each set of experiments, we consider both simple and mixed workloads. For each test of simple workloads, we create 8 jobs of the same benchmarks with the same input data. The size of each data size is 20 GB. Therefore, there are overall 160 GB data processed in each experiment and each job has 80 map tasks and 2 reduce tasks. All jobs are consecutively submitted to the Hadoop system with an interval of 2 seconds.

To further validate the effectiveness of OMO, we evaluate the performance with mixed workloads consisting of different benchmarks. We conduct eight job sets (Set A to H) of mixed jobs whose details are introduced in Table. III. Set A is mixed with all six benchmarks including both heavy-shuffling and light-shuffling ones. A recent trace from Cloudera shows that about 34% of jobs have at least the same amount of output data as their inputs [24]. So, during the 12 jobs in Set A, there are 4 heavy-shuffling jobs and 8 light-shuffling jobs. Each benchmark has two jobs, one with 20 GB input data and the other with 10 GB input data. Set B is a mixed job set with the two heavy-shuffling benchmarks: Terasort and Sequence Count. For each benchmark, there are 8 jobs, four with 20 GB input data and the other four with 10 GB input data. Set C is for scalability experiments of OMO.

TABLE III: Sets of Mixed Jobs

Job Set	Benchmarks	Job #	Input Size	map, reduce #
A	All benchmarks	6	20 GB	80, 2
		6	10 GB	40, 1
B	TeraSort, SeqCount	4	20 GB	80, 2
		4	10 GB	40, 1
C	All benchmarks	12	20 GB	80, 2
		12	10 GB	40, 1

Makespan Performance of Lazy Start: First, we disable *Batch Finish* algorithm in the Execution Predictor (**EP**) module in OMO to show the performance of *Lazy Start*. Fig. 3 shows the makespan performance of FAIR SCHEDULER, FRESH and *Lazy Start* in both simple and mixed benchmarks. Due to the page limit, we show the simple workloads evaluation results with three benchmarks.

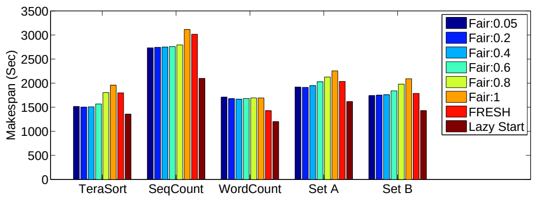

Fig. 3: Execution time under FAIR SCHEDULER, *FRESH* and *Lazy Start* (with 20 slave nodes)

In the simple workloads experiments, for heavy-shuffling benchmarks, such as Terasort and Sequence Count, FAIR SCHEDULER can achieve best makespan performance when the slowstart is set as 0.05 or 0.2 and *Lazy Start* improves 11.6% and 15.9% in makespan compared to the best one in FAIR SCHEDULER, and 24.5% and 23.8% compared to *FRESH*. For light-shuffling benchmarks, such as Word Count, FAIR SCHEDULER results in similar performance with different values of the slowstart. Since the shuffling time is short, *FRESH* achieves the good performance. On average, the makespan in *Lazy Start* is 27.8% shorter than FAIR SCHEDULER and 15.8% shorter than *FRESH*.

In the mixed workloads experiments, *Fair-1* yields the worst performance with different sets of jobs in FAIR SCHEDULER. In job set A, *Lazy Start* improves 18.1% of makespan compared to the best performance in FAIR SCHEDULER and 20.2% to *FRESH*. In job set B, *Lazy Start* improves 15.6% and 20.7% of makespan compared to the best performance in FAIR SCHEDULER and *FRESH*.

Makespan Performance of Batch Finish: To show the evaluation results of *Batch Finish*, we disable *Lazy Start* technique in OMO. Fig. 4 shows the makespan performance of *Fair-1*, *FRESH* and *Batch Finish* in both simple and mixed workloads. The value of the slowstart is 1 for all these schedulers. *FRESH* achieves better performance in makespan than *Fair-1*. On average, the makespan of *FRESH* is 7.26% and 12.3% less than *Fair-1* in simple and mixed workloads experiments. And *Batch Finish* decreases the makespan by 7.1% and 11% compared to *FRESH*.

