BUILDING EFFICIENT LARGE-SCALE BIG DATA PROCESSING PLATFORMS

A Dissertation Presented

by

JIAYIN WANG

Submitted to the Office of Graduate Studies,
University of Massachusetts Boston,
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2017

Computer Science Program

BUILDING EFFICIENT LARGE-SCALE BIG DATA PROCESSING PLATFORMS

A Dissertation Presented

by

JIAYIN WANG

Approved as to style and content by:

Bo Sheng, Associate Professor
Chairperson of Committee

Wei Ding, Associate Professor
Member

Dan Simovici, Professor
Member

Honggang Zhang, Associate Professor
Member

Dan Simovici, Program Director
Computer Science Program

Peter Fejer, Chairperson
Computer Science Department

ABSTRACT

BUILDING EFFICIENT LARGE-SCALE BIG DATA PROCESSING PLATFORMS

MAY 2017

JIAYIN WANG

B.S., XIDIAN UNIVERSITY, XI'AN, CHINA

M.S., UNIVERSITY OF MASSACHUSETTS BOSTON

Ph.D., UNIVERSITY OF MASSACHUSETTS BOSTON

Directed by: Professor Bo Sheng, Associate Professor

In the era of big data, many cluster platforms and resource management schemes are created to satisfy the increasing demands on processing a large volume of data. A general setting of big data processing jobs consists of multiple stages, and each stage represents a generally defined data operation such as filtering and sorting. To parallelize the job execution in a cluster, each stage includes a number of identical tasks that can be concurrently launched at multiple servers. Practical clusters often involve hundreds or thousands of servers processing a large batch of jobs. Resource management, that manages cluster resource allocation and job execution, is extremely critical for the system performance.

Generally speaking, there are three main challenges in resource management of the new big data processing systems. First, while there are various pending tasks from different jobs and stages, it is difficult to determine which ones deserve the priority

to obtain the resources for execution, considering the tasks' different characteristics such as resource demand and execution time. Second, there exists dependency among the tasks that can be concurrently running. For any two consecutive stages of a job, the output data of the former stage is the input data of the later one. The resource management has to comply with such dependency. The third challenge is the inconsistent performance of the cluster nodes. In practice, run-time performance of every server is varying. The resource management needs to dynamically adjust the resource allocation according to the performance change of each server.

The resource management in the existing platforms and prior work often rely on fixed user-specific configurations, and assumes consistent performance in each node. The performance, however, is not satisfactory under various workloads. This dissertation aims to explore new approaches to improving the efficiency of large-scale big data processing platforms. In particular, the run-time dynamic factors are carefully considered when the system allocates the resources. New algorithms are developed to collect run-time data and predict the characteristics of jobs and the cluster. We further develop resource management schemes that dynamically tune the resource allocation for each stage of every running job in the cluster. New findings and techniques in this dissertation will certainly provide valuable and inspiring insights to other similar problems in the research community.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to the following people. I would not have been able to complete my dissertation without the support of them.

Firstly, I would like to thank my advisor, Professor Bo Sheng, for his continuous guidance and encouragement throughout my Ph.D. life with his patience, motivation, enthusiasm, and immense knowledge. I thank him for introducing me to the wonders of scientific research. I could not have imagined having a better advisor and mentor for my Ph.D. study. Meanwhile, I would like to offer my special thanks my committee, Professor Dan Simovici, Professor Wei Ding, and Professor Honggang Zhang, for their great support and insightful feedback and comments.

In addition, I wish to acknowledge the help provided by my research collaborators: Professor Ningfang Mi, Doctor Yi Yao, Doctor Ying Mao and Zhengyu Yang. My thanks also go to my labmates Teng Wang and Son Nam Nguyen for working together, and for all the fun we have had in the last five years. Many thanks to my friends in the Computer Science department Ting Zhang, Dong Luo, Dawei Wang, Yahui Di, and Akram Bayat, for the friendship and support.

Finally, I would like to thank everyone in my family. I am grateful to my parents Genjian Wang and Xiaopei Guo, and also my cousin Doctor Lingyin Li (my eternal role model and cheerleader) for always supporting me and loving me. Especially, I wish to thank my husband Shaohua Jia and my son Chenrui Jia for standing by me through the tough times and bringing happiness into my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The past decades have witnessed a major change in data processing platforms, as the rapid growth of big data requires more and more applications to scale out to large clusters. From the statistics of IDC reports [1], there are 4.4 zettabytes (4.4 billion terabytes) of data exist in the digital universe and this number will be increased to 44 zettabytes by 2020. Besides structured data in database, unstructured data sets are generated in both academia and industry by various of modern technologies such as gene sequencers, wearable sensors, social networks (e.g., Facebook and Twitter), radio frequency ID (RFID), Internet of things (IoTs), and smartphones. Meanwhile, big data analytics is well used in almost everywhere of our daily life, such as healthcare, business, finance, traffic control, manufacturing, and retail. Powerful and efficient computer systems are required to process big data in a timely fashion. However, the traditional database system deployed in a single server is inadequate to deal with big data because of its increasing of volume, variety, velocity, and veracity. Therefore, many cluster-based scalable platforms have been developed to serve the growing demands for processing big data in parallel. Fig. 1.1 shows the family of approaches and mechanisms of emerging large-scale data processing systems. In a cluster, the input and output data are stored in a distributed file system such as HDFS (Hadoop Distributed File System). Upon the storage layer are deployed by different large-scale data computing systems such as MapReduce/Hadoop [2], Hadoop YARN [3], Mesos [4], Tez [5], and Spark [6]. A series of eco-systems are built upon

them to process multiple types of data and applications such as Hive [7], Pig [8], Shark [9], Storm [10], and Mahout [11].

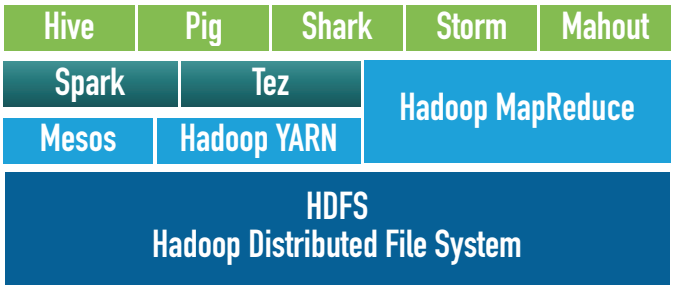

Figure 1.1: Typical deployment of large-scale big data computing systems

A general setting of big data jobs consists of a sequence of processing stages, and each stage represents a generally defined data operation such as filtering, merging, sorting, and mapping. To parallelize the job execution in a large-scale cluster, each stage includes a number of identical tasks that can be concurrently launched at multiple servers. This general setting of multi-stage data processing includes a wide range of data analysis products in practice. For example, MapReduce/Hadoop represents a typical two-stage process. Other representative applications with multi-stage data processing include chained MapReduce jobs for SQL-on-Hadoop queries, iterative machine learning algorithms (e.g., pagerank, logistic regression, k-mean, and others in Mahout library), and scientific computation (e.g., data assimilation in GFS weather forecasting and hydrology, and partial differential equation based simulation).

In a large-scale cluster, resource management is extremely critical for the performance. It has been frequently reported that the resource utilization in big data processing systems is unexpectedly low. For example, a production cluster with thousands of servers at Twitter managed by Mesos has reported its aggregated CPU utilization lower than 20% even though 80% resource capacities of the cluster are reserved.

This dissertation aims to build efficient large-scale big data processing systems by exploring new resource management schemes to dynamically tune the resource allocation based on the characteristics of the data processing systems and job properties.

1.1 Research Challenges

In this section, we discuss the common characteristics of the current big data computing systems, and the main research challenges in resource management based on these characteristics. Our work is motivated by the following challenges.

1. *Various workloads.* Big data processing systems usually serve numerous kinds of applications submitted by various users concurrently. Every job consists of multiple stages and one unique operation is provided in each stage. Tasks from different stages of different jobs require dissimilar resource demands and yield varying execution time. For example, a typical MapReduce job contains two stages: map and reduce. The map stage focuses on disk I/O processing and the reduce stage is responsible for data analysis. Therefore, tasks in the map stage usually rely more on the disk I/O and the ones in the reduce stage require more resources on memory and CPU. Even in the reduce stage, different operations rely on different amounts of resources. For instance, in the text processing, sorting usually requests more memory resource than word counting. With different demands of resources, it is challenging to assign appropriate resources to numerous types of applications with multiple stages in order to give consideration to the execution times of all jobs and also fair resource sharing for each job.

2. *Dependency of stages.* Dependency usually exists between stages of every application in big data processing systems. For any two consecutive stages of a job, the output data of the former stage is the input data of the later one. On the one hand, the later stage cannot finish before its former stage. The resource management in the cluster needs to comply with such dependency. On the other hand, the later

one can start earlier before the completion of the former one. The first component in each stage (except the first stage in a job) is called shuffling. It is responsible for transferring the output data generated from the former stage to itself. Starting the shuffling phase earlier can help improve the performance by overlapping the shuffling phase and the finish of the previous stage. We find that this unique period plays an important role on the resource management. Appropriately determining the finish time of one stage and the start time of the next stage can avoid idle resources or accumulated data waiting for processing. However, different stages of various jobs generate varying sizes of data and the data transferring rate is changing all the time according to the system bandwidth and interference by the shuffling operations from other jobs. Therefore, it is challenging for the resource management to determine the best timing to start the later stages for different jobs.

3. *Inconsistent performance of the cluster nodes.* During the execution of any cluster in practice, the run-time performance of every node is varying. Many factors can impact the system performance of a server in the cluster, such as the number of processes the server is running at the same time, the memory occupation percentage, and the competition of reading/writing operations to the disk. It requires the scheduler to adjust the resource allocation in real time according to the performance change of each server.

4. *Node failures in the cluster.* In a large-scale cluster, node failures and stragglers (slow servers) are normality in practical. Fault tolerance is usually executed in the computing systems to handle this kind of issues. Speculation mechanism is a common solution to mitigate the impact of such failures. Basically, when a straggler is detected, a copy of its tasks (speculative tasks) will be created and assigned to another server to finish faster so that a job would not be as slow as the misbehaving tasks. Three requests should be taken into consideration for an effective and efficient speculative mechanism. First, it should be able to accurately identify stragglers

from normal slow nodes when every node in the cluster has different performance (i.e., a speculation mechanism should work well in a heterogeneous cluster). Second, when a node becomes abnormally slow, the speculation mechanism should detect this straggler rapidly. Finally, once a straggler is identified, its duplicated tasks should be assigned to appropriate nodes with good performance, since assigning duplicated tasks back to the stragglers or even slower nodes cannot improve the system performance or reduce the job execution time. However, we find that the existing speculative mechanisms cannot satisfy these three demands. In chapter 5, we will introduce our efficient speculative mechanism.

Above all, these characteristics provides both challenges and opportunities for the resource management schemes for the large-scale big data processing system. However, the existing ones have not thoroughly addressed these issues. In the dissertation, we will introduce our new scheduling and resource management approaches to improving the efficiency, e.g., reducing the execution times of a batch of jobs (i.e., makespan) and increasing the system resource utilization, in the large-scale cluster computing systems.

1.2 Dissertation Contributions

The main contributions of this dissertation on building efficient large-scale big data processing platforms contain the following components.

1. We develop a new approach, named FRESH [12], to achieve fair and efficient resource allocation and scheduling for the computing clusters with various workloads. The main intuition is to allocate resources for each stage first, and then further split the resource across multiple jobs that have active tasks in that stage. The targets of our approach include minimizing the makespan as the major objective and meanwhile improving the fairness without degrading the makespan. As shown in Fig. 1.2, FRESH is a workload-based resource alloca-

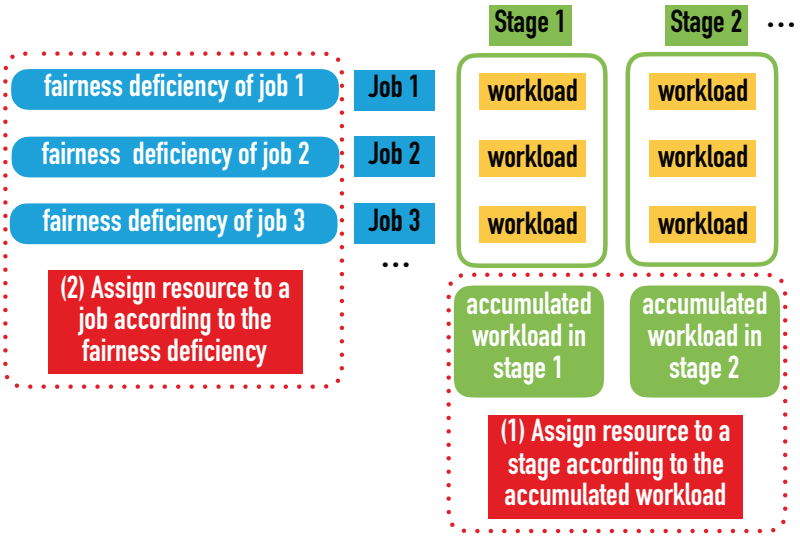

Figure 1.2: General mechanism of FRESH

tion scheme which contains two major techniques. First, a real-time monitor in FRESH records the workload of every stage in each running job. Every time there are idle resources in the cluster, based on the accumulated workload of each stage, the idle resources will be assigned to an appropriate stage, represented as *stage i*. Second, an overall fairness mechanism detects the fairness deficiency of every job with the active *stage i*. Then the idle resource will be allocated to the tasks in *stage i* of the job with the least fairness.

