IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 23, 2021, accepted June 3, 2021, date of publication June 16, 2021, date of current version June 28, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3089907

Optimizing Internal Overlaps by Self-Adjusting Resource Allocation in Multi-Stage Computing Systems

ALLEN YANG[1], JIAYIN WANG[1], YING MAO[2], YI YAO[3], NINGFANG MI[4], (Member, IEEE), AND BO SHENG[5]

[1]Department of Computer Science, Montclair State University, Montclair, NJ 07043, USA
[2]Department of Computer and Information Science, Fordham University, New York City, NY 10458, USA
[3]Google, Mountain View, CA 94043, USA
[4]Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115, USA
[5]Department of Computer Science, University of Massachusetts Boston, Boston, MA 02125, USA

Corresponding author: Jiayin Wang (jiayin.wang@montclair.edu)

This work was supported in part by the National Science Foundation under Grant CNS-2018575 and Grant CNS-1552525, in part by the National Science Foundation Career Award under Grant CNS-1452751.

ABSTRACT With the rise of big data, more and more users will launch computing systems to process a large volume of data in various applications. A Scheduling algorithm is crucial to the performance of the processing platforms, especially when they are concurrently executing a batch of jobs. Such jobs usually represent multiple stages. Each stage produces the intermediate data which will be piped to the next stage for further processing. However, the scheduling problem in a big data computing system is different from the traditional multi-stage job scheduling problem as for any two consecutive stages, the later stage usually starts before the former stage is finished to ''shuffle'' the intermediate data. In this paper, we consider MapReduce/Hadoop as a representative computing system and develop a new strategy named OMO, Optimize MapReduce Overlap with a Good Start (Reduce) and a Good Finish (Map). A MapReduce job contains two consecutive phases: map and reduce. Our general target is to optimize the internal overlap between these two phases. There are two new techniques included in our solution, Lazy start of reduce tasks and Batch finish of map tasks, which aim to approach an effective alignment of the two phases based on the characteristics of the MapReduce process. OMO has been implemented on the Hadoop system with extensive experiments for performance evaluation. The results show that OMO's performance is superior in terms of total completion time (i.e., makespan) of a batch of jobs.

INDEX TERMS MapReduce jobs, Hadoop scheduling, reduced makespan, resource management.

I. INTRODUCTION

In the past few years, we have all witnessed the rise of big data and various processing platforms such as Hadoop [1], Mesos [2] and Spark [3], which have been widely adopted in both academia and industry for various applications. With more and more users launching a computing system to process a large amount of data, the data processing performance becomes significant for the computing systems.

The purpose of this paper aims to establish an efficient scheduling scheme in big data computing systems to better the resource consumption and reduce the makespan (i.e., the total completion length) of a set of applications. With finite resources available in a computing system, an advanced scheduling algorithm is imperative to the performance, especially when performing a batch of jobs in parallel. In the absence of proper management, the resources could be used inefficiently, which results in an extended completion time of the applications. A general big data processing job contains multiple processing stages, and each stage represents a generally defined data operation such as mapping, filtering, sorting, and merging. For each stage, a typical job consists of multiple tasks. In this case, the scheduling algorithm must consider both job-level and task-level management of resources. Additionally, unlike the traditional two-stage job scheduling problem, that each stage is independent, dependency usually exists

The associate editor coordinating the review of this manuscript and approving it for publication was Chi-Tsun Cheng.

between stages of every application in big data computing systems. For any two consecutive stages of a job, the output data of the former stages (i.e., the intermediate data) is piped to the input of the later stage. First, the later stage cannot complete before its former stage. However, the later stage can start to ''shuffle'' the intermediate data before the completion of the former stage. These kinds of factors cause the scheduling design to be particularly difficult and the current approaches have not completely discussed such matters.

In this paper, We take MapReduce [4] as a representative, which has developed to be a prevalent programming model for processing large data sets. Its open-source implementation Hadoop and a serial of corresponding co-systems are well applied in various fields. A general MapReduce job consists of two stages: map and reduce. And a map/reduce stage is made up of many identical map/reduce tasks. The map tasks take the raw data as input and process them concurrently. The intermediate data output from the map stage is in a form of $< key, value >$, which is shuffled to reduce tasks for computation. Furthermore, the final results are conducted by the reduce stage.

This work advances a novel technique, called OMO, that specifically targets on optimizing the overlap in between map and reduce stages. This overlapping period plays an essential part in MapReduce processing, particularly when the map stage produces large quantities of information for shuffling. An effective alignment of the map and the reduce stages could decrease the completion time of a job. Compare to the previous work, our solution takes much more dynamic aspects into consideration and assigns the resources according to the prediction of the future task execution as well as resource availability. Specifically, OMO consists of 2 new strategies, lazy start of reduce tasks, and batch finish of map tasks. The former strategy efforts to discover the most optimal timing to begin the reduce tasks in order to make sure that an adequate amount of time is allocated for reduce tasks to shuffle the intermediate data, whilst containers can be assigned in order to assist map tasks to the fullest. We present a new prediction model which is made to estimate the availability of resources in the near future which assists in making a scheduling decision. The later strategy aims to raise the trailing map tasks execution priority, this way they could complete in waves. Unlike the previous work that targets the wave-like execution during the map stage, we just concentrate on the last batch of map tasks. Both strategies capture features of the overlap in a MapReduce execution and accomplish good alignment of the map as well as the reduce stages.

In summary, the contributions of this paper include (1) We first develop a new monitoring component that records the amount of the resources released in the past. This information serves the new techniques in our solution to predict the resource release frequency in the future. (2) We create a novel strategy, lazy start of reduce tasks, that estimates the execution time of the map stage as well as the shuffling step of the reduce stage and further derives the best timing to begin the reduce stage so as to minimize the break from the end of the map phase to the end of the shuffling phase. (3) A new technique is developed, batch finish of map tasks, in order to mitigate the extra overhead caused by the misalignment of the tailing map tasks. (4) We present a complete implementation on a Hadoop platform. Experiment-based evaluation validates our design and shows a significant improvement in performance

The rest of this paper is arranged as follows. We present the related work of this paper in Section II. Section III presents the problem formulation demonstrated in this paper. In Section IV, we describe the particulars of the results. Section V illustrates the investigational results of OMO. Lastly, the summary of our work is provided in Section VI.

II. RELATED WORK

Among various computing systems, Hadoop and its next-generation Hadoop YARN systems have been widely used in both academia and industry. During multiple research fields in Hadoop and Hadoop YARN, job scheduling and resource allocation is crucial. By default, Fair Scheduler [5], Capacity Scheduler [6] and DRF [7] are embedded in the native Hadoop/Hadoop YARN method to guarantee each job can get a correct share of their available resources. To enhance the functioning of the computing methods, many scheduling works concentrate on unique directions. Some significant instructions and relevant functions are introduced as follows.

Some scheduling algorithms focus on job characteristics. To satisfy the predefined deadline, ARIA [8] and Deadline constraint Scheduler [9] allocate suitable resources to tasks. And Sparrow [10] concentrates on scheduling issues with a great number of little jobs.

Another scheduling direction takes considers resource allocation and aims to improve resource utilization of the cluster. In this region, RAS [11] increases resource utilization across machines and meets the job completion deadline. Our previous work FRESH [12] has developed dynamic slot configurations in Hadoop system based on FIFO and FAIR SCHEDULER. And we have developed dynamic resource management schemes in [13], [14] depending on the various workloads of different jobs. A fine-grained resource scheduling, Haste [15], concentrates on improving resource utilization by leveraging the information of requested resources, resource capabilities, and dependence between jobs. Similarly, a work named ABS-YARN [16] uses a state-of-the-art resource negotiator to quickly make decisions at the modeling level and reduce unneeded costs. Lastly in the topic of resource management, by harvesting run time latency, Toposch [17] can co-locate batches and microservices.

Data locality is also considered in the scheduling mechanisms to optimize the locality of jobs' input data in the distributed systems. Some task scheduling mechanisms focus on optimizing the locality of jobs' input data in the distributed file system. NKS [18], [19] claims data placement in the homogeneous environment. Some recent works [20]–[22] distribute input information in heterogeneous clusters based on the disk capacity of every node. However, the variety of

resource capabilities of every node, as well as the resource requirements of every task aren't considered gradually in such approaches. In other data locality approaches, dynamic voltage and frequency scaling (DVFS) [23] is used. Using DVFS, the CPU frequency can be dynamically changed for the current task based on the slack time from previously completed tasks.