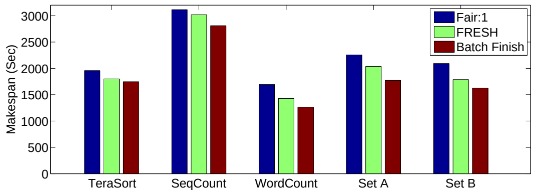

Fig. 4: Execution time under FAIR SCHEDULER, *FRESH* and *Batch Finish* (with 20 slave nodes)

Performance of OMO: Finally, we show the evaluation results of OMO, the combination of the two techniques above. First, we show the makespan performance of OMO with both simple and mixed workloads compared with other schedulers. Then we illustrate the breakdown execution time in both the map phase and the shuffling phase of each experiment.

The experiment results of three simple workloads and two sets of mixed workloads are shown in Fig. 5. *Fair:best* represents the best makespan performance in FAIR SCHEDULER. From the evaluation results, *Lazy Start* shows more significant improvement of makespan in heavy-shuffling benchmarks and *Batch Finish* shows better performance in light-shuffling benchmarks. OMO takes advantage of both techniques and achieves better makespan performance than either of them. On average, OMO improves 26% and 29.3% in makespan compared to *Fair:best* and *FRESH*.

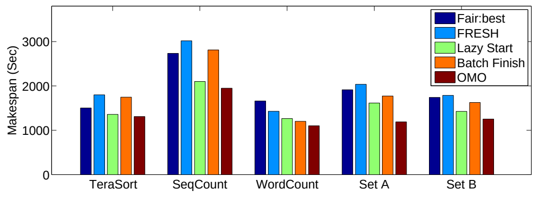

Fig. 5: Execution time under FAIR SCHEDULER, *FRESH*, *Lazy Start*, *Batch Finish* and *OMO* (with 20 slave nodes)

2) Comparison with YARN: In addition, Table. IV shows the comparison with YARN. Note that not all the benchmarks are available in the YARN distribution and we only use Terasort in our experiments with YARN. In each experiment, there are 8 Terasort jobs with 20 GB input data. For each YARN job, we set the CPU requirement of each task to be 2 core so that there are at most 4 tasks running concurrently at each node. YARN uses a new mechanism to assign reduce tasks. Basically, for each job, reduce tasks can be assigned according to the processing of the map phase (slowstart) and a memory limitation for reduce tasks in the cluster (maxReduceRampupLimit). We use the default configuration in the experiments. During the experiments, we set the memory demand of each map/reduce task of each job to be 2 GB, 4 GB, 6 GB and 8 GB, represented by YARN:2, YARN:4, YARN:6 and YARN:8 in Table. IV. In the YARN cluster, the makespan of YARN:2 is about 6.6% larger than the one with other memory requirements. And OMO improves the makespan by 17.3% compared to YARN:2 and 11.6% compared to the others. Note that YARN adopts a fine-grained resource management implying inherited advantages over the Hadoop system OMO is built on. But OMO still outperforms the YARN system. Our design can be easily extended and ported to the YARN system which is a part of our future work. When it is done, we certainly expect a more significant overall improvement.

TABLE IV: Execution time of Terasort benchmark under YARN and OMO (with 20 slave nodes).

	YARN:2	YARN:4	YARN:6	YARN:8	OMO
Makespan	1583s	1490s	1464s	1481s	1319s

3) Scalability: Finally, we test Set C on a large cluster with 40 slave nodes to show the scalability of OMO and the evaluation results are shown in Table V. We can observe a consistent performance gain from OMO as in the smaller cluster of 20 slave nodes with Set A. Compared to FAIR SCHEDULER and FRESH, OMO reduces the makespan by 37%. The improvement is consistent with the experiments with Set A and the 20-node cluster.