2. We develop a new strategy, OMO [13] to improve the makespan of batch jobs by optimizing the overlap between two active consecutive stages. The basic approach is to let multiple jobs fairly share the system resource, and then focus on the resource allocation for consecutive stages in each job. OMO considers dynamic factors at the run time and allocates the resources based on the dependency of stages in every job. Fig. 1.3 shows the general mechanism of OMO. For any two active consecutive stages of a job, a novel prediction module in OMO estimates the resource availability in the future and further predicts the remaining execution time of the former stage and the shuffling time of the later

stage. Based on this estimation, with idle resources available in the cluster, OMO checks whether there exists a job that should start its later stage so that there is sufficient time for the later stage to shuffle data from the previous one. If such job exists, the idle resources will be assigned to the later stage. Otherwise, a job with most fairness deficiency will be chosen to execute the tasks in its former stage.

3. We present ESPLASH [14], an efficient resource allocation scheme to mitigate the impact of node failures or stragglers on the system performances. As a new speculative mechanism, ESPLASH contains the following major components (shown in Fig. 1.4). First, to identify stragglers from nodes with various performance, we cluster all the nodes into different levels according to their computing performance. In this case, nodes in the same cluster perform similar computing abilities. Second, ESPLASH identifies stragglers by monitoring the task's execution time and progress rate. Within the statistics of the task progress rate, a straggler can be quickly detected in the beginning execution period of a task. Finally, ESPLASH monitors the performance of every node in the cluster and assign duplicated tasks to the most appropriate nodes.

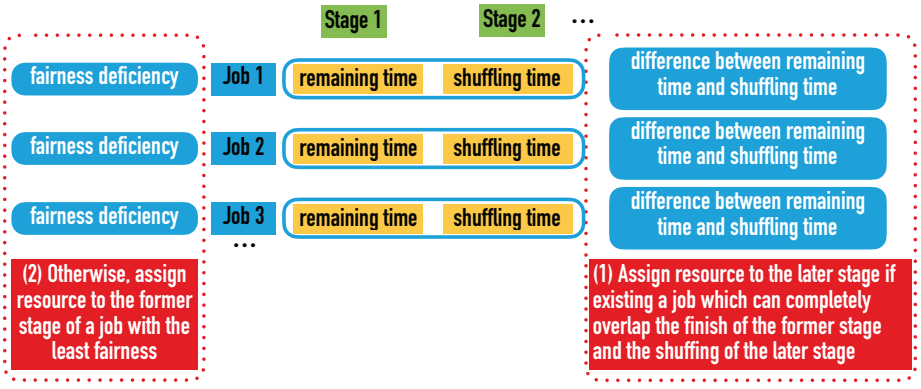

Figure 1.3: General mechanism of OMO

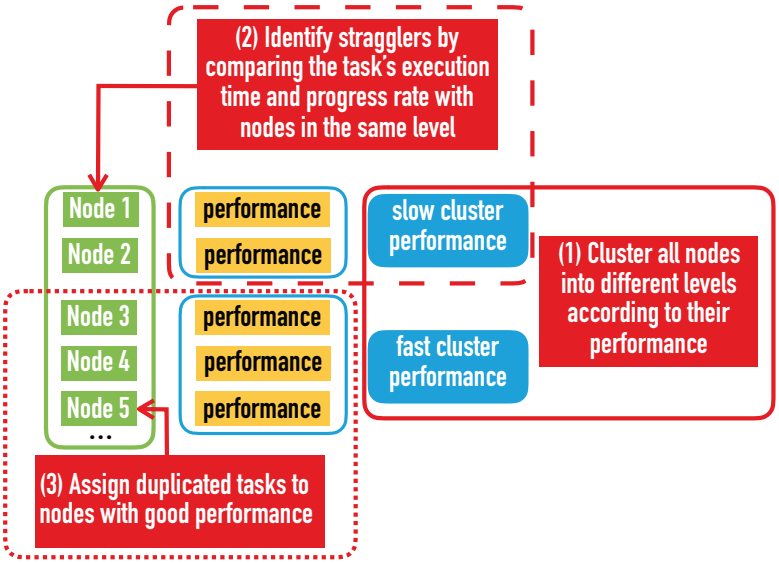

Figure 1.4: General mechanism of eSplash

Since MapReduce/Hadoop [2] and its next generation Hadoop YARN [3] are well used in both academia and industry, we consider them as representatives computing platforms. Our implementations and experiments are conducted on these two platforms. In FRESH and OMO, 10 new components are developed with about 1400 lines of code in each work. 11 modules of Hadoop YARN v2.7.1 are modified in eSplash with 1500 lines of code. We have evaluated the efficiency and effectiveness of these new schemes on large clusters in cloud computing platforms, including Amazon EC2 and NSF CloudLab. The results show significant performance improvements. FRESH and OMO can decrease averagely 16% to 38% of makespan compared to default schedulers in Hadoop. With a node failed in the cluster, on average, the increased makespan under eSplash is 67% less than the ones under default schemes in Hadoop YARN.

1.3 Dissertation Organization

The dissertation is organized as follows. Chapter 2 introduces the overview of the representative computing platforms, including MapReduce programming paradigm,

the architecture of two cluster computing systems: Hadoop and Hadoop YARN, and the default scheduling policies in these platforms. In Chapter 3, we present our first approach FRESH that allocate resources for each stage based on the estimation of the workload. It aims to reduce the makespan of jobs and also consider the fairness of each job. Chapter 4 introduces another solution OMO that is focused on the dependency of consecutive stages in a job. Chapter 5 presents our new speculation scheme eSplash which improves the system performance by efficiently allocate resources for redundant task execution. Finally, we summarize our work and conclude the dissertation in Chapter 6.

CHAPTER 2

BACKGROUND

In this chapter, we introduce some basic concepts and knowledge of the representative computing platforms. Four main aspects will be illustrated. First, we introduce the MapReduce programming model, especially the work flow of the two stages in a MapReduce application. Second, the structures of both Hadoop and Hadoop YARN platforms are be elaborated. Our work is implemented as plug-in components in these platforms. Finally, tree typical scheduling policies, FIFO, Fair, and Capacity are introduced in this chapter. They are all default schedulers in most big data computing systems and can be set in the system configuration file. We consider them as the baseline to compare with in our performance evaluation.

2.1 MapReduce

MapReduce is a programming model introduced by Google for processing large data sets on clusters of computers. Fig. 2.1 shows the parallel data processing scheme of MapReduce. A typical MapReduce job contains two stages: map and reduce. Each stage consists of multiple identical tasks. A map task (mapper) processes one block of the raw data which is stored in a shared distributed file system and generates intermediate data in a form of <key, value>. There are three components in the reduce stage: shuffling, copy and reducer. Shuffling is responsible for copying intermediate data from the map phase to the reduce phase. Once shuffling finishes all the intermediate data transfer, copy component collects data and the reducer component processes the intermediate data and produces the final results.

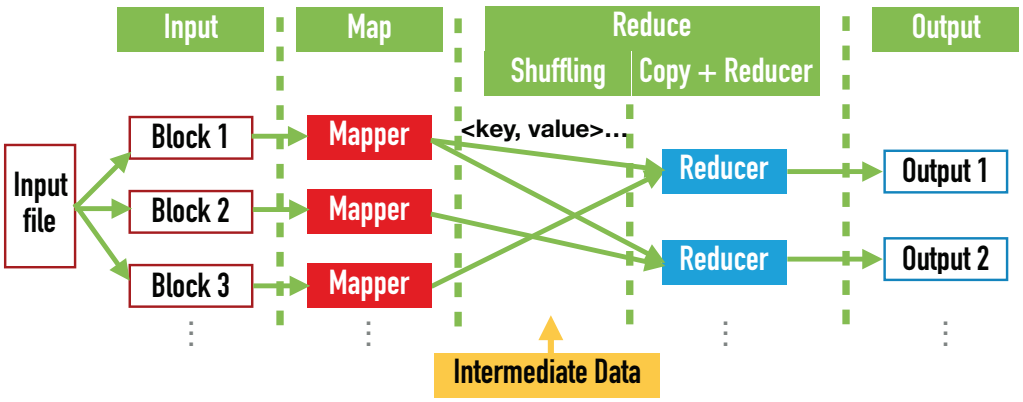

Figure 2.1: MapReduce data processing scheme.

2.2 Hadoop MapReduce

The Apache Hadoop MapReduce is an open source implementation of MapReduce. The structure of Hadoop is shown in Fig. 2.2. In a Hadoop cluster, there is one centralized master node and several distributed slave nodes. Two major components are contained in Hadoop: Hadoop distributed file system (HDFS) and MapReduce job execution system. The local disks from slave nodes are combined together as the distributed file system. All input and output data are split into multiple data blocks and saved separately in it. Each data block can have multiple redundant copies for fault tolerance and data locality. A centralized NameNode deployed in the master node is responsible for the management of HDFS. It stores metadata, file names and block locations. In each slave node, a DataNode module manages the stored data.

In the job execution, users submit applications to the master node. All jobs are scheduled and managed by the JobTracker in the master node. One job consists of multiple map/reduce tasks. The scheduler in the JobTracker is in charge of assigning tasks to the TaskTracker of the appropriate slave nodes, and each TaskTracker is responsible for executing tasks. In each slave node, resources are represented as task slots. TaskTracker manages a number of map/reduce slots that can be used to run either map tasks or reduce tasks. One task slot can execute one task at a time.

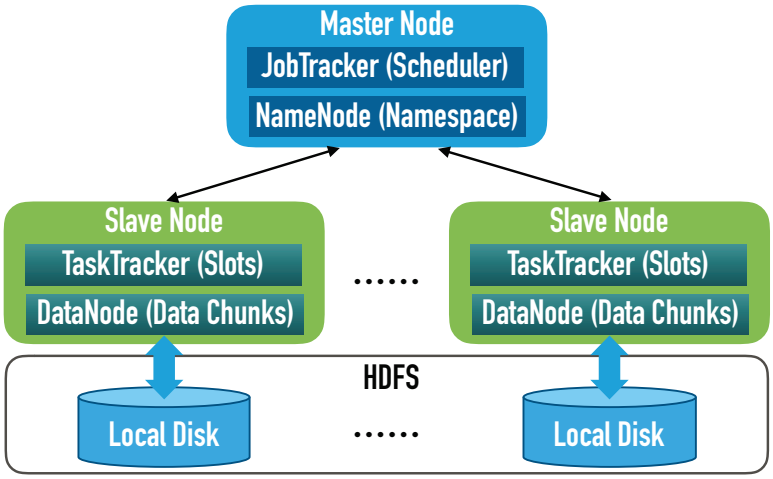

Figure 2.2: Structure of Apache Hadoop.

Map (resp. reduce) tasks can only run on map (resp. reduce) slots. The number of map/reduce slots in each TaskTracker can be set in the system configuration file. Once the Hadoop cluster is set up, such configuration cannot be changed.

2.3 Hadoop YARN

Hadoop YARN is the next generation of Hadoop. Similar to Hadoop MapReduce, in Hadoop YARN, there is a centralized master node running ResourceManager and several distributed slave nodes deployed by NodeManager on each slave. However, Hadoop YARN has two main differences from the first generation of Hadoop. First, YARN supports fine-grained resource management. Each slave node specifies resource capacity in the format of CPU cores and memory. Each task specifies a resource demand while submitted and ResourceManager responds the resource demand by granting a resource container in a slave node. The second difference is that the resource management and job coordination are separated in YARN. A new component ApplicationMaster takes care of the job coordination. And the ResourceManager is just responsible for scheduling. In this case, YARN can support different frameworks on the same cluster, such as MapReduce and Spark.

Shown in Fig. 2.3 is the execution of a job in Hadoop YARN. When users submit a job to the ResourceManager, an ApplicationMaster will be launched in a NodeManager. Then the ApplicationMaster requests resource for the tasks of the job to the ResourceManager. ResourceManager responses it multiple tokens to the containers. Each token describes the resource a task can get. Within the tokens, ApplicationMaster will launch containers in the NodeManager and assign tasks for execution.

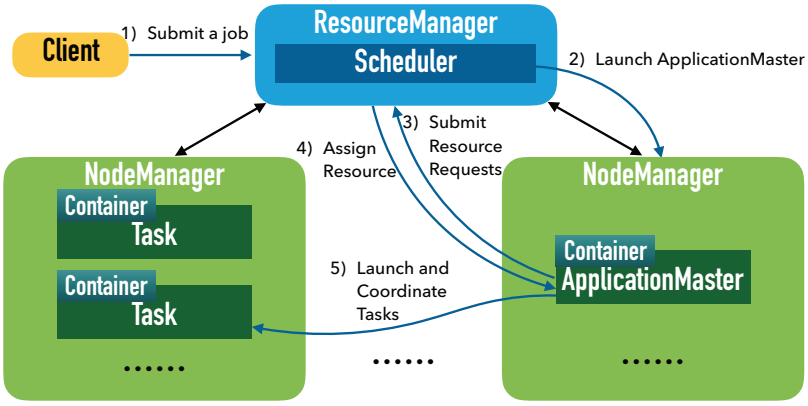

Figure 2.3: Job execution in Hadoop YARN

2.4 Typical Scheduling Policies

With a large volume of jobs submitted in a batch by multiple users, resource management and scheduling play an important role on the performance of every job and the overall system. Researchers have put tremendous efforts on job scheduling, resource management, program design, and Hadoop applications [15, 16, 17, 18, 19, 20, 21, 22, 23]. Among them, FIFO, Fair, and Capacity are typical ones that are provided by most of the big data computing systems.

In FIFO scheduling, all jobs are placed in a queue in the order of the submission time. The job submitted first is served first. Once its tasks have been all allocated for execution, the next job in the queue is run. Fair Scheduling aims to allocate fair resource sharing across jobs over time. Priorities and weights may be configured for

different jobs in resource allocation. Capacity scheduling offers similar functionality as the Fair scheduling. Under Capacity, multiple job queues are defined for different users. The scheduler offers equal resource for each queue while FIFO scheduling is provided for the jobs in the same queue.