A heterogeneous environment is normally in practice because of various hardware and software configurations in each node of the cluster. In this way, Tetris [24] packs jobs to machines according to their multiple resource requirements. Additionally, LATE [25], Hopper [26], Grass [27] and eSplash [28] are proposed to prevent unnecessary speculative executions to enhance the performance in heterogeneous clusters.

In addition to Hadoop/Hadoop YARN, Mesos [2] and Spark [3] are another two widely adopted computing systems. Mesos introduces a distributed two-level scheduling mechanism to share clusters and data efficiently between different platforms such as MapReduce, Dryad [29] and others. Spark is a computing framework that offers shared distributed memory based on the resilient distributed dataset (RDD). Our work can be integrated into these platforms as a single low-level scheduler.

III. PROBLEM FORMULATION

In this paper, we consider Hadoop system as the service platform for MapReduce. There are two main branches of Hadoop frameworks available currently, Hadoop [30] and Hadoop YARN [31]. We will formulate our solutions in both systems. Our implementation is based on the first generation of Hadoop. However, the techniques we present in Hadoop can be easily extended to Hadoop YARN. Furthermore, we will also compare the performance with Hadoop YARN within our evaluation (Section V).

In our problem setting, we consider a Hadoop cluster consisting of a head node and multiple worker nodes. Each node contains multiple containers which indicate its capacity of serving tasks. Each container can either be set as a map container or reduce container to serve a map task or reduce task, respectively. Furthermore, we assume that a batch of n jobs are assigned to the cluster with totally S containers for processing. In the native Hadoop system, the numbers of map containers and reduce containers in the cluster have to be specified by the cluster administrator. And a map/reduce container is dedicated to serve map/reduce tasks throughout the lifetime of the cluster. In Hadoop YARN, the numbers of containers can be calculated based on the available resources of the system and the resource requests of each job. We will illustrate this equation in Section IV. In this paper, however, we present a dynamic container configuration scheme which has been established in our prior work [12], [32], where the scheduler can decide to set a container as a map container or reduce container during the job execution dynamically. In this case, no configuration of the number of map/reduce containers is necessary before setting up the cluster. The allocation of containers to serve as either map or reduce tasks will be done dynamically on the fly. The benefits of dynamical container configuration have been shown in the prior work [12], [32]. The details of its implementation are omitted in this paper since our focus is a different scheduling strategy based on the dynamic container configuration. Essentially, we aim to develop a scheduling algorithm that allocates tasks to appropriate containers to minimize the makespan of the given set of MapReduce jobs. Table 1 displays the list of notations that are used throughout the paper.

TABLE 1. Notations.

n	number of jobs
D	number of active jobs
S	number of containers in the cluster
J_i	i-th job
m_i	number of map tasks of i-th job
r_i	number of reduce tasks of i-th job
R_{ik}	i-th available resource in the worker node k
rm_{ij}	i-th resource requirement for the map task of job j
rr_{ij}	i-th resource requirement for the reduce task of job j
F_o	observed container release frequency
A_o	observed number of *available containers*
F_e	estimated container release frequency
A_e	estimated number of *available containers*
T_m	completion time of a map task
T_s	completion time of the shuffling step
T_w	length of a historical window
R_t	number of containers released in the t-window
f_t	container release frequency in the t-th window, $f_t = R_t/T_w$
a_t	number of available containers in t-th window
m_i'	number of pending map tasks of job J_i
β	gap from the finish of map phase to the finish of shuffling phase
d	data size generated by one map task
B	network bandwidth

IV. OUR SOLUTION: OMO

We introduce our solution OMO in this section. Our target is reducing the overall execution time of MapReduce jobs. Two new techniques, lazy start of reduce tasks and batch finish of map tasks are developed in OMO. In the rest of this section, firstly, we introduce a monitor module which serves for both techniques as a building block. Secondly, we describe the details of the two techniques individually. Finally, the complete algorithm integrating both of them is introduced. The whole solution is primarily developed as a new Hadoop scheduler. The details of the implementation are introduced in Section V.

A. CONTAINER RELEASE FREQUENCY

Both of our new strategies depend on an essential parameter which is the estimated frequency of container release in the MapReduce system. This parameter represents the future resource availability of the system, which is significant for Hadoop scheduler to make decisions of resource allocation. Although it is an essential factor for the system performance, all the prior work ignored it. Before discussing the details

of our solution in the following subsection, we first present the fundamental method we adopt to estimate the container release frequency.

There are two parameters, F_o and F_e defined in OMO to represent the observed container release frequency and estimated container release frequency, i.e., F_o or F_e containers released per time unit. F_o is a measurement value acquired by monitoring the job execution, and F_e is the estimation of the future release frequency that will be used by the scheduler. Also, a new concept of **available containers** is introduced to describe the containers that might be potentially released soon. Available containers contain both the containers serving map tasks, and the containers serving a jobs reduce tasks if its map stage has been completed. In other words, the available containers exclude the containers serving the reduce tasks of a job with unfinished map tasks in which instance the release time of the containers is undetermined. In OMO, we suppose that for a particular circumstance, the container release frequency is proportional to the number of available containers.

Specifically, we keep track of a historical window to measure the number of released containers and the number of available containers in the window indicated by R_t and a_t (for the t-th window), respectively.

In the first generation of Hadoop system, since the total number of containers is prior-set by the system administrator, R_t can be obtained by checking the available containers of the system. For Hadoop YARN, considering a cluster with M worker nodes and N types of resource (e.g., number of CPU cores and available memory), R_{ik} represents the i-th resource available in the worker node k. For a job j, req_i indicates the i-th resource requirement of a single task, which is either rm_{ij} for a map task and rr_{ij} for a reduce task. The number of containers released in the t-window can be represented as:

$$R_t = \sum_{k=1}^{M} min_{i \in N} \lfloor \frac{R_{ik}}{req_i} \rfloor$$

Assuming that the window size is T_w seconds, the container release frequency in this window will be $f_t = \frac{R_t}{T_w}$ and the ratio between container release frequency and the number of available containers 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 particular time period. For each window t, we thus record the (f_t, a_t) pairs and obtain the average value of the container release frequency F_o and the average number of available containers denoted by A_o. We use the typical technique of exponential moving average (EMA) to capture the dynamics throughout the execution,

$$F_o(t) = \beta \cdot f(t) + (1 - \beta) \cdot F_o(t - 1),$$
$$A_o(t) = \beta \cdot a(t) + (1 - \beta) \cdot A_o(t - 1),$$

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.

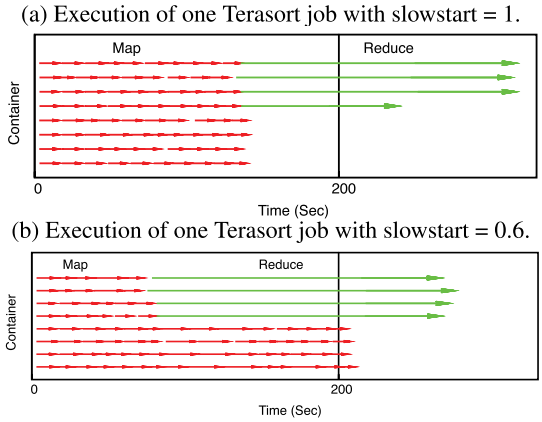


FIGURE 1. Container allocation of one Terasort job with dynamic container configuration.

When estimating F_e for a future scenario, we first need to identify the number of available containers (A_e) in that scenario. After that, based upon the assumption that the container releasing frequency is proportional to the number of available containers, we can calculate F_e as

$$F_e = \frac{F_o \cdot A_e}{A_o}. \quad (1)$$

This component of estimating the container release frequency will be used by both of our new techniques which will be described later in this section. We will present 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 overlap between the map and reduce stages of MapReduce by adjusting the start time of the reduce stage of the MapReduce jobs. Firstly, we show how a conventional Hadoop system manages the overlap period. Then, we will introduce the motivation for our design. Finally, the intuitions of our solution will be presented with the details of the algorithm.

1) MOTIVATION

The overlap between the map and reduce stages is an essential feature of MapReduce jobs. In MapReduce process, the reduce stage usually begins before the completion of the map stage, i.e., some reduce tasks may be simultaneously running with the map tasks of the same job. The advantage of this design is to enable the reduce tasks to start shuffling (i.e., preparing) the intermediate data (partially) generated by map tasks before the whole map stage is done to save the execution time of the reduce tasks. A system parameter *slowstart* in Hadoop system is set to indicate when to start the reduce stage. Specifically, *slowstart* is a fractional value that represents the lower bound of the map stage's progress. The reduce stage is now allowed to start until the map stage's progress exceeds *slowstart*.