TABLE V: Makespan of Set H with 40 slave nodes

	Fair-default	Fair-1	FRESH	OMO
Makespan of Set D	1885s	2258s	2037s	1238s

Overall, OMO achieves an excellent and stable makespan performance with both simple workloads and mixed workloads of different sets of jobs.

VI. CONCLUSION

This paper studies the scheduling problem in a Hadoop cluster serving a batch of MapReduce jobs. Our goal is to reduce the overall makespan by the appropriate slot allocation. We develop a new scheme OMO which particularly optimizes the overlap between the map and reduce phases. We have implemented our solution on the Hadoop platform, and conducted extensive experiments with various workloads and settings. The results show a significant improvement on the makespan compared to a conventional Hadoop system, especially for heavy-shuffling jobs.

Acknowledgment: This project was supported by National Science Foundation grant CNS-1552525.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," vol. 51, no. 1, Jan. 2008, pp. 107–113.
[2] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/
[3] Fair scheduler. [Online]. Available: http://hadoop.apache.org/common/docs/r1.0.0/fair_scheduler.html
[4] Capacity scheduler. [Online]. Available: http://hadoop.apache.org/common/docs/r1.0.0/capacity_scheduler.html
[5] M. Isard, V. Prabhakaran, J. Currey *et al.*, "Quincy: Fair scheduling for distributed computing clusters," in *SOSP*, 2009, pp. 261–276.
[6] M. Zaharia, D. Borthakur *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010.
[7] J. Tan, X. Meng, and L. Zhang, "Performance analysis of coupling scheduler for mapreduce/hadoop," IBM T. J. Watson Research Center, Tech. Rep., 2012.
[8] ——, "Coupling task progress for mapreduce resource-aware scheduling," in *INFOCOM'13*, 2013, pp. 1618–1626.
[9] Y. Guo, J. Rao, and X. Zhou, "ishuffle: Improving hadoop performance with shuffle-on-write," in *ICAC*, 2013, pp. 107–117.
[10] Y. Yao, J. Wang, B. Sheng *et al.*, "Using a tunable knob for reducing makespan of mapreduce jobs in a hadoop cluster," in *CLOUD*, June 2013, pp. 1–8.
[11] ——, "Self-adjusting slot configurations for homogeneous and heterogeneous hadoop clusters," *Cloud Computing, IEEE Transactions on*, pp. 1–14, March 2015.
[12] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters," in *CLOUD*, 2014.
[13] M. Zaharia, A. Konwinski *et al.*, "Improving mapreduce performance in heterogeneous environments," in *OSDI*, 2008.
[14] Y. Yao, J. Tai, B. Sheng, and N. Mi, "Scheduling heterogeneous mapreduce jobs for efficiency improvement in enterprise clusters." in *IM*, pp. 872–875.
[15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *SIGMOD*, 2012, pp. 25–36.
[16] Y. Le, J. Liu, F. Ergün, and D. Wang, "Online load balancing for mapreduce with skewed data input," in *INFOCOM*, 2014.
[17] Apach Hadoop YARN. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[18] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *SOSP*, 2013, pp. 69–84.
[19] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek *et al.*, "Omega: Flexible, scalable schedulers for large compute clusters," in *EuroSys*, 2013, pp. 351–364.
[20] B. Hindman, A. Konwinski, M. Zaharia *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," ser. NSDI. USENIX, 2011, pp. 22–22.
[21] M. Isard, M. Budiu, Y. Yu *et al.*, "Dryad: Distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007, pp. 59–72.
[22] Nsf cloudlab. [Online]. Available: https://www.cloudlab.us/
[23] Purdue mapreduce benchmarks suite. [Online]. Available: http://web.ics.purdue.edu/~fahmad/benchmarks.htm
[24] S. A. Y. Chen and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads." ser. VLDB, 2012, pp. 1802–1813.