Above scheduling policies are default settings in most computing systems such as Hadoop MapReduce, Hadoop YARN and Spark. However, they do not consider the efficiency of resource utilization of the system. In the following chapters, we introduce our resource management schemes and compare the system performance under our approaches with the one under default scheduling policies.

CHAPTER 3

WORKLOAD-BASED RESOURCE ALLOCATION

In this chapter, we focus on the scheduling problems on resource allocation for different stages of jobs based on their workloads. We proposes a new resource management scheme called FRESH. In FRESH, we develop a new monitoring component to detect the run-time workloads of each stage in the cluster. Based on the monitoring results, FRESH dynamically adjusts the resource allocation for different stages in order to improve the system resource utilization and reduce the makespan of batch jobs. In addition, an improved fairness mechanism in FRESH guarantees that equal resources are assigned to each running job in the cluster. FRESH is implemented in Hadoop MapReduce platform but its general mechanism can be easily utilized in other computing systems. The following parts of this chapter indicate the details of FRESH, including motivation, solution, and the evaluation of FRESH. We also evaluate the performance of default scheduling policies to show the improvement of FRESH in both system performance and fairness of applications.

3.1 Background and Motivation

One big challenge for Hadoop users is how to appropriately configure their systems. As a complex system, Hadoop is built with a large set of system parameters. While it provides the flexibility to customize a Hadoop cluster for various applications, it is often difficult for the user to understand those system parameters and set the optimal values for them. In this work, we target on an extremely important Hadoop parameter, slot configuration, and develop a suite of solutions to improve the

performance, with respect to the makespan of a batch of jobs and the fairness among them.

In a classic Hadoop cluster, each job consists of multiple map and reduce tasks. The concept of "slot" is used to indicate the capacity of accommodating tasks on each node in the cluster. Each node usually has a predefined number of slots and a slot could be configured as either a map slot or a reduce slot. The slot type indicates which type of tasks (map or reduce) it can serve. At any given time, only one task can be running per slot. While the slot configuration is critical for the performance, Hadoop by default uses fixed numbers of map slots and reduce slots at each node throughout the lifetime of a cluster. The values are usually set with heuristic numbers without considering job characteristics. Such static settings certainly cannot yield the optimal performance for varying workloads. Therefore, our main target is to address this issue and improve the makespan performance. Besides the makespan, fairness is another performance metric we consider. Fairness is critical when multiple jobs are allowed to be concurrently executed in a cluster. With different characteristics, each job may consume different amount of system resources. Without a careful plan and management, some jobs may starve while other take advantages and finish the execution much faster. Prior work has studied this issue and proposed some solutions. But we found that the previous work did not accurately define the fairness for this two-phase MapReduce process. In this work, we present a novel fairness definition that captures the overall resource consumption. Our solution also aims to achieve a good fairness among all the jobs.

Specifically, we propose a new approach, "FRESH", to achieving fair and efficient slot configuration and scheduling for Hadoop clusters. Our solution attempts to accomplish two major tasks: (1) decide the slot configuration, i.e., how many map/reduce slots are appropriate; and (2) assign map/reduce tasks to available slots. The targets of our approach include minimizing the makespan as the major objective and mean-

while improving the fairness without degrading the makespan. FRESH includes two models, static slot configuration and dynamic slot configuration. In the first model, FRESH derives the slot configuration before launching the Hadoop cluster and uses the same setting during the execution just like the conventional Hadoop. In the second model, FRESH allows a slot to change its type after the cluster has been started. When a slot finishes its task, our solution dynamically configures the slot and assigns it the next task. Our experimental results show that FRESH significantly improves the performance in terms of makespan and fairness in the system.

3.2 Problem Formulation

We consider that a user submits a batch of n jobs, $J = \{J_1, J_2, \ldots, J_n\}$, to a Hadoop cluster with S slots in total. Each job J_i contains $n_m(i)$ map tasks and $n_r(i)$ reduce tasks. Let s_m and s_r be the total numbers of map slots and reduce slots in the cluster, i.e., $S = s_m + s_r$. We assume that an admission control mechanism is in effect in the cluster such that there is an upper bound limit on the number of jobs that can run concurrently. Specifically, we assume in this work that at any time, there are at most k jobs running in map phase and at most k jobs running in reduce phase. Thus, the maximum number of active jobs in the cluster is $2k$. Here, k is a user-specified parameter for balancing the trade-off between fairness and makespan. Our objective is to minimize the makespan (i.e., the total completion length) of the job set J while achieving the fairness among these jobs as well.

To solve the problem, we develop a new scheduling solution FRESH for allocating slots to Hadoop tasks. Essentially, we need to address two issues. First, given the total number of slots, how to allocate them for map and reduce, i.e, how many map slots and reduce slots are appropriate. Second, when a slot is available, which task should be assigned to it. FRESH considers two models, i.e., static slot configuration (see Fig. 3.1) and dynamic slot configuration (see Fig. 3.2). In the first model, the numbers

of map and reduce slots are decided before launching the cluster, similar to the conventional Hadoop system. In this model, we assume that job profiles are available as prior knowledge. Our goal is to derive the best slot setting, thus addressing the first issue. During the execution, FAIR SCHEDULER is used to assign tasks to available slots. In the second model of dynamic slot configuration, FRESH allows a slot to change its type in an online manner and thus dynamically controls the allocation of map and reduce slots. In addition, FRESH includes an algorithm that assigns tasks to available slots.

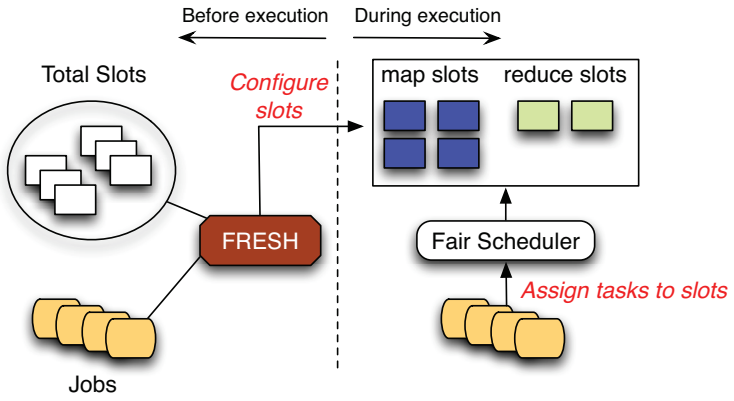

Figure 3.1: Static slot configuration.

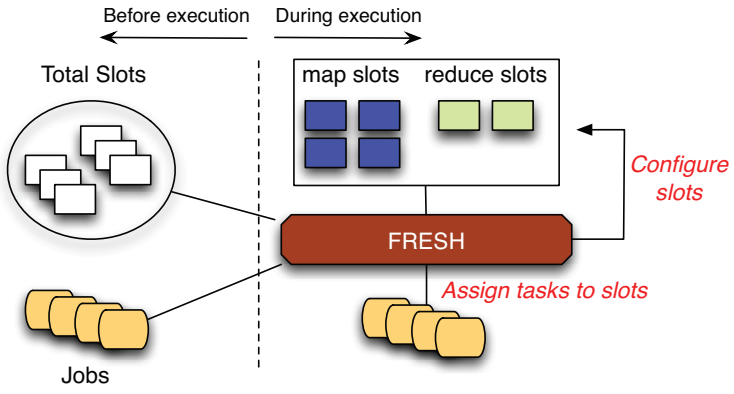

Figure 3.2: Dynamic slot configuration.

3.3 Our Solution: FRESH

In this section, we present our algorithm in FRESH. It contains two components: static slot configuration and dynamic slot configuration.

3.3.1 Static Slot Configuration

First, we present our algorithm in FRESH for static slot configuration, where the assignments of map and reduce slots are preset in configuration files and loaded when the Hadoop cluster is launched. Our objective is to derive the optimal slot configuration given the workload profiles of a set of Hadoop jobs.

We assume that the workload of each job is available as prior knowledge. This information can be obtained from historical execution records or empirical estimation. Let $\bar{t}_m(i)$ and $\bar{t}_r(i)$ be the average execution time of a map task and a reduce task of job J_i. We define $w_m(i)$ and $w_r(i)$ as the workloads of map tasks and reduce tasks of J_i, which represent the summation of the execution time of all map tasks and reduce tasks of J_i. Therefore, $w_m(i)$ and $w_r(i)$ can be defined as:

$$w_m(i) = n_m(i) \cdot \bar{t}_m(i), \quad w_r(i) = n_r(i) \cdot \bar{t}_r(i).$$

Let c_m and c_r represent the number of slots that a job can occupy to run its map and reduce tasks. Recall that FAIR SCHEDULER is used to assign tasks and each active job is evenly allocated slots for its tasks. In addition, under our admission control policy, a busy cluster has k jobs concurrently running in map phase and k jobs concurrently running in reduce phase. Therefore, we have c_m and c_r defined as follows:

$$c_m = \frac{s_m}{k}, \quad c_r = \frac{s_r}{k}.$$

We develop a new algorithm (see Algorithm 3.3.1) to derive the optimal static slot configuration. Our basic idea is to enumerate all possible settings of s_m and s_r, and calculate the makespan for any given pair (s_m, s_r). We use $\mathcal{M}$ and $\mathcal{R}$ to

represent the sets of jobs that are currently running in their map phase and reduce phase, respectively. Each element in $\mathcal{M}$ and $\mathcal{R}$ is the job index of the corresponding running job. Initially, $\mathcal{M} = \{1, 2, \ldots, k\}$ and $\mathcal{R} = \{\}$. According to their definitions, when job J_i finishes its map phase, the index i will be moved from $\mathcal{M}$ to $\mathcal{R}$. The sizes of $\mathcal{M}$ and $\mathcal{R}$ are upper-bounded by the parameter k. Additionally, we use W_i to represent the *remaining workload* of J_i in the current phase (either map or reduce phase). Before J_i enters the map phase, W_i is set to its workload of the map phase $w_m(i)$, i.e., $W_i = w_m(i)$. During the execution in the map phase, W_i will be updated according to the progress. When job J_i finishes its map phase, W_i will be set to its workload of the reduce phase, i.e., $W_i = w_r(i)$.

Algorithm 3.3.1 presents the details of our solution. The outer loop enumerates all possible slot configurations (i.e., s_m and s_r). For each particular configuration, we first calculate the workloads of each job's map and reduce phases, i.e., $w_m(i)$ and $w_r(i)$, and set the initial value of W_i (see lines 3–5). In line 6, we initialize some important variables, where $\mathcal{M}$ and $\mathcal{R}$ are as defined above, $\mathcal{R}'$ represents an ordered list of pending jobs that have finished their map phase, but have not entered their reduce phase yet, and T, initialized as zero, records the makespan of this set of jobs. The core component of the algorithm is the while loop (see lines 7–22) that calculates the makespan and terminates when both $\mathcal{M}$ and $\mathcal{R}$ are empty. In this loop, our algorithm mimics the execution order of all the jobs. Both $\mathcal{M}$ and $\mathcal{R}$ keep intact until one of the running jobs finishes its current map (or reduce) phase. In each round of execution in the while loop, our algorithm finds the first job that changes the status and then updates the job sets accordingly. This target job could be in either map or reduce phase depending on its *remaining workload* and the number of slots assigned to each job. In Algorithm 3.3.1, lines 8–9 find two jobs (J_u and J_v) which have the minimum remaining workloads in $\mathcal{M}$ and $\mathcal{R}$, respectively. These two jobs are the candidates to first finish their current phases. Variables c_m and

c_r represent the number of slots assigned to each of them under FAIR SCHEDULER scheduling policy. Thus, the remaining execution times for J_u and J_v to complete their phases are $\frac{W_u}{c_m}$ and $\frac{W_v}{c_r}$, respectively.

If J_u finishes its map phase first (the case in lines 10–16), then we remove u from $\mathcal{M}$, update the current makespan, and set the remaining workload of J_u to the workload of its reduce phase (line 11). We also update the remaining workloads of all other active jobs in $\mathcal{M}$ and $\mathcal{R}$ (lines 12–13). In addition, the algorithm picks a new job to enter its map phase in line 14. Finally, we add u to $\mathcal{R}$ to start its reduce phase if the capacity limit of $\mathcal{R}$ is not reached. Otherwise, u is added to the tail of the pending list $\mathcal{R}'$ (lines 15–16).

The function `DeductWorkload` is called to update the remaining workloads for active jobs in $\mathcal{M}$ or $\mathcal{R}$. As shown below, the inputs of this function include a job set $\mathcal{A}$ (e.g., $\mathcal{M}$, $\mathcal{R}$) and the value of the completed workload w. The remaing workload of each job i in $\mathcal{A}$ is then updated by deducting w.

```
function DeductWorkload(A, w){
    /*A: a set of job IDs, w: a workload value */
    for i ∈ A do W_i ← W_i − w end }
```

Once job J_v finishes its reduce phase (see the other case in lines 17–21), we update the current makespan as well as the remaining workloads of all other active jobs in $\mathcal{M}$ and $\mathcal{R}$. Similarly, index v is removed from $\mathcal{R}$. If $\mathcal{R}'$ is now not empty, then the first job in $\mathcal{R}'$ will be moved to $\mathcal{R}$. At the end, in lines 22–23, the algorithm compares the present makespan T to the variable Opt_MS which keeps track of the minimum makespan, and updates Opt_MS if needed. Another auxiliary variable Opt_SM is used to record the corresponding slot configuration. The time complexity of Algorithm 3.3.1 is $O(S \cdot N_T^2)$, where $N_T = \sum_i (n_m(i) + n_r(i))$ is the total number of tasks of all the jobs. In practice, the computation overhead of Algorithm 3.3.1

is quite small. For example, with 500 slots and 100 jobs each having 400 tasks, the computation overhead is 0.578 seconds on a desktop server with 2.4GHz CPU.