Table 2 shows some simplified experimental results of execution times with different values of slowstart. First, we conduct one Terasort job in a Hadoop cluster with two slave nodes (Amazon AWS m3.xlarge instances) and each slave node is configured with 2 map containers and 2 reduce containers. The input data is 8GB wiki category links data and there are 80 map tasks and 4 reduce tasks created in the job. Then we conduct 3 Terasort jobs with 10 slave nodes to show the results of multiple jobs.

TABLE 2. Execution times of 1 and 3 Terasort jobs with different slowstart values in Traditional Hadoop systems.

Slowstart	0.5	0.6	0.7	0.8	0.9	1
Execution time of 1 job	309	307	311	312	320	336
Execution time of 3 jobs	291	259	275	272	283	317

C. LAZY START OF REDUCE TASKS

The goal of our first technique is to optimize the overlap between the map and reduce stages of MapReduce by adjusting the start time of the reduce stage of the MapReduce jobs. Firstly, we show how a conventional Hadoop system manages the overlap period. Then, we will introduce the motivation for our design. Finally, the intuitions of our solution will be presented with the details of the algorithm.

1) MOTIVATION

The overlap between the map and reduce stages is an essential feature of MapReduce jobs. In MapReduce process, the reduce stage usually begins before the completion of the map stage, i.e., some reduce tasks may be simultaneously running with the map tasks of the same job. The advantage of this design is to enable the reduce tasks to start shuffling (i.e., preparing) the intermediate data (partially) generated by map tasks before the whole map stage is done to save the execution time of the reduce tasks. A system parameter *slowstart* in Hadoop system is set to indicate when to start the reduce stage. Specifically, *slowstart* is a fractional value that represents the lower bound of the map stage's progress. The reduce stage is now allowed to start until the map stage's progress exceeds *slowstart*.

Table 2 shows some simplified experimental results of execution times with different values of slowstart. First, we conduct one Terasort job in a Hadoop cluster with two slave nodes (Amazon AWS m3.xlarge instances) and each slave node is configured with 2 map containers and 2 reduce containers. The input data is 8GB wiki category links data and there are 80 map tasks and 4 reduce tasks created in the job. Then we conduct 3 Terasort jobs with 10 slave nodes to show the results of multiple jobs.

In practice, it is incredibly difficult for users to define the value of slowstart before launching the cluster. In addition, the pre-configured value cannot be the optimum for numerous job workloads. In this case, we develop a new technique, lazy start of reduce tasks, to optimize the performance. We hold off the start of the reduce stage as much as possible until intermediate data shuffling will incur an extra delay in the process. Ideally, a perfect alignment of the map stage and reduce stage happens when the last reduce task completes the data shuffling right after the last map task is finish. However, simply utilizing the slowstart threshold is hard to accomplish the best performance since it relies on not just the progress of the map stage, but also various other factors such as the map task execution time as well as shuffling time. In the remainder of this subsection, we describe our solution to determine the start time of the reduce stage during the execution of the job. First, we present an algorithm that considers the single job execution to illustrate our design intuition, and then extend it for numerous job execution.

TABLE 3. Execution times of 1 and 3 Terasort jobs with different slowstart values and dynamic container configuration.

Slowstart	0.5	0.6	0.7	0.8	0.9	1
Execution time of 1 job	287	277	278	299	306	316
Execution time of 3 jobs	255	234	262	310	335	358

2) SINGLE JOB

The original design of the slowstart parameter in Hadoop shows that the progress of the map stage is definitely significant for determining when to begin the first reduce task. However, the best start time of the reduce stage also relies on a list of factors as follows.

(1) Shuffling time: This can be conducted by the size of intermediate data created by map tasks and the network bandwidth of the system. Intuitively, a job producing more intermediate data in the map stage requires a longer shuffling time in its reduce stage. Therefore, we need to begin the reduce tasks earlier. Generally, the size of the intermediate data is proportional to the map stage progress. As a result, a respectable estimation of the overall intermediate data can be completed by monitoring the completed map tasks. (2) Map task execution time: The advantage of beginning reduce tasks prior to the end of the map stage is to overlap the shuffling in reduce stage with the execution of the remainder of the map tasks (the last couple of waves of map tasks). Thus, given a specific shuffling time, we prefer to begin the reduce stage later, if each map task needs a longer time to complete. (3) Frequency of container release: The frequency of a container released and available in the cluster is also a crucial factor. In both map and reduce stages, the trailing tasks have an important effect on the overlapping period. Once we specify a target begin time for the last reduce task, we can make use of the information of container release frequency to conduct when we should begin the reduce stage.

In our solution, we monitor the three parameters listed above to determine the optimal start time of the reduce stage. Before introducing the detailed algorithm, a design principle is presented and proved.

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

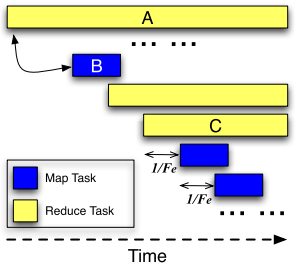


FIGURE 2. Illustration of the proof.

Proof: We can prove this principle by contradiction. Assume that the best arrangement does not follow this principle. In other words, there are some map tasks launched, after the first reduce task is begun, and before the last couple of reduce tasks are started. We identify the last such map tasks, e.g., task B in Fig. 2, and the first reduce task, e.g., task A in Fig. 2. 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 completion time of the map stage could become earlier because task A occupies a slot at a later time point and that slot could serve map tasks before task A is begun. Meanwhile, the shuffling performance maintains the same, i.e., the gap from the end of the map stage 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. 2 Consequently if we consider the completion 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 eventually obtain a solution where reduce tasks are consecutively executed with no interruption of map tasks. □

Based on the above principle, we examine the gaps from the ends of the map stage to the shuffling phase and obtain the optimal start time of the reduce stage to lessen this gap. Assume the running job has m map and r reduce tasks. If a container frees up, our scheduler needs to assign it to a new task. When the reduce stage has not started, there are just two options to serve a map task or to serve a reduce task which starts the reduce stage. Let β be the time gap from the end of the map stage to the end of the shuffling stage (see Fig. 3) and assume that variable x represents the number of pending map tasks. We first derive β as a function x and afterward decide the time to start the reduce stage. Also, we use T_m to represent the average execution time of a map task, and T_s to indicate the estimated shuffling time of the last reduce task.

Considering that all reduce tasks are executed consecutively and reduce task containers will not be released till the completion of a job, after the last reduce task is allocated, the number of available containers becomes $S - r$. Consequently, the estimated frequency of container release is decreased to $F_e = \frac{F_o \cdot (S-r)}{S}$. Assuming that the containers are released at a constant rate with an interval of $\frac{1}{F_e}$ between

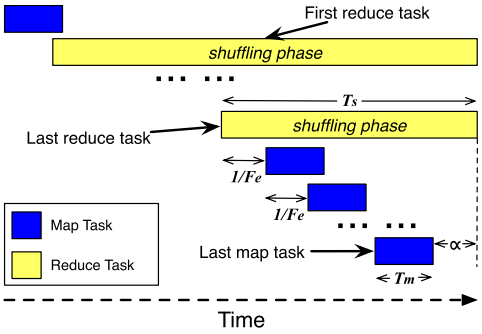


FIGURE 3. Lazy start of reduce tasks: illustrating the alignment of map stage and shuffling phase.

any two consecutive releases, the execution time for the remaining map tasks can be expressed as $\frac{x}{F_e} + T_m$ (see Fig. 3). Therefore, we express β as a function of x:

$$\beta = T_s - (\frac{x}{F_e} + T_m) = T_s - (\frac{x \cdot S}{F_o \cdot (S - r)} + T_m), \quad (2)$$

where F_o is a measured value as defined in Section IV-C and T_m records the average execution of a map task. Both F_o and T_m are updated once a task is completed. 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). Thus, T_s can be expressed as:

$$T_s = \frac{d \cdot m}{B \cdot r}$$

Throughout the execution of a job, our scheduler forms β as the function of x and then calculates the values with various x whenever a container is released. When the actual number of the pending map tasks m' satisfies the equation as shown below, the reduce stage will begin, i.e., the first reduce task will be allocated to the currently available container:

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

3) MULTIPLE JOBS

Furthermore, we extend our design to serve multiple MapReduce jobs. Our scheduler is built on Fair scheduler which equally allocates containers to all the active jobs. Specifically, if there are S containers in the cluster and D jobs running concurrently, each job can occupy $\frac{S}{D}$ containers and the effective container release frequency for each job is $\frac{F_o}{D}$.