Algorithm 3.3.1: Static Slot Configuration

for $s_m = 1$ to S **do**
  $s_r = S - s_m$
  for $i = 1$ to n **do**
    $w_m(i) = n_m(i) \cdot \bar{t}_m(i), \quad w_r(i) = n_r(i) \cdot \bar{t}_r(i)$
    $W_i = w_m(i)$
  end for
  $\mathcal{M} = \{1, 2, \ldots, k\}, \mathcal{R} = \{\}, \mathcal{R}' = \{\}, T = 0$
  while $\mathcal{M} \bigcup \mathcal{R} \neq \phi$ **do**
    $u = \arg\min_{i \in \mathcal{M}} W_i, \quad c_m = \frac{s_m}{|\mathcal{M}|}$
    $v = \arg\min_{i \in \mathcal{R}} W_i, \quad c_r = \frac{s_r}{|\mathcal{R}|}$
    if $\frac{W_u}{c_m} < \frac{W_v}{c_r}$ **then**
      $\mathcal{M} \leftarrow \mathcal{M} - u, \quad T = T + \frac{W_u}{c_m}, \quad W_u = w_r(u)$
      DeductWorkload($\mathcal{M}, w_m(u)$)
      DeductWorkload($\mathcal{R}, \frac{w_m(u)}{c_m} \cdot c_r$)
      pick a new job from J and add its index to $\mathcal{M}$
      if $|\mathcal{R}| < k$ **then** $\mathcal{R} \leftarrow \mathcal{R} + u$
      else add u to the tail of $\mathcal{R}'$
    else
      $\mathcal{R} \leftarrow \mathcal{R} - v, \quad T = T + \frac{W_v}{c_r}$
      DeductWorkload($\mathcal{R}, W_v$)
      DeductWorkload($\mathcal{M}, \frac{W_v}{c_r} \cdot c_m$)
      if $|\mathcal{R}'| > 0$ **then** move $\mathcal{R}'[0]$ to $\mathcal{R}$
    end if
  end while
  if $T < Opt\_MS$ **then** $Opt\_MS = T, Opt\_SM = s_m$
end for
return Opt_SM and Opt_MS

3.3.2 Dynamic Slot Configuration

Then we turn to discuss the model in FRESH for dynamic slot configuration. The critical target of this model is to enable a slot to change its type (i.e., map or reduce) after the cluster is launched. To accomplish it, we develop solutions for both configuring slots and assigning tasks to slots. In addition, we redefine the fairness of resource consumption among jobs. Therefore, our goal is to minimize the makespan of

jobs while achieving the best fairness without degrading the makespan performance. The rest of this section is organized as follows. We first introduce the new *overall fairness* metric. Then we present the algorithm for dynamically configuring map and reduce slots. Finally, we describe how FRESH assigns tasks to the slots in the cluster.

3.3.2.1 Overall Fairness Measurement

Fairness is an important performance metric for our algorithm design. However, the traditional fairness definition does not accurately reflect the total resource consumptions of jobs. In this subsection, we present a new approach to quantify fairness measurement, where we define the resource usage in MapReduce process and use Jain's index [24] to represent the level of fairness.

In a conventional Hadoop system, FAIR SCHEDULER evenly allocates the map (resp. reduce) slots to active jobs in their map (resp. reduce) phases. Although the fairness is achieved in map and reduce phases separately, it does not guarantee the fairness among all jobs when we combine the resources (slots) consumed in both map and reduce phases. For example, assume a cluster has 4 map slots and 4 reduce slots running the following 3 jobs: J_1 (2 map tasks and 9 reduce tasks), J_2 (3 map tasks and 4 reduce tasks), and J_3 (7 map tasks and 3 reduce tasks). Assume every task can be finished in a unit time. The following table shows the slot assignment with FAIR SCHEDULER at the beginning of each time point ('M' and 'R' indicate the type of slots allocated for the jobs). Eventually, all three jobs are finished in 5 time units. However, they occupy 11, 7, and 10 slots respectively.

	0	1	2	3	4
J_1	2(M)	4(R)	2(R)	1(R)	2(R)
J_2	1(M)	2(M)	2(R)	1(R)	1(R)
J_3	1(M)	2(M)	4(M)	2(R)	1(R)

In this work, we define a new fairness metric, named *overall fairness*, as follows. At any time point T, let J' represents the set of currently active jobs in the system, and T_i represents the starting time of job J_i in J'. We use two matrices $t_m[i,j]$ and $t_r[i,j]$ to represent the execution times of job J_i's j-th map task and j-th reduce task, respectively. Note that these two matrices include the unfinished tasks. Therefore, the resources consumed by job J_i by time T can be expressed as

$$r_i(T) = \frac{\sum_j t_m[i,j] + \sum_j t_r[i,j]}{T - T_i}. \tag{3.1}$$

where the above formula represents the effective resources J_i has consumed during the period of $T - T_i$. The bigger $r_i(T)$ is, the more resources J_i has been assigned to. In addition, we use Jain's index on r_i to indicate the overall fairness ($F(T)$) at the time point T, i.e.,

$$F(T) = \frac{(\sum_i r_i(T))^2}{|J'| \sum_i r_i^2(T)},$$

where $F(T) \in [\frac{1}{|J'|}, 1]$ and a larger value indicates better fairness.

3.3.2.2 Configure Slots

The function of configuring slots is to decide how many slots should serve map/reduce tasks based on the current situation. Specifically, when a task is finished and a slot is freed, our system needs to determine the type of this available slot in order to serve other tasks. In this subsection, we present the algorithm in FRESH that appropriately configures map and reduce slots.

First of all, our solution makes use of the statistical information of the finished tasks from each job. This information is available in Hadoop system variables and log files. Let $\bar{t}_m(i)$ and $\bar{t}_r(i)$ be the average execution times of job J_i's map task and reduce task, respectively. Once a task completes, we can access its execution time and then update $\bar{t}_m(i)$ or $\bar{t}_r(i)$ for job J_i which that particular task belongs

to. In addition, we use $n'_m(i)$ and $n'_r(i)$ to indicate the number of remaining map tasks and reduce tasks, respectively, in job J_i. The *remaining workload* of a job J_i is then defined as follows: $w'_m(i) = n'_m(i) \cdot \bar{t}_m(i), w'_r(i) = n'_r(i) \cdot \bar{t}_r(i)$, where $w'_m(i)$ is J_i's remaining map workload and $w'_r(i)$ is the remaining reduce workload of J_i. Finally, we estimate the total remaining workloads of all the pending map and reduce tasks. Let RW_m represents the summation of all the remaining map workloads of jobs in $\mathcal{M}$ while RW_r represents the summation of all the remaining reduce workloads for jobs in $\mathcal{R}$ and $\mathcal{R}'$. RW_m and RW_r can be calculated as:

$$RW_m = \sum_{i \in \mathcal{M}} w'_m(i), \quad RW_r = \sum_{i \in \mathcal{R} \bigcup \mathcal{R}'} w'_r(i).$$

Note that RW_m includes the jobs running in their map phases (in $\mathcal{M}$) while RW_r includes the jobs running in their reduce phases (in $\mathcal{R}$) as well as the jobs that have finished their map phases but wait for running their reduce phase (in $\mathcal{R}'$).

The intuition of the algorithm in FRESH is to keep jobs in their map and reduce phases in a consistent progress so that all jobs can be properly pipelined to avoid waiting for slots or having idle slots. Therefore, the numbers of map and reduce slots should be proportional to the total remaining workloads RW_m and RW_r, i.e., the heavier loaded phase gets more slots to serve its tasks. However, this idea may not work well in the map-reduce process because there could be a sudden change on the remaining workloads. The problem arises when a job finishes its map phase and enters the reduce phase. Based on the definition of RW_m and RW_r, this job will bring its reduce workload to RW_r and a new job which starts its map phase then add its new map workload to RW_m. Such workload updates, however, could greatly change the weights of RW_m and RW_r. For example, if $RW_m >> RW_r$, most of slots are map slots. Assume a reduce-intensive job just finishes its map phase and incurs a lot of reduce workload, the system has no sufficient reduce slots to serve the new reduce tasks. It takes some time for the cluster to adjust to this sudden workload

change as it has to wait for the completions of many map tasks before configuring those released slots to be reduce slots.

We develop Algorithm 3.3.2 to derive the optimal slot configuration in an online mode. We follow the basic design principal with a threshold-based control to mitigate the negative effects from those sudden changes in map/reduce workloads. When a map/reduce task is finished, the algorithm collects the task execution time and updates a set of statistical information including the average task execution time, the number of remaining tasks, the remaining workload of job J_i, and the total remaining workloads (see lines 1–4). Following that, the algorithm calls a function, called `CalExpSm`, to calculate the expected number of map slots ($expSm$) based on the current statistical information. If the expectation is more than the current number of map slots (s_m), this free slot will become a map slot. Otherwise, we set it to be a reduce slot.

Algorithm 3.3.2: Configure a Free Slot

1: **if** a map task of job J_i is finished **then**
2: update $\bar{t}_m(i)$, $n'_m(i)$, $w'_m(i)$ and RW_m
3: **end if**
4: **if** a reduce task of job J_i is finished **then**
5: update $\bar{t}_r(i)$, $n'_r(i)$, $w'_r(i)$ and RW_r
6: **end if**
7: $expSm$ =`CalExpSm()`
8: **if** $expSm > s_m$ **then** set the slot to be a map slot
9: **else** set the slot to be a reduce slot

The details of the function `CalExpSm` are presented in Algorithm 3.3.3. We use θ_{cur} to represent the expected ratio of map slots based on the current remaining workload. In line 2, we choose an active job J_a which has the minimum map workload, i.e., job J_a is supposed to first finish its map phase. If J_a is still far from the completion of its map phase, then the risk of having a sudden workload change is low and the function just returns $\theta_{cur} \cdot S$ as the expected number of map slots. We set a parameter τ_1 as the progress threshold. When job J_a is close to the end of its map phase, i.e., the progress

exceeds τ_1, the function will consider the potential issue with the sudden change of the workload (lines 6–13). Essentially, the function tries to estimate the map and reduce workloads when J_a enters its reduce phase, and calculate the expected ratio of map slots θ_{exp} at that point. A sudden change of the workload happens when θ_{exp} is quite different from the ratio of map slots θ_{cur} we get based on the current configuration. In the case that a sudden change is predicted, we will use $\theta_{exp} \cdot S$ instead as the guideline for new slot configuration. Otherwise, the function still returns $\theta_{cur} \cdot S$.

Specifically in Algorithm 3.3.3, when J_a finishes its remaining map workload $w'_m(a)$, we assume the other jobs in $\mathcal{M}$ have made roughly the same progress. Thus the total map workload will be reduced by $w'_m(a) \cdot k$. Then a new job will join the set $\mathcal{M}$, let it be job J_b, and $w_m(b)$ will be added to the total remaining workload RW_m (line 7). Meanwhile, J_a will belong to either set $\mathcal{R}$ or $\mathcal{R}'$ and its reduce workload $w_r(i)$ will be added to the total remaining reduce workload RW_r (line 8). Variable θ_{exp} in line 9 denotes the expected ratio of map slots at that point. Next, the function estimates the number of map slots following the configuration ratio θ_{cur} when J_a finishes its map phase. It involves the number of slots freed from the current point. It is apparent that there is no other map slot released based on the definition of J_a, thus we just need to estimate the number of available reduce slots during this period. In Algorithm 3.3.3, we use η to represent this number and estimate its value based on the following Theorem 3.3.1. In line 11, we predict the number of map slots s'_m using the current configuration ratio, i.e., $\eta \cdot \theta_{cur}$ slots will become map slots. Eventually, the function compares the estimated ratio to the expected ratio in line 12. If the difference is over a threshold τ_2, we will consider the future expected ratio θ_{exp}. Otherwise, we will continue to use the current configuration ratio θ_{cur}.

Theorem 3.3.1 *Assume reduce tasks are finished at a rate of one per r time units, the number (η) of available reduce slots when J_a finishes its remaining map workload is equal to:*

Algorithm 3.3.3: Function $CalExpSm()$

1: $\theta_{cur} = \frac{RW_m}{RW_m + RW_r}$
2: $a = \arg\min_{i \in \mathcal{M}} w'_m(i)$
3: **if** the progress of job $J_a < \tau_1$ **then**
4: return $\theta_{cur} \cdot S$
5: **else**
6: Let job J_b be the next that will start its map phase
7: $RW'_m = RW_m - w'_m(a) \cdot k + w_m(b)$
8: $RW'_r = RW_r + w_r(a)$
9: $\theta_{exp} = RW'_m/(RW'_r + RW'_m)$
10: calculate η using Theorem 3.3.1
11: $s'_m = s_m + \theta_{cur} \cdot \eta$
12: **if** $\frac{|\frac{s'_m}{S} - \theta_{exp}|}{\theta_{exp}} > \tau_2$ **then** return $\theta_{exp} \cdot S$
13: **else** return $\theta_{cur} \cdot S$
14: **end if**

$$\eta = \frac{\sqrt{m_a^2 + 4 \cdot c \cdot w'_m(a)} - m_a}{2 \cdot c \cdot r}.$$

where $c = \frac{\theta_{cur}}{2 \cdot r \cdot k}$, $w'_m(a)$ *indicates the remaining map workload of* J_a, *and* m_a *is the number of map slots assigned to* J_a.

Proof Assume job J_a will finish its remaining map workload $w'_m(a)$ in x time units. According to our assumption, a reduce task will be finished every r time units. Therefore, when J_a finishes its map phase, there will be $\eta = \frac{x}{r}$ reduce slots released as well. Assume that we use θ_{cur} to allocate slots and k jobs in $\mathcal{M}$ are evenly assigned newly released slots. Job J_a will continuously obtain $\frac{x \cdot \theta_{cur}}{r \cdot k}$ new map slots. It is equivalent to having half of them $\frac{x \cdot \theta_{cur}}{2 \cdot r \cdot k}$ from the beginning. Therefore, the remaining time for J_a to finish the map phase can be estimated as

$$x = w'_m(a)/(\frac{x \cdot \theta_{cur}}{2 \cdot r \cdot k} + m_a),$$

where m_a is the number of map slots currently assigned to J_a. By solving the above equation, we have

$$x = \frac{\sqrt{m_a^2 + 4 \cdot c \cdot w'_m(a)} - m_a}{2 \cdot c}.$$

Thus:

$$\eta = \frac{\sqrt{m_a^2 + 4 \cdot c \cdot w'_m(a)} - m_a}{2 \cdot c \cdot r}.$$

3.3.2.3 Assign Tasks to Slots

Once the type of the released slot is determined, FRESH will assign a task to that slot. Basically, we need to select an active job and let the available map/reduce slot serve a map/reduce task from that job. In FRESH, we follow the basic idea in FAIR SCHEDULER but use the new *overall fairness* metric instead: calculate the resource consumption for each job based on Eq.(3.1) and choose the job with the most deficiency of *overall fairness*.

Algorithm 3.3.4: Assign a Task to a Slot

```
1:  Initial: C = {}, now ← current time in system
2:  if the slot is configured for map tasks then C ← M
3:  else C ← R
4:  for each job J_i ∈ C do
5:     total_i = 0
6:     for each task j in job J_i do
7:        if task j is finished then e_j ← f_j − s_j
8:        else if task j is running then e_j ← now − s_j
9:        else e_j ← 0
10:       total_i = total_i + e_j
11:    end for
12:    r_i = total_i / (now − T_i)
13: end for
14: s = arg min_{i∈A} r_i
15: assign a task of job J_s to the slot
```

Algorithm 3.3.4 illustrates our solution in FRESH for assigning a task to an available slot. We use $\mathcal{C}$ to indicate a set of candidate jobs. Initially, $\mathcal{C}$ is empty and we use variable *now* to indicate the current time. When a slot is configured to serve map tasks (using Algorithm 3.3.2), $\mathcal{C}$ is a copy of $\mathcal{M}$. Otherwise, $\mathcal{C}$ is a copy of $\mathcal{R}$ (lines 2–3). The outer loop (lines 4–11) then calculates the resource consumption for each job in $\mathcal{C}$ at the current time. Variable $total_i$, initialized as 0 in line 5, is used

to record the total execution time for all finished and running tasks in job J_i (lines 6–10). We use e_j to denote the execution time of task j. If task j has been finished, its execution time e_j is the difference between its finish time f_j and its start time s_j (line 7). If task j is still running, then e_j is equal to the current time now deducted by its start time s_j (line 8). Once the total execution time for job J_i is obtained, we get the resources consumption r_i by normalizing the total execution time $total_i$ by the duration between the current time and the start time of job J_i (T_i), as shown in line 11. Finally, the job with the minimum resource consumption is chosen to be served by the available slot.

3.4 Performance Evaluation

In this section, we evaluate the performance of FRESH and compare it with other alternative schemes. We use FRESH-static and FRESH-dynamic to represent our static slot configuration and dynamic slot configuration respectively.

3.4.1 Experimental Setup and Workloads

First, we introduce our implementation details, the cluster setting and the workload for the evaluation.

Implementation

We have implemented FRESH on Hadoop version 0.20.2. For FRESH-static, we develop an external program to derive the best slot setting and apply it to the Hadoop configuration file. The Hadoop system itself is not modified for FRESH-static. To implement FRESH-dynamic, we have added a few new components to Hadoop. First, we implement the admission control policy with the parameter k, i.e., at most k jobs are allowed to be concurrently running in map phase or in reduce phase. Second, we create two new modules in $JobTracker$. One module updates the statistical information such as the average execution time of a task, and estimates

the remaining workload of each active job. The other module is designed to configure a free slot to be a map slot or a reduce slot and assign a task to it according to the algorithms in section 3.3.2. The two threshold parameters in Algorithm 3.3.3 are set as $\tau_1 = 0.8$ and $\tau_2 = 0.6$. We have tested with different values and found that the performance is close when $\tau_1 \in [0.7, 0.9]$ and $\tau_2 \in [0.5, 0.7]$. Due to the page limit, we omit the discussion about these two heuristic values. In addition, job profiles (execution time of tasks) are generated based on the experimental results when a job is individually executed. However, we randomly introduce $\pm 30\%$ bias to the measured execution time and use them as the job profiles representing rough estimates.

3.4.1.1 Hadoop Cluster

All the experiments are conducted on Amazon AWS cloud computing platform. We create a Hadoop clusters consisting of 11 m1.xlarge Amazon EC2 instances [25], one master node and 10 slave nodes. Each node is set to have 4 slots since an m1.xlarge instance at Amazon EC2 has 4 virtual cores. Totally, there are $S = 40$ slots in the cluster. In addition, to represent the scalability of FRESH, Fig. 3.10 shows a set of experiments in a larger cluster which doubles the slave nodes, i.e., one master node and 20 slave nodes with $S = 80$ slots totally.

3.4.1.2 Workloads

Our workloads for evaluation consider general Hadoop benchmarks with large datasets as the input. In particular, four datasets are used in our experiments including 4GB/8GB wiki category links data, and 4GB/8GB movie rating data. The wiki data includes the information about wiki page categories and the movie rating data is the user rating information. We choose the following six Hadoop benchmarks from Purdue MapReduce Benchmarks Suite [26] to evaluate the performance.

- *Classification*: Take the movie rating data as input and classify the movies based on their ratings.

- *Invertedindex*: Take a list of Wikipedia documents as input and generate word to document indexing.
- *Wordcount*: Take a list of Wikipedia documents as input and count the occurrences of each word.
- *Grep*: Take a list of Wikipedia documents as input and search for a pattern in the files.
- *Histogram Rating*: Generate a histogram of the movie rating data (with 5 bins).
- *Histogram Movies*: Generate a histogram of the movie rating data (with 8 bins).

3.4.2 Performance Evaluation

In this subsection, we present the performance of FRESH and compare to other solutions. Given a batch of MapReduce jobs, our major performance metrics are makespan, i.e., the finish time of the last job, and fairness among all jobs. We mainly compare to the conventional Hadoop system with the Fair scheduler and static slot configuration. In our setting, each slave has 4 slots, thus there are three possible static settings in conventional Hadoop, 1 map slot/3 reduce slots, 2 map slots/2 reduce slots, and 3 map slots/1 reduce slot. We use *Fair-1:3*, *Fair-2:2*, and *Fair-3:1* to represent these three settings respectively.

We have conducted two categories of tests with different job workloads, *simple workloads* consist of the same type of jobs (selected from the six MapReduce benchmarks), and *mixed workloads* represent a set of hybrid jobs. In the reset of this subsection, we separately present the evaluation results with these two categories of workloads.

3.4.2.1 Performance with Simple Workloads

For testing *simple workloads*, we generate 10 Hadoop jobs for each of the above 6 benchmarks. Every set of 10 jobs share the same input data set and they are consecutively submitted to the Hadoop cluster with an interval of 2 seconds. In addition, we have tested different values of k to show the effect of the admission control policy, particularly $k = 1, 3, 5, 10$.

Fig. 3.3 shows the performance of the makespan. First, we observe that in conventional Hadoop, the best performance is achieved mostly when $k = 1$, i.e., only one job in map and reduce phase which is equivalent to FIFO(First-In-First-Out) scheduler. It indicates that while improving the fairness, the FAIR SCHEDULER sacrifices the makespan of a set of jobs. The major cause is the resource contention among jobs which prolongs the execution of each task. Our solution, FRESH-static performs no worse than the best setting with FAIR SCHEDULER. In some workload such as "Classification", the improvement is adequate. In addition, FRESH-dynamic always yields a significant improvement in all the tested settings. For example, FRESH-Dynamic improves in average about 32.75% in makespan compared to the FAIR SCHEDULER with slot ratio 2:2. In addition, our dynamic slot configuration mitigates the effect of different values of k, thus holding a relatively flat curve in Fig. 3.3.

Fig. 3.4 shows the overall fairness designed in section. 3.2 during the same experiments. In most of the cases, the fairness value is a increasing function on k. Especially when $k = 10$, where all jobs are allowed to be concurrently executed (no admission control), almost all settings obtain a good fairness value. Since all jobs are from the same benchmark in *simple workload*, they finish their map phases in wave and enter the reduce phase roughly in the same time. Thus the FAIR SCHEDULER performs well in this case (k=10). Overall, FRESH-Dynamic outperforms all other schemes and achieves very-close-to-1 fairness value even when $k = 3$ or $k = 5$.

3.4.2.2 Performance with Mixed Workloads

Additionally, we evaluate the performance with mixed workloads consisting of different benchmarks. We specifically form five sets (Set A to Set E) of mixed jobs whose details will be introduced when we present the evaluation results. For mixed workloads, the order of the jobs has a big impact on the performance. In our experiments, after generating all the jobs, we conduct the Johnsons's algorithm [27], which can build an optimal two-stage job schedule, to determine the order, i.e., the workload for evaluation is a *ordered* list of mixed jobs.

Our mixed workloads are specified as follows. Set A∼C contain equal number of jobs from every benchmark. The details of Set A are listed in the following Table 3.1. It has two jobs from each benchmark, one job uses 4G data set and the other uses 8G dataset. Set B is a smaller workload, with one job from each benchmark, and all jobs use 8G dataset. In addition, Set C represent a large workload which doubles the workload of Set A, i.e., 4 jobs from each benchmark.

Table 3.1: Set A: 12 mixed jobs

JobID	**Benchmark**	**Dataset**	**Map #**	**Reduce #**
01	Classification	8GB Movie Rating	270	250
02	Classification	4GB Movie Rating	129	120
03	Invertedindex	8GB Wikipedia	256	250
04	Invertedindex	4GB Wikipedia	128	120
05	Wordcount	8GB Wikipedia	256	200
06	Wordcount	4GB Wikipedia	128	100
07	Grep[a-g][a-z]*	8GB Wikipedia	270	250
08	Grep[a-g][a-z]*	4GB Wikipedia	128	100
09	Histogram_ratings	8GB Movie Rating	270	250
10	Histogram_ratings	4GB Movie Rating	129	120
11	Histogram_movies	8GB Movie Rating	270	200
12	Histogram_movies	4GB Movie Rating	129	100

In addition, we create another two sets of jobs (Set D and Set E) to represent map-intensive and reduce-intensive workloads. Based on our experiments with simple workloads, benchmark "Inverted Index" and "Grep" are reduce-intensive, and

"Classification" and "Histogram Rating" are map-intensive. Thus, we set 12 jobs in Set D with 8 jobs from map-intensive benchmarks and 12 jobs in Set E with 8 jobs from reduce-intensive benchmarks.

Table 3.2: Set D: 12 map-intensive mixed jobs

Benchmark	**Job #**	**Dataset**	**Map #**	**Reduce #**
Classification	2	8GB Movie Rating Data	270	250
Classification	2	4GB Movie Rating Data	129	120
Wordcount	1	8GB Wikipedia	256	200
Wordcount	1	4GB Wikipedia	128	100
Histogram_ratings	2	8GB Movie Rating Data	270	250
Histogram_ratings	2	4GB Movie Rating Data	129	120
Histogram_movies	1	8GB Movie Rating Data	270	200
Histogram_movies	1	4GB Movie Rating Data	129	100

Table 3.3: Set E: 12 reduce-intensive mixed jobs

Benchmark	**Job #**	**Dataset**	**Map #**	**Reduce #**
Invertedindex	2	8GB Wikipedia	256	250
Invertedindex	2	4GB Wikipedia	128	120
Wordcount	1	8GB Wikipedia	256	200
Wordcount	1	4GB Wikipedia	128	100
Grep[a-g][a-z]*	2	8GB Wikipedia	270	250
Grep[a-g][a-z]*	2	4GB Wikipedia	128	100
Histogram_movies	1	8GB Movie Rating Data	270	200
Histogram_movies	1	4GB Movie Rating Data	129	100

First, Fig. 3.5 shows the makespan and the overall fairness with Set A and different values of k. For the makespan, FRESH-dynamic is always superior to FAIR SCHEDULER with static slot configurations. The best performance in the makespan of FAIR SCHEDULER is achieved when the ratio of map slots to reduce slots is 2:2 (*Fair-2:2*). Compare to *Fair-2:2*, FRESH-dynamic still improves the makespan by 27.62% when $k = 5$. Compared to the performance of the overall fairness with *simple workload*, the FAIR SCHEDULER performs much worse with the mixed workloads (the best value is around 0.8) because diverse jobs finish their map phases at different

time points. On the other hand, FRESH-Dynamic significantly improves the overall fairness, especially when $k \geq 5$ (with fairness values around 0.95). However, conventional FAIR SCHEDULER yields much lower fairness values with the mixed workloads (the best value is around 0.8). handle the overall fairness with different values of k. When k is greater than 1, FRESH achieves much better fairness than Fair scheduler.

Fig. 3.6 illustrates the tasks execution and slots assignments. Different colors denote different jobs in the set. The axis y indicates the slot index, where slot 1$\sim$40 are all possible map slots and slot 41 $\sim$ 80 are all possible reduce slots in the cluster. Recall that there are totally 40 slots in the cluster. The entire execution process can be divided into three stages. The first stage is from the beginning to about 650 seconds. As no job finishes its map phase and no reduce task is available, all slots are assigned to map phase. The second stage is from 650 seconds to 1500 seconds. After the first job finishes its map phase, FRESH-dynamic starts to leverage the number of map slots and reduce slots in the cluster according to the total remaining workloads in map phase and reduce phase. In the beginning of this stage (around 650 seconds), only a few slots are assigned as reduce slots, while most slots are still occupied by map tasks since the total remaining workload in map phase is heavier. Starting from about 800 seconds to 1000 seconds, the number of map slots becomes similar to the number of reduce slots as the remaining workloads in map phase and reduce phase are getting close. After 1000 seconds, the remaining workload eventually dominates, and more slots are assigned to reduce phase. The third stage is the last 300 seconds in the execution, where all map tasks are finished and all slots are reduce slots. Furthermore, Fig. 3.7 shows the overall fairness during the experiment with Set A.