Following Eq. (2), we calculate the gap β for each job J_i,

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

where x_i represents the number of the pending map tasks of J_i, and parameters $T_s(i)$ and $T_m(i)$ are particular to J_i. The estimated container release frequency F_e can be estimated as

$$\frac{F_o \cdot A_e}{A_o \cdot D},$$

where F_o and A_o are common parameters for all the jobs. With multiple jobs executing, A_o may not be the same as S as in the case of single job execution. When calculating β 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,

$$\beta = T_s(i) - (\frac{x_i \cdot A_o \cdot D}{F_o \cdot (A_o - r_i)} + T_m(i)). \tag{3}$$

Finally, the reduce stage should be started when the number of pending map tasks m_i' satisfies the following equation:

$$m_i' = \arg \min_{x \in [m_i', m_i]} \beta.$$

Multiple jobs may satisfy the above equation, in which case our scheduler will allocate the container to the job that has occupied the fewest containers among all the candidates.

The details of our algorithm are indicated in Algorithm 1. Function **LazyStartReduce()** is intended to return the index of the job that should begin its reduce stage. If there is no candidate, the function will return "−1". The variable *res* records a list of candidate job indexes. Specifically, lines 1–6 sets the number of active jobs (D). Lines 7–21 enumerate all the running jobs that have not begun their reduce stages, and use Eq. (3) to decide whether there are candidate jobs to begin the reduce stage. Eventually, if there are multiple candidates in *res*, the algorithm returns the index of the job with the minimum occupation on containers (lines 22–27).

Algorithm 1 Function LazyStartReduce()

```
1:  D = 0, res = {}
2:  for i = 1 to n do
3:    if J_i is running then
4:      D ← D + 1
5:    end if
6:  end for
7:  for i = 1 to n do
8:    if J_i is executing and has not begun its reduce stage
      then
9:      β_OPT = T_s(i) − ((m'_i·A_o·K)/(F_o·(A_o−r_i)) + T_m(i))
10:     selected = true
11:     for x = m'_i + 1 to m_i do
12:       β = T_s(i) − ((x_i·A_o·K)/(F_o·(A_o−r_i)) + T_m(i))
13:       if β < β_OPT then
14:         selected=false; break;
15:       end if
16:     end for
17:     if selected == true then
18:       res = res + {i}
19:     end if
20:   end if
21: end for
22: if res is empty then
23:   return −1
24: else
25:   sort all job indexes in res in the ascending order of the
      number of occupied containers
26:   return the first index in the sorted list
27: end if
```

D. BATCH FINISH OF MAP TASKS

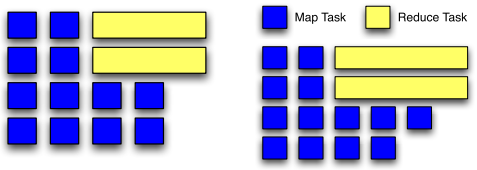
Our second technique is called batch finish of map tasks, which is designed to improve the performance of the map stage by assigning the trailing map tasks to be completed in one batch. In this subsection, we first indicate how the alignment of map tasks impacts the execution time of a MapReduce job. Then we introduce our algorithm to improve the performance.

1) MOTIVATIONS

In the design of a Hadoop system, the map tasks are expected to be complete in waves to accomplish good performance. In the case of unaligned map tasks, specifically the trailing map tasks, a delay may occur which significantly affects the overall job execution time. With a misalignment, the last couple of pending map tasks will cause an additional round of execution in the map stage causing the initiated reduce tasks to wait for the completion of these map tasks. As a result, the occupied slots will be used less efficiently.

However, in practice, map tasks are hardly aligned as waves because the number of map tasks may not be a multiple of the number of the allocated slots. In a conventional Hadoop system, the number of map slots in the cluster can be configured as a parameter, which is unknown to the user who submits the job. In our dynamic slot configuration solution, the same problem still exists and the number of slots allocated to map tasks is even more uncertain as there are no reserved slots for map or reduce tasks. Additionally, when multiple jobs are running concurrently, the misalignment of map tasks is more severe because of the heterogeneous execution times of map and reduce tasks, as well as different scheduling policies. Fig. 4 shows a simplified example of running one MapReduce job. Assume that each map task can be completed in a time unit and each reduce task also requires a one-time unit to complete after its shuffling stage. Fig. 4a illustrates the execution process with 12 map tasks where the map stage is finished with 4 rounds and the total execution time is 5-time units. In Fig. 4b, however, there is an additional map task causing an extra round in the map stage. The total execution time becomes 6-time units, which is a 20% increase compared to Fig. 4a.

Fig. 5 shows an experiment with three jobs: terasort, wordcount, and k-means in a Hadoop cluster with 4 slave nodes. Each node has 3 map containers and 1 reduce container. The input data of each job is 8GB. There are 32 map tasks in the k-means and wordcount, and 50 map tasks for the terasort job. The number of reduce tasks for each job is 1. These three jobs are running with Fair scheduler and the slowstart is set to 0.6. The X-axis is execution time and the Y-axis shows the task containers in the cluster. container 1 to 12 are map containers



(a) Map tasks are finished in 4 waves

(b) The tailing map task incurs an additional round

FIGURE 4. Example: one additional map task increases the execution time of the given job by 20%.

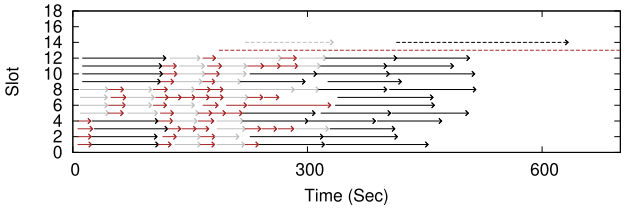


FIGURE 5. Experiment with 3 jobs in a Hadoop cluster with Fair scheduler: Solid lines represent map tasks and dashed lines represent reduce tasks.

and 13 to 16 are reduce containers. Apparently, the map tasks of all three jobs are not well aligned after the first wave.

Therefore, in our solution, we aim to allocate the trailing map tasks in a batch to solve this problem. Our intuition is to adjust the Hadoop scheduler to increase the priority of the trailing map tasks when allocating tasks to available containers, even though it may violate its original policy. The decision relies on the number of pending map tasks and an estimate of the future container release frequency. Essentially, given the number of the pending map tasks of a job, if the scheduler predicts that the cluster will release a sufficient number of containers in a short time window, it will preserve those future containers to serve the pending map tasks. The benefit is that the target job's map stage can be finished faster and the containers occupied by its reduce tasks will become available more quickly. The drawback is that a possible delay may happen to other active jobs because those reserved future containers could otherwise serve them.

2) ALGORITHM DESIGN

In this subsection, we will introduce the algorithm design of OMO. First, the candidate jobs for batch finish of map tasks must have started their reduce stage. Otherwise, if we apply this strategy to the jobs which have not begun their reduce stage, the result is equivalent to beginning their reduce stages after the map stages with no overlap. Second, for each candidate job, our scheduler evaluates both the benefit and drawback of finishing the pending map tasks in a batch. After that, we choose the job that can generate the most reward to apply this strategy.

Specifically, we analyze each job that has started its reduce stage and decide whether the batch finish of its map tasks is appropriate. We first examine the performance under regular Fair scheduler and then compare to the case, if we complete 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 container release frequency F_o, a container will be assigned to job J_i every $\frac{K}{F_o}$, where D is the number of active jobs in the cluster. Under Fair scheduler, J_i will finish its map phase in t_{fair} time units,

$$t_{fair} = \frac{K \cdot m'_i}{F_o} + T_m(i). \tag{4}$$

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

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

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 containers become available and the container release frequency will be increased to $F_e = \frac{F_o \cdot (A_o + r_i)}{A_o}$. To contribute s containers to other jobs, the time required is

$$t_{batch} = \frac{s}{F_e} = \frac{s \cdot A_o}{F_o \cdot (A_o + r_i)}. \tag{5}$$

If $t_{batch} < t_{fair}$, the batch finish of map tasks becomes superior since it accomplishes the same scenario, i.e., J_i's map stage is completed and all the other jobs get s containers, in a shorter time period. The details are indicated in Algorithm 2. Function **BatchFinishMap** returns the index of the job that needs to apply the batch finish to its pending map tasks. Variable c represents the index of the candidate job. The function will return "-1" if there is no such candidate. Generally, the algorithm consists of a loop (lines 7–14) that enumerates every active job and calculates t_{fair} and t_{batch} to further decide whether it is worthy to apply the strategy. Variable max is defined to temporarily record the current maximum difference between t_{fair} and t_{batch}. Ultimately, the index of the job with the maximum benefit will be returned.