In addition, we present the results with Set B$\sim$E. We test FRESH-Dynamic with three different settings of k: only one job, half of the jobs, and all the jobs, represented by *FRESH:1*, *FRESH:half*, and *FRESH:all* respectively. We compare to the conventional Hadoop with FIFO scheduler, Capacity scheduler [28] and FAIR SCHED-

Table 3.4: Set B: 6 mixed jobs

JobID	Benchmark	Dataset	Map #	Reduce #
01	Classification	8GB Movie Rating Data	270	250
02	Invertedindex	8GB Wikipedia	256	250
03	Wordcount	8GB Wikipedia	256	200
04	Grep[a-g][a-z]*	8GB Wikipedia	270	250
05	Histogram_ratings	8GB Movie Rating Data	270	250
06	Histogram_movies	8GB Movie Rating Data	270	200

Table 3.5: Set C: 24 mixed jobs

Benchmark	Job #	Dataset	Map #	Reduce #
Classification	2	8GB Movie Rating Data	270	250
Classification	2	4GB Movie Rating Data	129	120
Invertedindex	2	8GB Wikipedia	256	250
Invertedindex	2	4GB Wikipedia	128	120
Wordcount	2	8GB Wikipedia	256	200
Wordcount	2	4GB Wikipedia	128	100
Grep[a-g][a-z]*	2	8GB Wikipedia	270	250
Grep[a-g][a-z]*	2	4GB Wikipedia	128	100
Histogram_ratings	2	8GB Movie Rating Data	270	250
Histogram_ratings	2	4GB Movie Rating Data	129	120
Histogram_movies	2	8GB Movie Rating Data	270	200
Histogram_movies	2	4GB Movie Rating Data	129	100

uler. For Capacity scheduler, we create two queues and each queue has the same number of task slots. Each queue can obtain at most 90% slots in the cluster. Also, we separate jobs equally to these queues. The test results are illustrated in Fig. 3.8 and Fig. 3.9. In Fig. 3.8, in most of the cases, Fair scheduler yields the worst performance of makespan with different sets of jobs. On average, FRESH improves 31.32% of makespan compared to the Fair scheduler and 25.1% to Capacity scheduler. When a set of jobs is neither map-intensive or reduce-intensive (Set B and Set C), FIFO performs as well as FRESH. However, when workloads in map and reduce phases are unbalanced (Set D and Set E), FRESH improves 24.47% of makespan compared to FIFO. Overall, FRESH achieves an excellent and stable makespan per-

formance with different sets of jobs. In Fig. 3.9, FIFO apparently yields the worst performance in fairness. When the number of concurrently running jobs k is greater than 1, the performance of FRESH outperforms the Fair scheduler with above 0.95 fairness values in all the tested cases. Especially, in the experiment with Set B, FRESH achieves almost 1 fairness. Finally, we test Set B on a large cluster with 20 slave nodes and the results are shown in Fig. 3.10. We can observe a consistent performance gain from FRESH as in the smaller cluster of 10 slave nodes. Compared to Fair scheduler, FRESH reduces the makespan by 31.1%. In summary, FRESH yields a significant improvement in both makespan and fairness under both simple and mixed workloads.

3.5 Related Work

Job scheduling is an important direction for improving the performance of a Hadoop system. The default FIFO scheduler cannot work fairly in a shared cluster with multiple users and a variety of jobs. Fair scheduler [29] and Capacity Scheduler [28] are widely used to ensure each job can get a proper share of the available resources. Both of them consider fairness separately in map phase and reduce phase, but not the overall executions of jobs.

To improve the performance, Quincy [17] and Delay Scheduling [15] optimize data locality in the case of Fair scheduler. But these techniques trade fairness off against data locality. Coupling Scheduler in [30, 31, 32] aims to mitigate the starvation of reduce slots in Fair scheduler and analyze the performance by modeling the fundamental scheduling characteristics for MapReduce. W. Wang [33] presents a new queueing architecture and proposes a map task scheduling to strike the right balance between data-locality and load-balancing. Another category of schedulers consider user-level goals while improving the performance. ARIA [18] allocates the appropriate amounts of resources to the jobs to meet the predefined deadline. iShuf-

fle [34] separates shuffle phase from reduce tasks and provides a platform service to manage and schedule data output from map phase. However, all these techniques are still based on static slot configurations.

Another important direction to improve performance in Hadoop is the resource aware scheduling. RAS [22] aims at improving resource utilization across machines and meeting jobs completion deadline. MROrchestrator [23] introduces an approach to detect task resource utilization at each TaskTracker as a local resource manager and allocate resources to tasks at the JobTracker as a global resource manager. Furthermore, some other work is focused on heterogeneous environments. M. Zaharia et al. proposes a LATE scheduler [35] to stop unnecessary speculative executions in a heterogeneous Hadoop cluster. LsPS [36] uses the present heterogeneous job size patterns to tune the scheduling schemes.

Finally, we have proposed TuMM [37, 38] in our prior work which dynamically adjusts slots configurations in Hadoop based on FIFO. This work, however, considers concurrent execution of multiple jobs, which is a completely different and more complicated problem setting. We also include a new objective of fairness in this work.

3.6 Summary

This work focus on the problem of resource allocation to various stages of multiple jobs. We choose Hadoop MapReduce as the representative. We study a Hadoop cluster serving a batch of MapReduce jobs. We target on the slot configuration and task scheduling problems. We develop FRESH, an enhanced version of Hadoop, which supports both static and dynamic slot configurations. In addition, we present a novel definition of the overall fairness. Our solution can yield good makespans while the fairness is also achieved. We have conducted extensive experiments with various

workloads and settings. FRESH shows a significant improvement on both makespan and fairness compared to a conventional Hadoop system.

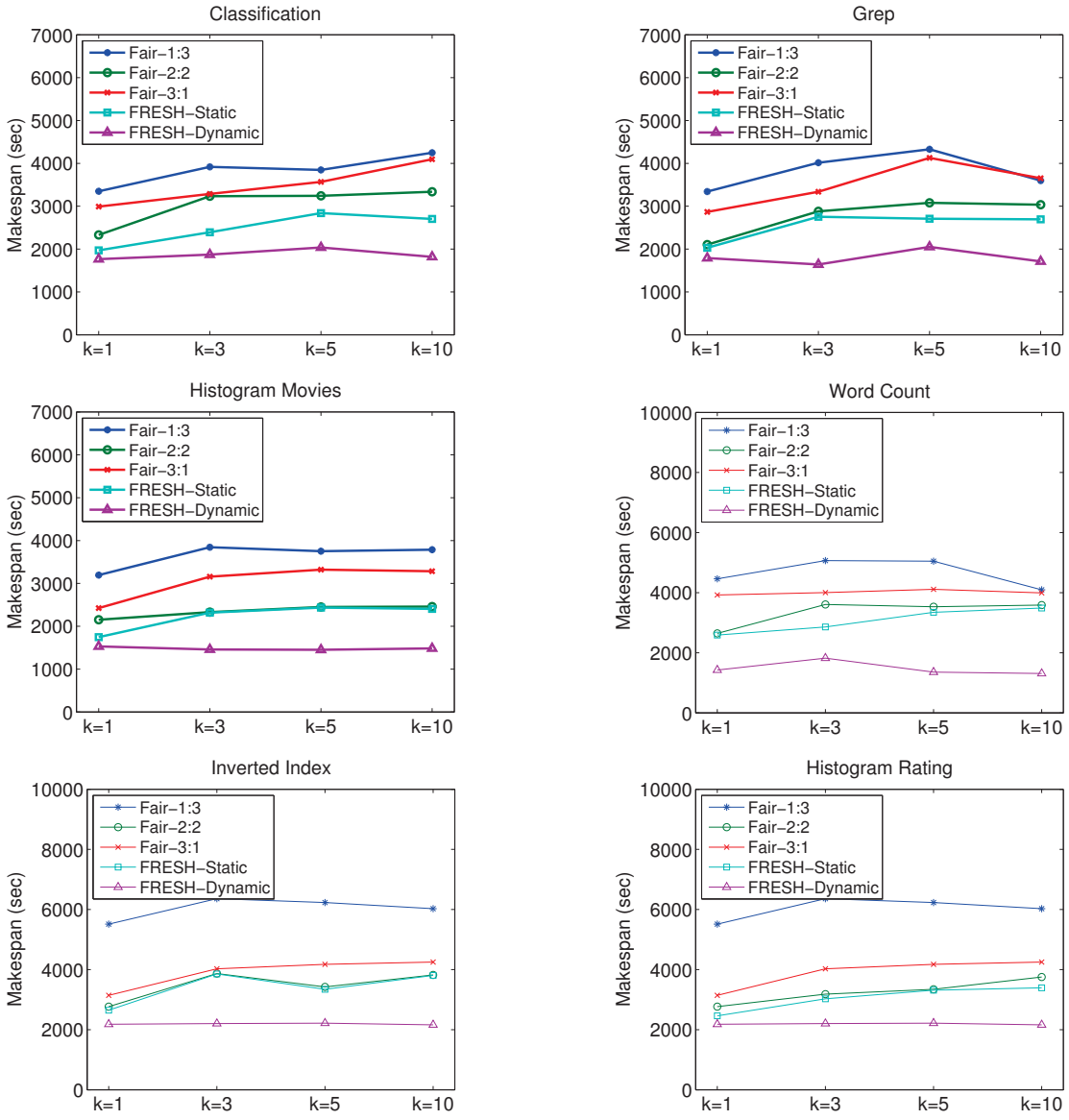

Figure 3.3: Makespan of simple workloads under FRESH and FAIR SCHEDULER

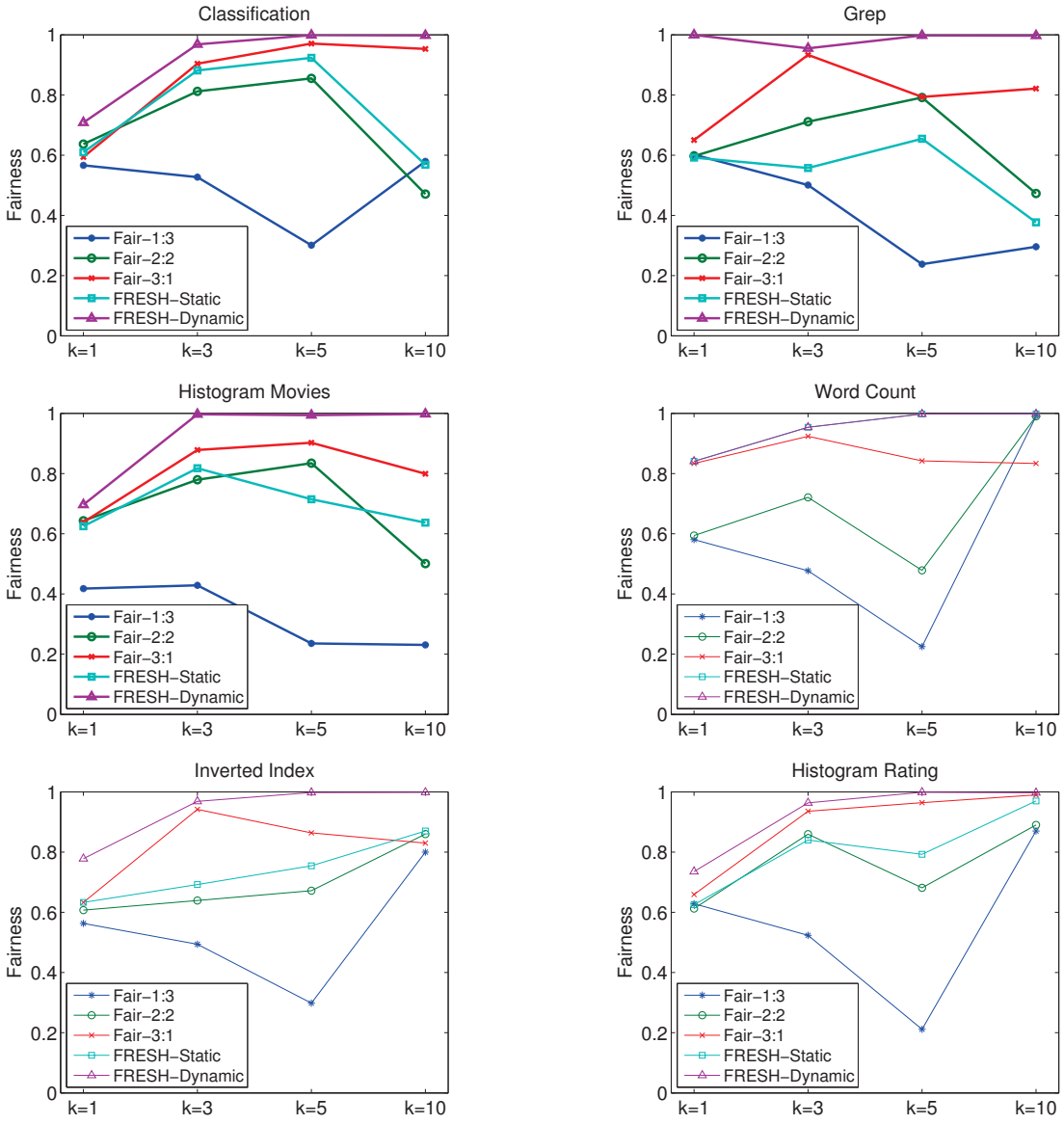

Figure 3.4: Fairness of of simple workloads under FRESH and FAIR SCHEDULER

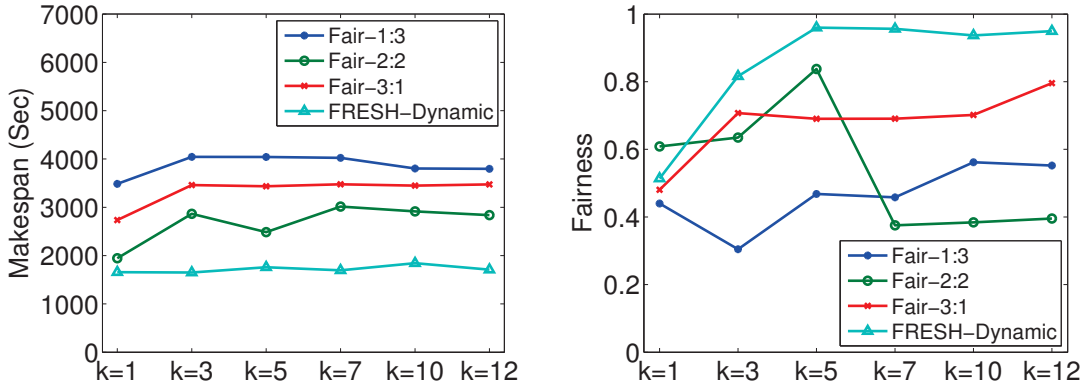

Figure 3.5: Makespan and fairness of set A with different values of k

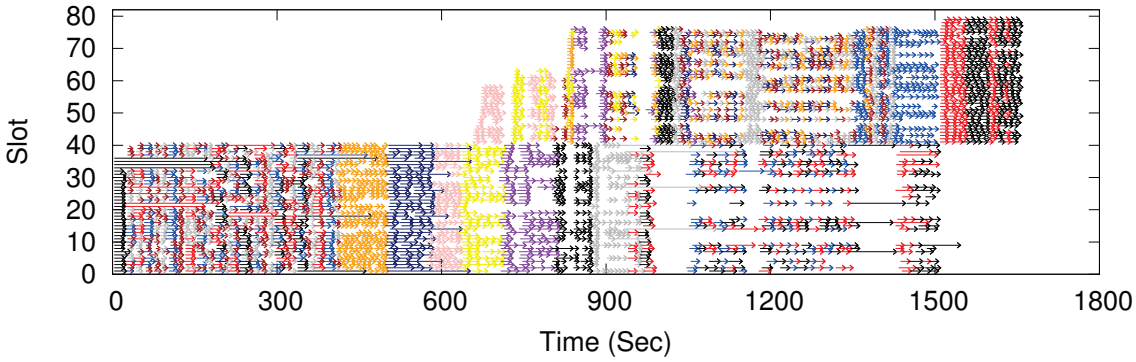

Figure 3.6: Set A tasks execution and slots assignments with FRESH-dynamic($k = 5$)

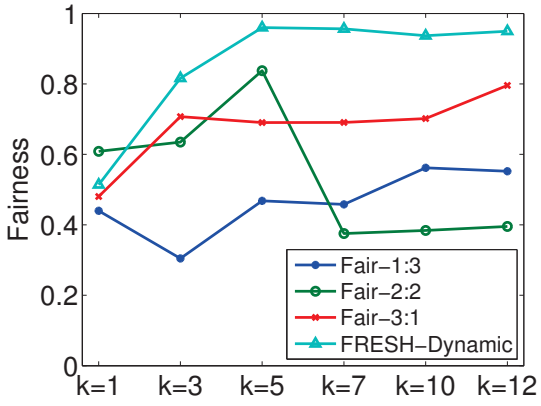

Figure 3.7: Fairness of set A with different values of k

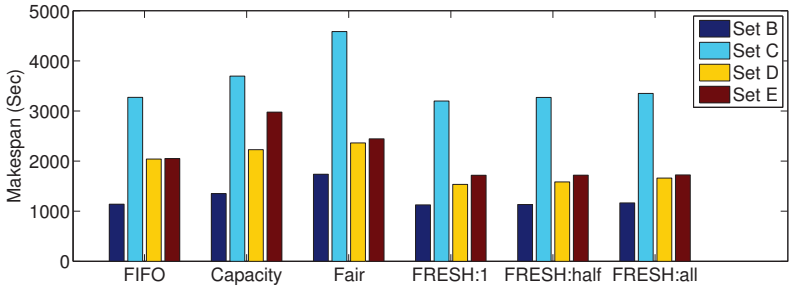

Figure 3.8: Makespan of set B∼E (with 10 slave nodes)

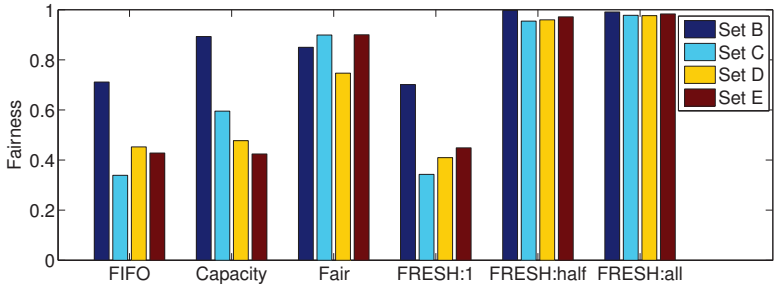

Figure 3.9: Fairness of set B∼E (with 10 slave nodes)

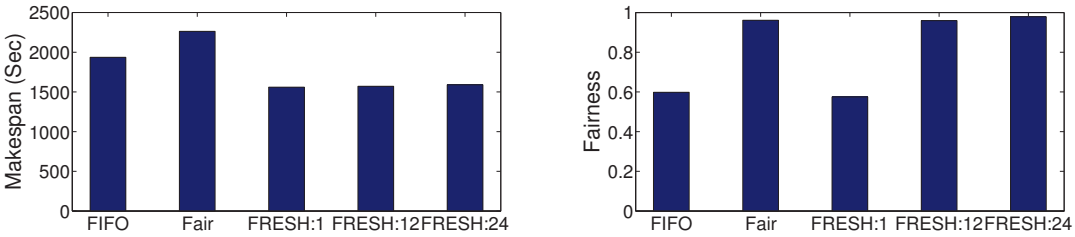

Figure 3.10: Makespan and fairness of set B with 20 slave nodes

CHAPTER 4

DEPENDENCY-BASED RESOURCE ALLOCATION

In this chapter, we investigate the dependency between two active consecutive stages of every job and aim to improve the efficiency of big data computing systems by optimizing the overlap between the consecutive stages. A new scheduling mechanism called OMO is developed to achieve this target. In OMO, we develop a new prediction module to estimate the execution time of each stage. Based on the prediction results, OMO dynamically adjusts the starting of the later stage of every job to achieve a good alignment of the two consecutive stages. We also consider Hadoop MapReduce as a representative and implement OMO as a plug-in scheduler in Hadoop. In the following chapter, we introduce the detailed motivation and solution of OMO. In the evaluation section, simulation results are shown to prove the validation of our prediction mechanism and experimental results are illustrated to show the performance improvement of OMO compared to default schedulers in Hadoop and our work FRESH.

4.1 Background and Motivation

This work 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 work 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 predication 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 slots 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.

In summary, the contributions of this work 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 develop a new technique, lazy start of reduce tasks, that estimates the execution time of the map phase and the shuffling step of the reduce phase, and derives the best time to start the reduce phase in order to minimize the gap from the end of the map phase to the end of the shuffling phase. (3) We develop a new technique, batch finish of map tasks, 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 on performance.

4.2 Problem Formulation

In this work, we consider a Hadoop cluster as the MapReduce service platform. Currently, there are two branches of Hadoop frameworks available, Hadoop [2] and Hadoop YARN [3]. 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 [3]. We will also compare the performance with Hadoop YARN in our evaluation (Section 4.4).

Table 4.1: 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	observed slot release frequency
A_o	observed # of *available slots*
F_e	estimated slot release frequency
A_e	estimated # of *available slots*
T_m	execution time of a map task
T_s	execution time of the shuffling phase
T_w	length of a historical window
R_t	# 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	size of data generated by one map task
B	network bandwidth

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 [37, 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 work 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 4.1 lists the notations we use in the rest of this work.

4.3 Our Solution : OMO

In this section, we present our solution eSplash 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 integrate both of them. The entire solution is mainly developed as a new Hadoop scheduler. The implementation details will be introduced in Section 4.4.

4.3.1 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 we adopt 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 a new concept of **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 scenario, we first need to determine the number of available slots (A_e) in that scenario. Then, based on the assumption that the slot releasing frequency is proportional to the number of available slots, we can calculate F_e as

$$F_e = \frac{F_o \cdot A_e}{A_o}. \tag{4.1}$$

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.

4.3.2 Lazy Start of Reduce Tasks

The goal of our first technique is to optimize the start time of the reduce phase of the 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 followed by the details of the algorithm.

4.3.2.1 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 the 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) generated 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.

Table 4.2 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 slots and 2 reduce slots. 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.

Apparently, setting slowstart to 1 yields the worst performance because all data shuffling happens after the map phase is finished. For the singe job execution, there

Table 4.2: 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

is no big difference in our experiments when slowstart is less than 1. The reduce slots have been reserved to serve reduce tasks and there is almost no disadvantages for starting reduce tasks early. When executing multiple jobs, however, early start of reduce tasks incurs performance penalty because the occupied reduce slots cannot serve reduce tasks of other jobs and the running reduce tasks will be idle most of time, waiting for more intermediate data from map tasks if started too early.

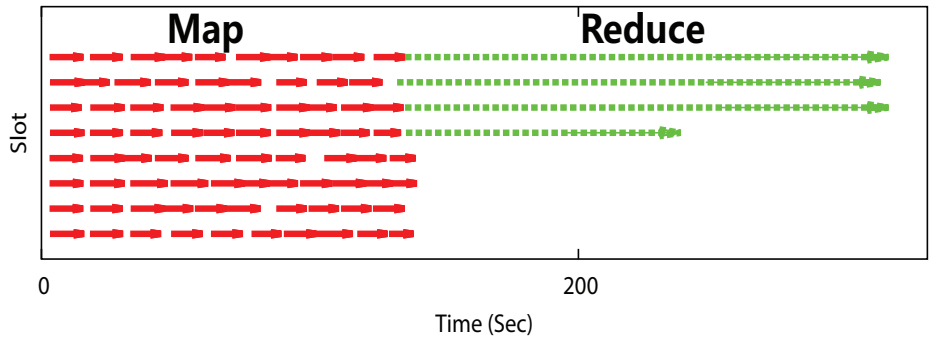

Figure 4.1: Slot allocation of one Terasort job with dynamic slot configuration (slowstart = 1).

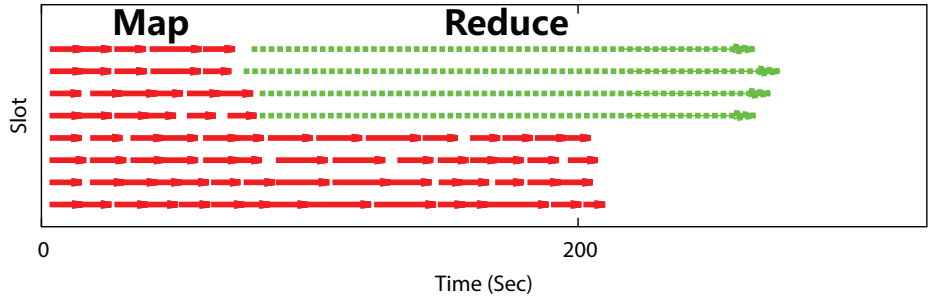

Figure 4.2: Slot allocation of one Terasort job with dynamic slot configuration (slowstart = 0.6).

In our solution with dynamic slot configuration, when to start reduce tasks of a job becomes more critical to the performance even for the single job execution. In our setting, all the pre-configured slots in the cluster serve as a resource pool, and they can be arbitrarily set as map or reduce slots during the execution of jobs. 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. We run the same experiments as the ones above with dynamic slot configuration and the execution time results are shown in Table 4.3. Fig. 4.1 and Fig. 4.2 additionally illustrates the slot allocation during the execution of one Terasort job with two different values of slowstart. We observe that setting the slowstart value too high or too low will both degrade the system performance. With both 1 and 3 jobs, the optimal value of slowstart in our tests is 0.6.

Table 4.3: Execution times of 1 and 3 Terasort jobs with different slowstart values and dynamic slot 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

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 work, we develop a new technical, *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.

4.3.2.2 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 information of 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.

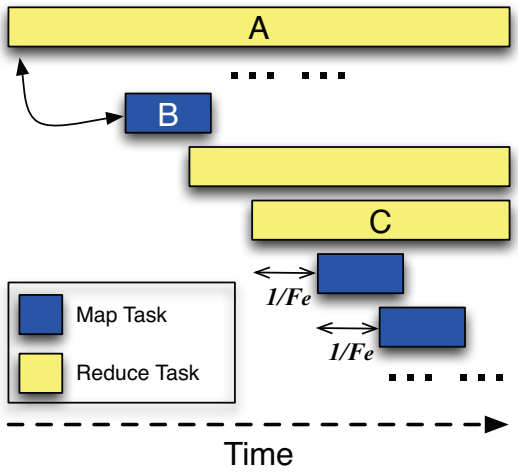

Figure 4.3: Illustration of the proof.

Principle 4.3.1 *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 can prove this principle by contradiction. Assume that the best arrange 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. 4.3, and the first reduce task, e.g., task A in Fig. 4.3. 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 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 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. 4.3. 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 eventually get a solution where reduce tasks are consecutively executed with no interruption of map tasks.

Based on the above principal, we analyze the gap from the ends of the map phase to 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. 4.4) 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.