E. COMBINATION OF THE TWO TECHNIQUES

Finally, our scheduler integrates our two strategies indicated above into the baseline Fair scheduler for the execution of all jobs. The possible conflict between these two techniques is another challenge in the design. For instance, if there is a container released, the first technique may determine to assign this container to begin a jobs reduce phase, i.e., allocating a reduce task to it, but the second technique may prefer to reserve this container and also the following consecutive containers to finish another job's pending map tasks in a batch. In our solution, a simple strategy is used to address this issue: if there is a conflict, we provide a higher priority to the technique of lazy start of reduce tasks. The intuition is that when a new job begins its reduce stage, the decision of batch finish of map tasks could be affected because there is a new candidate for applying the technique.

Algorithm 2 Function BatchFinishMap ()

$D = 0, max = 0, c = -1$
for $i = 1$ to n **do**
  if J_i is running **then**
    $D \leftarrow D + 1$
  end if
end for
for $i = 1$ to n **do**
  if J_i is executing and has begun its reduce stage **then**
    Calculate t_{fair} and t_{batch} as in Eq. (4) and Eq. (5)
    if $t_{batch} < t_{fair}$ and $t_{fair} - t_{batch} > max$ **then**
      $max = t_{fair} - t_{batch}, c = i$
    end if
  end if
end for
return c

Specifically, we integrate Algorithm 1, Algorithm 2, and the Fair scheduler in Algorithm 3. When a container is released, the algorithm first calls the function LazyStartReduce(). If it selects a job that should start its reduce phase, the released container will be assigned to the job's first reduce task. If the function LazyStartReduce() does not find a candidate, then the algorithm considers the batch finish of map tasks. Similarly, if the function BatchFinishMap returns a candidate job, the released container will be assigned to serve a pending map task of the job. Finally, if neither of our new techniques finds a candidate job, our algorithm invokes the default policy in Fair scheduler.

Algorithm 3 Container Allocation

$i =$ LazyStartReduce()
if $i \geq 0$ **then**
  Allocate the released container to J_i's reduce task
else
  $i =$ BatchFinishMap();
  if $i \geq 0$ **then**
    Allocate the released container to J_i's map task
  else
    $i =$ FairScheduler();
    Allocate the released container to J_i
  end if
end if

V. PERFORMANCE EVALUATION

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

A. SYSTEM IMPLEMENTATION

We implemented our new scheduler OMO on Hadoop version 0.20.2 by adding a set of new components to support our solution. Fig 6 shows the details of the system implementation. The shadow parts are new modules created and other parts are existing modules in the native Hadoop system and

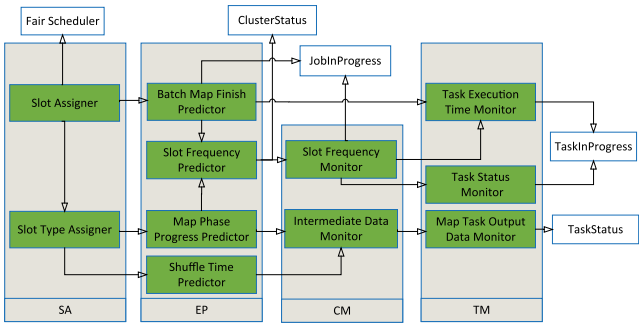


FIGURE 6. System implementation.

recalled by OMO. First, four new modules are developed into JobTracker: the Task Monitor (**TM**), the Cluster Monitor (**CM**), the Execution Predictor (**EP**) and the container Assigner (**SA**). **TM** records (1) the size of the intermediate data generated by each map task, (2) the running progress of each map/reduce task, (3) the completion time of each finished map/reduce task and (4) the numbers of the completed and pending map/reduce tasks of every job. Based on the statistics from **TM** and the number of concurrent jobs in the cluster from *JobInProgress*, **CM** is responsible for gathering the number of released containers in the cluster in real-time and updates the container release frequency dynamically. In addition, **CM** also collects the total size of intermediate data output by the map stage of every job.

According to the data listed above, **EP** predicts (1) the overall container frequency of the cluster, (2) the optimal point to execute the algorithm of the batch finish, (3) the time remaining of the map stage, and (4) the shuffling time of each job. In addition, **SA** is responsible for allocating a map/reduce task to every released container in the cluster by applying Algorithm 3 introduced in Section IV with the information obtained from **EP**.

Also, it should be noted that we have modified the fairness calculation in the traditional Fair scheduler, where the fairness of map containers and reduce containers are separately considered. Since we use dynamical container configuration, a container does not exclusively belong to either map or reduce container category. Therefore, we consider the total number of the containers assigned to each job and use it to calculate the deficiency for the Fair scheduler to make the scheduling decision.

B. TESTBED SETUP AND WORKLOADS

First, the cluster setting, and the workloads for the evaluation are presented.

1) HADOOP CLUSTER

The experiments were conducted on the NSF CloubLab platform at the University of Utah [33].he hardware specifics of each server are as follows. 8 ARMv8 cores at 2.4GHz, 64GB ECC memory, and 120GB storage. Two Hadoop clusters are created, a small one with 20 slave nodes and one large one with 40 slave nodes. There are 4 containers configured in each slave node.

OMO is compared to other scheduling algorithms on the 20-slave node system. The other clusters are launched to assess the expandability of OMO. In addition, another Hadoop YARN cluster with 20 slave nodes is set up as a control group. Rather than setting up specifics, each node claims 8 CPU cores along with 40 GB of memory as the resource volume.

2) WORKLOADS

We consider typical Hadoop benchmarks as the workload for evaluation with large data sets. Specifically, we choose six data sets in the experiments. The data sets are as follows: 10/20GB Wikipedia category links data, as well as 10/20GB movie score data from Netflix, and lastly 10/20 GB synthetic data. The Wikipedia data contains page category information. The Netflix data contains user ratings. Finally, the synthetic data was generated using TeraGen within Hadoop. Using the following Hadoop benchmarks from Purdue MapReduce Benchmarks [34] OMO is evaluated.

- *Terasort*:Sorts (key,value) pairs of data by the key with the synthetic data as input.
- *Sequence Count*:Counts all distinct sets of three successive words of each document with multiple Wikipedia files as input.
- *Word Count*: Counts the number of each word with multiple Wikipedia files as input.
- *Inverted Index*:Creates word to document indexing with multiple Wikipedia files as input.
- *Classification*:Classifies movies according to their ratings with the Netflix movie ranking data as input.
- *Histogram Movies*:Produces a histogram of the number of movies in each user ranking with the Netflix movie ranking data as input.

Table 4 illustrates the details of all six benchmarks in our experiments, consisting of the benchmark's name, input data type/size, intermediate data size, as well as the number of map and reduce tasks.

TABLE 4. 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

C. VALIDATION OF OMO DESIGN

The design of OMO mainly relies on two new techniques: container release rate prediction, and batch finish of the tailing map tasks. In this subsection, the experimental results that validate our design intuition are presented.

Fig. 7 shows the container release rate derived from an experiment with 12 mixed MapReduce jobs on a cluster of 20 nodes using Fair scheduler. A time window of 10 seconds is considered to receive the histograms of the container releases. In addition, we apply the container release rate estimation algorithm used in OMO and present the estimated value as the curve Estimation in the following Fig. 7. Overall, we observed that our estimation of the container release rate is close to the real value in the experiment. From the experimental trace, we find that the container release rate shows a high variance as we can see spikes in the curve. The estimation of OMO may not accurately predict the change when there is a big gap between two consecutive time windows. However, our algorithm usually catches up with the trend quickly in the next time window mitigating the negative impact on the performance. Above all, we believe that predicting resource availability in a large-scale cluster with a complex workload is a valid and feasible mechanism in practice. Later in this section, we will show the performance benefit gained from this technique.

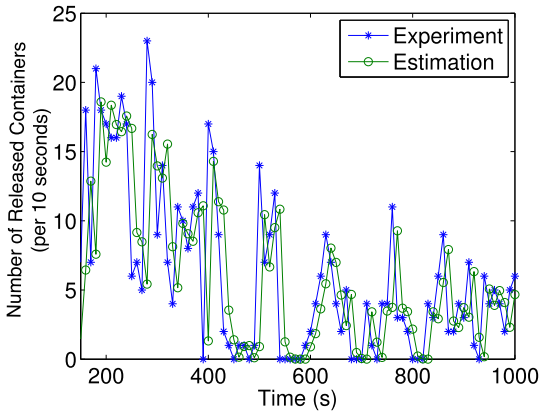


FIGURE 7. Container release prediction.

The other main technique in our solution is the batch finish of map tasks. The design of OMO primarily emphasizes the last batch of map tasks. 8 compares OMO to Fair scheduler with a set of 12 mixed MapReduce jobs running on a 20-node cluster. We define the last batch in a job as the last set of map tasks whose finish times are within 10 seconds. In 8, we observe that with Fair scheduler, the last batch of all the jobs contains no more than 10 tasks and half of the jobs that have less than 5 map tasks in the last batch. With OMO, on the other hand, the last batch of map tasks is usually much bigger. Especially for short map tasks, e.g., job 2 and job 3, OMO gives the map tasks higher priority and purges them quickly. In addition, there are cases where OMO yields an even fewer number of map tasks in the last batch than Fair scheduler. This is caused by the complicity and dynamics during the execution of the set of mixed jobs. Other factors may conflict with this technique when the scheduler makes the decision, e.g., starting a reduce task due to the lazy start algorithm, and starting a duplicate task for a failed or stale execution. Overall, the batch finish of map tasks in OMO is effective

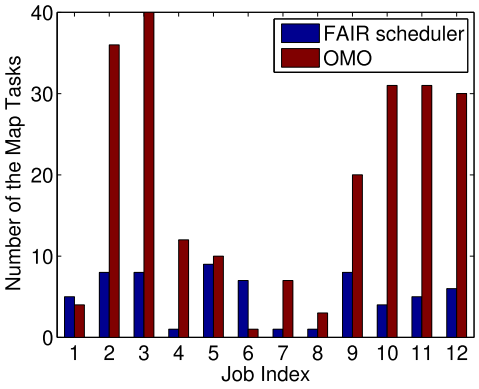


FIGURE 8. Last batch of map tasks.

from the experimental results. In the next section, we will show how it helps improve the overall performance.

D. EVALUATION

In this subsection, the performance of OMO is shown and it is compared to preexisting solutions. Mainly, OMO is compared to the following alternate scheduling algorithms listed in previous work

- Fair scheduler: Using the Hadoop's default container configuration, i.e, each slave has 2 map containers and 2 reduce containers per slave. The slowstart 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.
- FRESH [12]: One of the previous works, FRESH (Fair and efficient container configuration and scheduling for Hadoop Clusters) also implements dynamic container configuration. The slowstart is set to 1.

In Summary, our results include the following aspects:

- **Container Allocation**: We illustrate the detailed container allocation of OMO and other alternative schemes in Hadoop.
- **Performance**: We show the performance of the lazy start of reduce tasks, batch finish of map tasks, and the combination of such two techniques, represented as Lazy Start, Batch Finish and OMO. Given a batch of MapReduce jobs, our performance metrics are the makespan (the finish time of the last job) and the breakdown execution times of both the map phase and the shuffling phase. All experiments are conducted with simple workloads and mixed workloads.
- **Comparison to YARN**: We also compare some tests with the Fair scheduler in Hadoop YARN.
- **Scalability**: Finally, we show the scalability of OMO by experiments with different settings of input data sizes, job numbers, and cluster sizes.

1) CONTAINER ALLOCATION

First, we use TeraSort as an example to illustrate the container allocation of OMO and other alternative schedulers in Hadoop (Fig. 9). In each test, we use 8 jobs with the Terasort benchmark. The input data size is 20 GB for each job. There

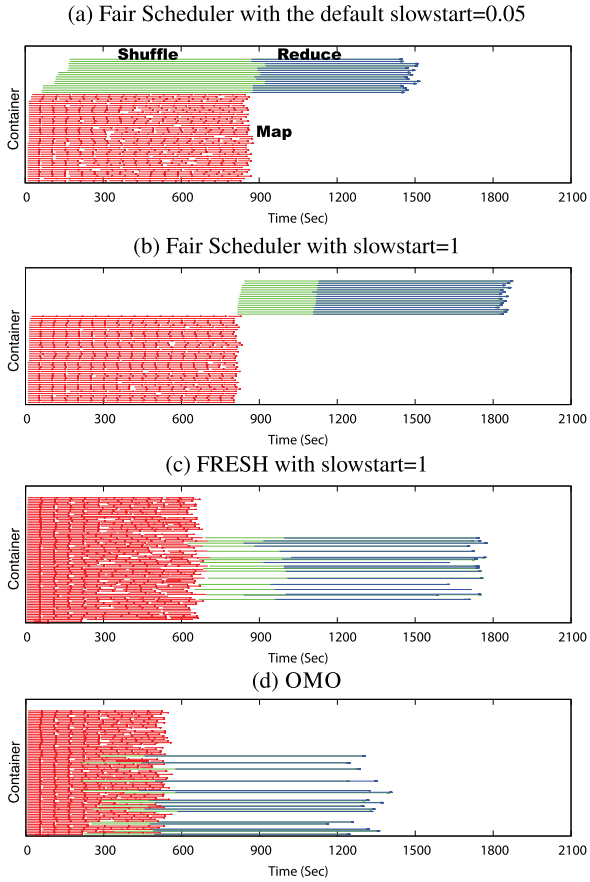


FIGURE 9. Container allocation in the execution of 8 Terasort jobs.

are of 160 GB data totally in each experiment. The X-axis is the execution time and the Y-axis shows all the containers in the cluster. The red lines show the execution of all map tasks, and the green and blue lines indicate the shuffling phase and the reduce phase in reduce tasks, respectively. For Fair-1, the shuffling phase lasts 293 seconds after the map phase is finished and this time is decreased to 36 seconds in Fair-0.05. But Fair-0.05 spends an additional 75 seconds in the map phase compared to Fair-1. Our solution OMO achieves 34.9% shorter execution time in the map phase than Fair-1 and takes only 43 seconds in the shuffling phase after the map phase is finished. For FRESH, since all the containers are assigned to the map tasks before the map phase is finished, the time cost in the map phase is 17.8% shorter than Fair-1. Our solution OMO yields 20.8% shorter execution time in the map phase than FRESH. Such improvement is achieved by Batch Finish.

2) PERFORMANCE

Our solutions are compared with other schedulers in a Hadoop Cluster with 20 slave nodes. First, we demonstrate the makespan of Lazy Start and Batch Finish separately. Afterward, we demonstrate the performance with the combination of both techniques.

Both simple and mixed workloads are taken into consideration in each set of tests. For each experiment of simple

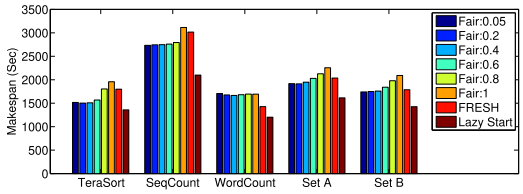


FIGURE 10. Execution time under FAIR SCHEDULER, FRESH and Lazy Start (with 20 slave nodes).

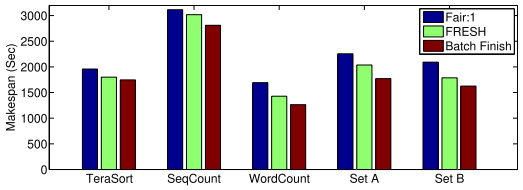


FIGURE 11. Execution time under FAIR SCHEDULER, *FRESH* and *Batch Finish* (with 20 slave nodes).

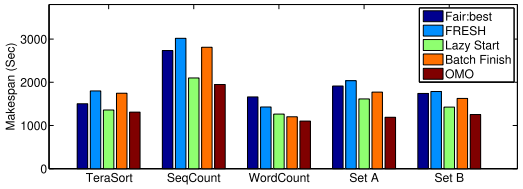


FIGURE 12. Execution time under FAIR SCHEDULER, *FRESH*, *Lazy Start*, *Batch Finish* and *OMO* (with 20 slave nodes).

workloads, 8 jobs with uniform benchmarks are created with matching identical input data. Each data set is 20 GB making the total data processed in each test 160 GB. Each job has 80 map tasks along with 2 reduce tasks. All jobs are submitted sequentially to the Hadoop system with an interval of 2 seconds.

In addition to simple workloads, to further authenticate the efficiency of our solution, we assess the system performance with varying workloads which includes various benchmarks. 8 Job sets (Set A to H) are then created with mixed jobs which are introduced in Table 5. Specifically, Set A is combined with a mixture of job types consisting of both heavy and light-shuffling ones. A recent suggestion from Cloudera has suggested that 34% of jobs have at least the same amount of output data as their inputs [35] Therefore, the 12 jobs consist of 8 light-shuffling and 4 heavy-shuffling jobs. Each benchmark has two jobs, one with 20 GB input data and the other with 10 GB. Set B is a combined job set consisting of only heaving-shuffling benchmarks: Terasort and Sequence Count. Every benchmark consists of 8 jobs, 4 with 20 GB of input data and the remaining with 10 GB. Sets C through H is designed for scalability experiments of OMO.

TABLE 5. 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
D	All benchmarks	12	30 GB	120, 3
E	All benchmarks	12	40 GB	160, 4
F	All benchmarks	18	20 GB	80, 2
G	All benchmarks	24	20 GB	80, 2
H	All benchmarks	12	20 GB	80, 2
		12	10 GB	40, 1

a: MAKESPAN PERFORMANCE OF LAZY START

First, the Batch Finish algorithm in the Execution Predictor (**EP**) Module is disabled. From this, we can indicate the performance of Lazy Start. The makespan performance of FRESH, Fair scheduler and Lazy Start with simple and mixed benchmarks are illustrated in Fig. 10. Considering the page limit for the simple workloads, we present the evaluation results with 3 benchmarks.

In testing cases of Fair scheduler, it had the best makespan performance with simple workloads and heavy shuffling (i.e. Terasort and Sequence Count) when the slowstart was set at 0.05 or 0.2. Lazy Start improves the makespan time by 11.6% and 15.9% compared to Fair schedule and 24.5% and 23.8% when compared to FRESH. Fair scheduler however has similar performance when testing with different values of slowstart when using light-shuffling benchmarks such as word count. FRESH also performs well since the shuffling time is short. The mean makespan in Lazy Start is 27.8% less than Fair scheduler and 15.8% less than FRESH.

When considering mixed workload testing, Fair-1 proved to be the least efficient with various sets of jobs within Fair scheduler. In job Set A, the Lazy Start performance is improved by 18.1% when compared to Fair schedulerr and by 20.2% when compared to FRESH. In job Set B Lazy Start decreases 15.6% and 20.7% of makespan when compared to the best scenario in Fair scheduler and FRESH

b: MAKESPAN PERFORMANCE OF BATCH FINISH

Furthermore, we disable Lazy Start in OMO to display the evaluation results of Batch Finish. The makespan performance of Batch Finish, FRESH and Fair-1 in simple and mixed workloads are illustrated in Fig. 11. In these cases, the slowstart value is set to 1 for all three schedulers. In terms of performance, FRESH demonstrates more desirable results over Fair-1 due to the dynamic container configuration. In simple and mixed workload testing, on Average, the makespan of FRESH is 7.26% AND 12.3% shorter than Fair-1. And compared to FRESH, Batch Finish cuts the makespan by 7.1% and 11%

c: PERFORMANCE OF OMO

Lastly, we demonstrate the results of OMO a combination of the techniques listed above. First using simple and mixed workloads, we illustrate the makespan performance compared to other schedulers. We then break down the execution in both map and shuffling stage of each trial and then illustrate the completion time of each stage.

Fig. 12 illustrates the results collected from testing of two sets of mixed workloads and three simple workloads.

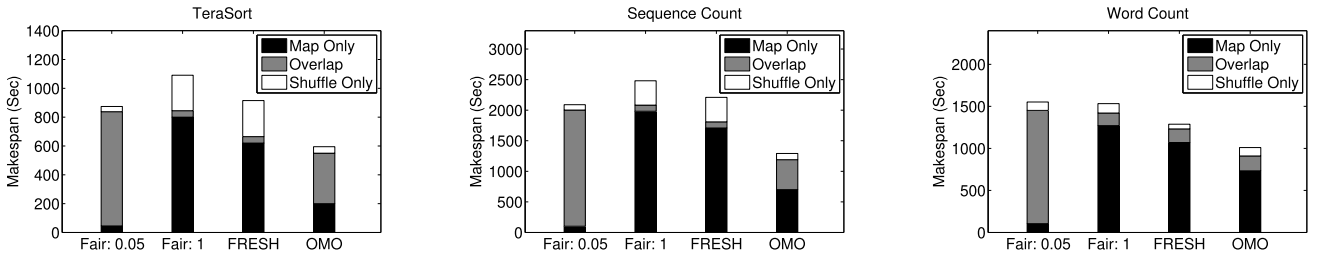


FIGURE 13. Execution time in map + shuffling phase of simple workloads.

Fair:best represents the best makespan performance in Fair scheduler with varying values of slowstart. Lazy Start decreases the heavy-shuffling benchmarks even more while Batch Finishh performs better when working in light-shuffling benchmarks. OMO benefits from both techniques and achieves the best performance when compared to either Lazy Start or Batch Finish. On average, OMO reduced the makespan by 26% and 29.3% when compared to Fair:best and FRESH.

Fig. 13 shows the details of the breakdown execution time in three steps of each experiment with simple workloads: Map Only represents the time span that only map tasks are executed, but no reduce tasks, Overlap represents the time span that both the map and the shuffling phase are running concurrently. Shuffle Only represents the time span that the shuffling phase continues after the map phase has finished. By increasing map task containers in the cluster, FRESH reduces 15.51 of the time span in the map phase compared to Fair scheduler. However, it takes more time in Shuffle Only than Fair:0.05. Overall, OMO decreases the execution time in Shuffle Only significantly with the help of Lazy Start and still optimize the time span of the map phase by the technique of Batch Finish.

3) COMPARISON WITH HADOOP YARN

Furthermore, as shown in Table 6 is the comparison with Hadoop YARN. Since not all benchmarks are available for Hadoop YARN distribution, only Terasort is used in the experiments. In each test, 8 Terasort jobs with 20 GB of input data is used. The CPU demand is set to 2 cores, this way at any one time only 4 tasks are running concurrently at each node. In Hadoop YARN, a novel mechanism is developed for the allocation of reduce tasks. Generally, for each job in YARN, when reduce tasks can be assigned for execution is based on the progress of the map stage and a memory threshold for reduce tasks in the cluster (*maxReduceRampupLimit*). For the experiments, the default configurations are used. We set, the memory requirement of map/reduce tasks of each job with different values in the experiment, including 2, 4, 6, and 8 GB, represented by YARN:2, YARN:4, YARN:6 and YARN:8 in Table 6. Within the Hadoop YARN cluster, the makespan of YARN:2 displays approximately 6.6% more than the ones with different memory demands. OMO reduces the makespan by 17.3% when compared to YARN2: and 11.6% when compared to the others. It should be noted that Hadoop YARN takes a fine-grained resource management which indicates inherited advantages over the traditional Hadoop system which OMO is built upon. However, it is shown that OMO still outclasses the Hadoop YARN system. Our OMO design can easily be extended and ported to the Hadoop YARN system which is a part of our future work.

TABLE 6. 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

4) SCALABILITY

Finally, we show the scalability of OMO with the experiments of different input data sizes job numbers and cluster sizes.

First, we test the input data scalability. We run the experiments of 12 mixed jobs with the input data size: 20 GB (Set C), 30 GB (Set D), and 40 GB (Set E). Fig. 14 (a) shows the evaluation results. The execution times with 30 GB and 40 GB inputs are 1.6 and 2.1 times of the one with 20 GB inputs. The growth of the execution time in OMO is proportional to the rise of the input data size.

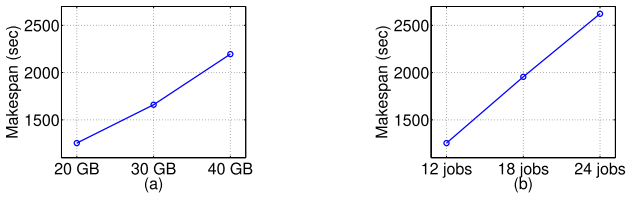


FIGURE 14. Execution time under *OMO* with: (a) different sizes of the input data and (b) different number of jobs.

Then, we test the number of jobs scalability, we run the experiments with the same set of mixed benchmarks. The input data of each job is 20 GB. Set C has 12 jobs, Set F has 18 jobs, and Set G has 24 jobs. Fig. 14 (b) shows the experiments results. The execution time in OMO grows linearly according to the number of jobs.

Finally, we evaluate the scalability of OMO by executing Set H on a large cluster with 40 slave nodes. The experiment results are shown in Table 7. We can observe a consistent performance gain from OMO $\gg$. The small cluster of 20 slave nodes with Set A. OMO reduces the makespan by 37% compared to Fair scheduler and FRESH, which is consistent with the tests using Set A on the 20-node cluster.

TABLE 7. Makespan of set H with 40 slave nodes.

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

Above all, OMO accomplishes an exceptional and stable makespan performance with both simple and mixed workloads of various sets of jobs.

VI. CONCLUSION

This paper studies the scheduling problem in a big data computing system with multiple internal stages, especially in a Hadoop cluster serving a batch of MapReduce jobs. Our goal is to reduce the overall makespan of multiple applications by appropriate resource allocation. A new scheme OMO is developed to optimize the overlap between the map and reduce phases. There are two new techniques introduced in OMO: lazy start of reduce tasks and batch finish of map tasks. Compared to previous work, our solution takes more dynamic factors and predicts resource availability into consideration when assigning the containers to jobs. We implemented OMO on the Hadoop cluster and evaluated all the techniques in multiple clusters of Cloud Lab by running representative MapReduce benchmarks with various workloads and settings. The evaluation results show a significant improvement in the makespan compared to both conventional Hadoop and Hadoop YARN systems, especially for heavy-shuffling jobs.

REFERENCES

[1] *Apache Hadoop*. Accessed: Apr. 19, 2021. [Online]. Available: http://hadoop.apache.org/
[2] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. NSDI USENIX*, 2011, p. 22.
[3] *Apache Spark*. Accessed: Apr. 19, 2021. [Online]. Available: https://databricks.com/spark/
[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[5] *Fair Scheduler*. Accessed: Apr. 19, 2021. [Online]. Available: https://hadoop.apache.org/docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/FairScheduler.html
[6] *Capacity Scheduler*. Accessed: Apr. 19, 2021. [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html
[7] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX Association, 2011, pp. 323–336. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972490
[8] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic resource inference and allocation for mapreduce environments," in *Proc. 8th ACM Int. Conf. Autonomic Comput. (ICAC)*, 2011, pp. 235–244.
[9] K. Kc and K. Anyanwu, "Scheduling Hadoop jobs to meet deadlines," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, Nov. 2010, pp. 388–392.
[10] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. 24th ACM Symp. Operating Syst. Princ. (SOSP)*, New York, NY, USA: ACM, 2013, pp. 69–84. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522716
[11] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware adaptive scheduling for mapreduce clusters," in *Proc. Middleware*, 2011.
[12] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "FRESH: Fair and efficient slot configuration and scheduling for Hadoop clusters," in *Proc. CLOUD*, Jun. 2014, pp. 761–768.
[13] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "OMO: Optimize MapReduce overlap with a good start (reduce) and a good finish (map)," in *Proc. IPCCC*, Dec. 2015, pp. 1–8.
[14] J. Wang, "Building efficient large-scale big data processing platforms," Ph.D. dissertation, Dept. Comput. Sci., Univ. Massachusetts Boston, Boston, MA, USA, 2017, vol. 348. [Online]. Available: https://scholarworks.umb.edu/doctoral_dissertations/348
[15] Y. Yao, J. Wang, B. Sheng, J. Lin, and N. Mi, "HaSTE: Hadoop YARN scheduling based on task-dependency and resource-demand," in *Proc. IEEE 7th Int. Conf. Cloud Comput.*, Washington, DC, USA: IEEE Computer Society, Jun. 2014, pp. 184–191, doi: 10.1109/CLOUD.2014.34.
[16] J.-C. Lin, I. C. Yu, E. B. Johnsen, and M.-C. Lee, "ABS-YARN: A formal framework for modeling Hadoop YARN clusters," in *Fundamental Approaches to Software Engineering*, P. Stevens and A. Wąsowski, Eds. Berlin, Germany: Springer, 2016, pp. 49–65.
[17] C. Hu, J. Zhu, R. Yang, H. Peng, T. Wo, S. Xue, X. Yu, J. Xu, and R. Ranjan, "TOPOSCH: Latency-aware scheduling based on critical path analysis on shared YARN clusters," in *Proc. IEEE 13th Int. Conf. Cloud Comput. (CLOUD)*, Oct. 2020, pp. 619–627.
[18] X. Zhang, Z. Zhong, S. Feng, B. Tu, and J. Fan, "Improving data locality of MapReduce by scheduling in homogeneous computing environments," in *Proc. IEEE 9th Int. Symp. Parallel Distrib. Process. Appl.*, May 2011, pp. 120–126.
[19] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in MapReduce," in *Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2012, pp. 419–426.
[20] J. Jin, J. Luo, A. Song, F. Dong, and R. Xiong, "BAR: An efficient data locality driven task scheduling algorithm for cloud computing," in *Proc. 11th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, IEEE Computer Society, May 2011, pp. 295–304.
[21] X. Zhang, Y. Feng, S. Feng, J. Fan, and Z. Ming, "An effective data locality aware task scheduling method for MapReduce framework in heterogeneous environments," in *Proc. Int. Conf. Cloud Service Comput.*, Dec. 2011, pp. 235–242.
[22] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum (IPDPSW)*, Apr. 2010, pp. 1–9.
[23] P. Li, L. Ju, Z. Jia, and Z. Sun, "SLA-aware energy-efficient scheduling scheme for Hadoop YARN," in *Proc. IEEE IEEE 17th Int. Conf. High Perform. Comput. Commun. 7th Int. Symp. Cyberspace Saf. Secur., IEEE 12th Int. Conf. Embedded Softw. Syst.*, Aug. 2015, pp. 623–628.
[24] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, Feb. 2015, doi: 10.1145/2740070.2626334.
[25] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proc. 8th USENIX Conf. Operating Syst. Design Implement. (OSDI)*. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855744
[26] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 379–392, Sep. 2015, doi: 10.1145/2829988.2787481.
[27] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "Grass: Trimming stragglers in approximation analytics," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Seattle, WA, USA: USENIX Association, 2014, pp. 289–302. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/ananthanarayanan
[28] J. Wang, T. Wang, Z. Yang, N. Mi, and B. Sheng, "ESplash: Efficient speculation in large scale heterogeneous computing systems," in *Proc. IEEE 35th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2016, pp. 1–8.
[29] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. EuroSys*, 2007, pp. 59–72.
[30] O'Reilly Media, *Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale*, 4th ed. Sebastopol, CA, USA: O'Reilly Media, 2015.
[31] *Apach Hadoop YARN*. Accessed: Apr. 19, 2021. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[32] Y. Yao, J. Wang, B. Sheng, and N. Mi, "Using a tunable knob for reducing makespan of mapreduce jobs in a Hadoop cluster," in *Proc. CLOUD*, Jun. 2013, pp. 1–8.
[33] *NSF Cloudlab*. Accessed: Apr. 19, 2021. [Online]. Available: https://www.cloudlab.us/
[34] *Purdue Mapreduce Benchmarks Suite*. Accessed: Apr. 19, 2021. [Online]. Available: https://engineering.purdue.edu/ puma/pumabenchmarks.htm
[35] S. A. Y. Chen and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," in *Proc. VLDB*, 2012, pp. 1802–1813.

ALLEN YANG received the degree in computer science from Montclair State University, in May 2021. His research interests include deep learning, machine learning, data science, big data, and cloud computing.

JIAYIN WANG received the bachelor's degree in electrical engineering from Xidian University, China, in 2005, and the Ph.D. degree from the University of Massachusetts Boston, in 2017. She is currently an Assistant Professor with the Computer Science Department, Montclair State University. Her research interests include big data analytics, cloud computing, and wireless networks.

YING MAO received the B.S. degree from the Commanding Communication Academy (currently, the National University of Defense Technology, Wuhan, China), the M.S. degree from University at Buffalo, and the Ph.D. degree from the University of Massachusetts Boston. He is currently an Assistant Professor with the Department of Computer and Information Science, Fordham University. His research interests include cloud computing, virtualization, resource management, and data-intensive platforms.

YI YAO received the Ph.D. degree in computer science from the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA. He is currently a Software Engineer with Google. His current research interests include resource management, scheduling, and cloud computing.

NINGFANG MI (Member, IEEE) received the B.S. degree in computer science from Nanjing University, China, in 2000, the M.S. degree in computer science from the University of Texas at Dallas, TX, USA, in 2004, and the Ph.D. degree in computer science from the College of William and Mary, VA, USA, in 2009. She is currently an Associate Professor with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA. Her current research interests include performance evaluation, capacity planning, resource management, simulation, data center, and cloud computing.

BO SHENG received the Ph.D. degree in computer science from the College of William and Mary, in 2010. He is currently an Associate Professor with the Department of Computer Science, University of Massachusetts Boston. His research interests include mobile computing, wireless networks, security, and cloud computing.