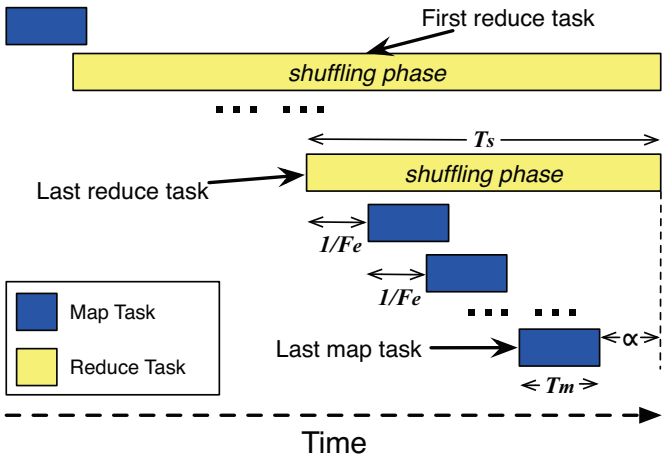

Figure 4.4: Lazy Start of Reduce Tasks: illustrating the alignment of map phase and shuffling phase.

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. 4.4). Therefore, we express α as a function of x:

$$\alpha = T_s - (\frac{x}{F_e} + T_m) = T_s - (\frac{x \cdot S}{F_o \cdot (S-r)} + T_m), \tag{4.2}$$

where F_o is a measured value as described in Section 4.3.2 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). Thus, 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:

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

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. (4.2), 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,

$$\alpha = T_s(i) - (\frac{x_i \cdot A_o \cdot K}{F_o \cdot (A_o - r_i)} + T_m(i)). \tag{4.3}$$

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 4.3.1. 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 7–21 enumerate all the running jobs that have not started their

reduce phases, and use Eq. (4.3) 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 22–27).

Algorithm 4.3.1: Function LazyStartReduce()

```
1:  K = 0, res = {}
2:  for i = 1 to n do
3:     if J_i is running then
4:        K ← K + 1
5:     end if
6:  end for
7:  for i = 1 to n do
8:     if J_i is running and has not started reduce phase then
9:        α_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:          α = T_s(i) − ((x_i · A_o · K) / (F_o · (A_o − r_i)) + T_m(i))
13:          if α < α_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:    return the index of the job in res with the least number of occupied slots
26: end if
```

4.3.3 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.

4.3.3.1 Motivations

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. Fig. 4.5 and Fig. 4.6 illustrate a simplified example of executing one MapReduce job. Assume that each map task can be finished in a time unit and each reduce task also needs a time unit to finish after its shuffling phase. Fig. 4.5 shows the execution process with 12 map tasks where the map phase is finished with 4 rounds and the total execution time is 5 time units. In Fig. 4.6, however, there is an additional map task causing an extra round in the map phase. The total execution time becomes 6 time units, i.e., a 20% increase compared to Fig. 4.5.

Fig. 4.7 shows an experiment with three jobs: terasort, wordcount and k-means in a Hadoop cluster with 4 slave nodes. Each node has 3 map slots and 1 reduce slot. 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 task for each job is 1. These three jobs are running with Fair scheduler and the slowstart

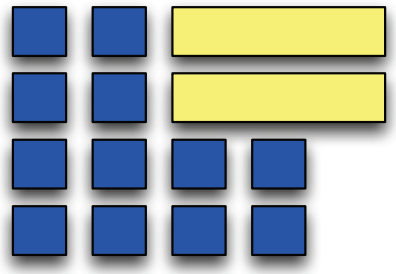

Figure 4.5: Map tasks are finished in 4 waves

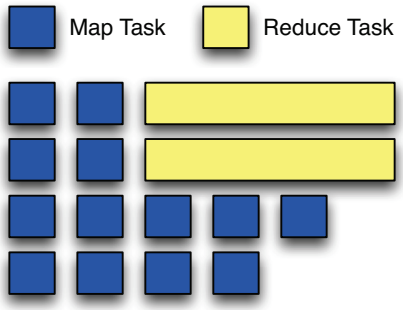

Figure 4.6: The tailing map task incurs an additional round

is set to 0.6. The X-axis is execution time and the Y-axis shows the task slots in the cluster. Slot 1 to 12 are map slots and 13 to 16 are reduce slots. Apparently, the map tasks of all three jobs are not well aligned after the first wave.

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.

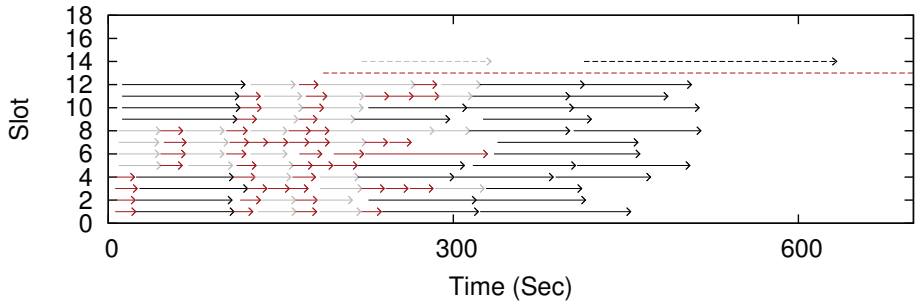

Figure 4.7: Experiment with 3 jobs in a Hadoop cluster with FAIR SCHEDULER: Solid lines represent map tasks and dashed lines represent reduce tasks.

4.3.3.2 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,

$$t_{fair} = \frac{K \cdot m'_i}{F_o} + T_m(i). \tag{4.4}$$

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

$$t_{batch} = \frac{s}{F_e} = \frac{s \cdot A_o}{F_o \cdot (A_o + r_i)}. \tag{4.5}$$

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 4.3.2. 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 7–14) 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 4.3.2: 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 ← K + 1
5:    end if
6: end for
7: for i = 1 to n do
8:    if J_i is running and has started reduce phase then
9:       Calculate t_fair and t_batch as in Eq. (4.4) and Eq. (4.5)
10:      if t_batch < t_fair and t_fair − t_batch > max then
11:         max = t_fair − t_batch, c = i
12:      end if
13:   end if
14: end for
15: return c
```

4.3.4 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.

Specifically, we integrate Algorithm 4.3.1, Algorithm 4.3.2, and the FAIR SCHEDULER in Algorithm 4.3.3. When a slot is released in the cluster, the algorithm first calls the function LazyStartReduce(). If it selects a job that should start its reduce phase, the released slot 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 slot 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 4.3.3: Slot Allocation

```
1: i = LazyStartReduce()
2: if i ≥ 0 then
3:     Allocate the released slot to J_i's reduce task
4: else
5:     i = BatchFinishMap();
6:     if i ≥ 0 then
7:         Allocate the released slot to J_i's map task
8:     else
9:         i = FairScheduler();
10:        Allocate the released slot to J_i
11:    end if
12: end if
```

4.4 Performance Evaluation

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

4.4.1 System Implementation

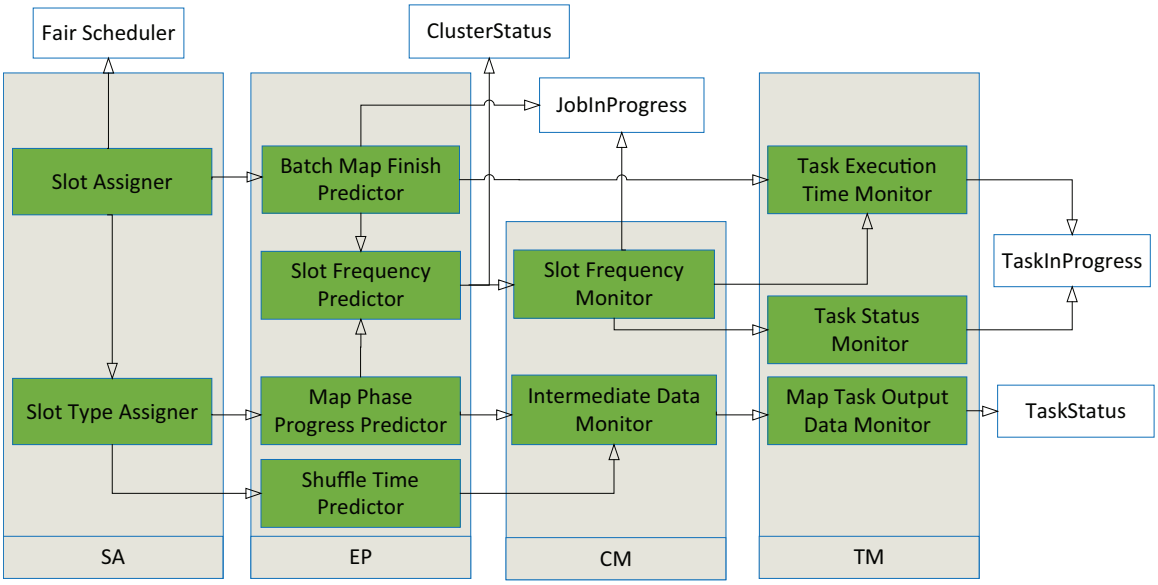

Figure 4.8: System implementation

We implemented our new scheduler OMO on Hadoop version 0.20.2 by adding a set of new components to support our solution. Fig 4.8 shows the details of the system implementation. The shadow parts are new modules created and other parts are existing modules in native Hadoop system and recalled by OMO. 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 by applying Algorithm 4.3.3 introduced in Section 4.3 with the information received from **EP**.

In addition, we have modified the fairness calculation in the traditional FAIR SCHEDULER, where the fairness of map slots and reduce slots are separately considered. Since we use dynamical slot configuration, a slot does not exclusively belong to either map or reduce slot category. Therefore, we consider the total number of the slots assigned to each job, and use it to calculate the *deficiency* for the FAIR SCHEDULER to make the scheduling decision.

4.4.2 Testbed Setup and Workloads

First, we introduce the cluster setting and the workloads for the evaluation.

4.4.2.1 Hadoop Cluster

All the experiments are conducted on NSF CloudLab computing platform at the University of Utah [39]. Each server has 8 ARMv8 cores at 2.4GHz, 64 GB ECC 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.

4.4.2.2 Workloads

Our workloads for evaluation consider general Hadoop benchmarks with large datasets as the input. In particular, we use six datasets in our experiments includ-

ing 10GB/20GB wiki category links data, 10GB/20GB Netflix movie rating data, and 10GB/20GB synthetic data. The wiki data includes the information about wiki page categories, 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 [26] to evaluate the performance.

- *Terasort*: Sort (key,value) tuples on the key with the synthetic data as input.
- *Sequence Count*: Count all unique sets of three consecutive words per document with a list of Wikipedia documents as input.
- *Word Count*: Count the occurrences of each word with a list of Wikipedia documents as input.
- *Inverted Index*: Generate word to document indexing with a list of Wikipedia documents as input.
- *Classification*: Classify the movies based on their ratings with the Netflix movie rating data as input.
- *Histogram Moives*: Generate a histogram of the number of movies in each user rating with the Netflix movie rating data as input.

Table 4.4 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 and reduce tasks.

Table 4.4: Benchmark characteristics

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

4.4.2.3 Validation of OMO Design

The design of OMO mainly relays on two new techniques: slot release rate prediction, and batch finish of the tailing map tasks. In this subsection, we present the experimental results that validate our design intuition.

Fig. 4.9 shows the slot release rate derived from an experiments with 12 mixed MapReduce jobs on a cluster of 20 nodes using Fair scheduler. We consider 10 seconds as a time window to derive the histograms of the slot releases. In addition, we apply the slot release rate estimation algorithm used in OMO and present the estimated value as the curve 'Estimation' in the following Fig. 4.9. Overall, we observe that our estimation of the slot release rate is close to the real value in the experiment. From the experimental trace, we find that the slot release rate in reality shows a high variance as we can see spikes in the curve. The estimation in OMO may not accurately predict the change when there is a big gap between two consecutive time windows. However, our algorithm usually catch up 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 complex workload is

a valid and feasible mechanism in practice. Later in this section, we will show the performance benefit we gain from this technique.

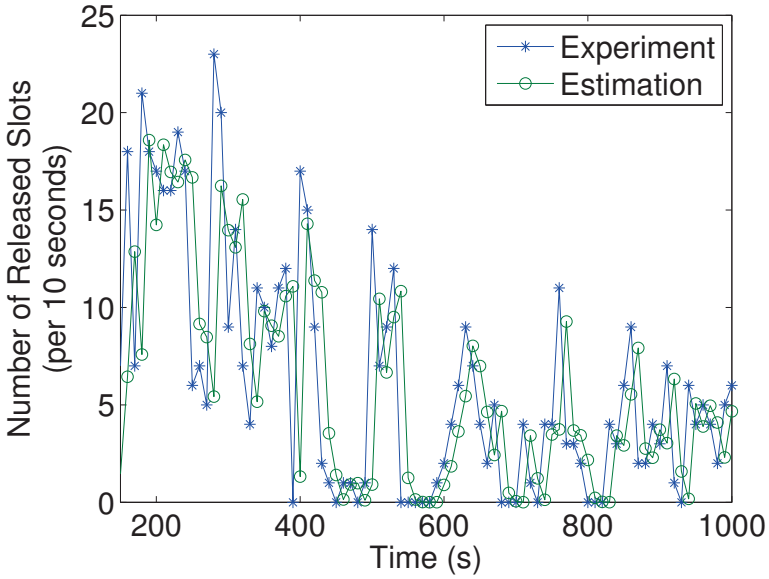

Figure 4.9: Slot release prediction

The other main technique in our solution is the batch finish of the map tasks. Our design mainly focus on the last batch of the map tasks. The following Fig. 4.10 compares OMO to Fair scheduler with a set of 12 mixed MapReduce jobs running on a 20-node cluster. We define *last batch* in a job as the last set of map tasks whose finish times are within 10 seconds. In Fig. 4.10, we observe that with Fair scheduler, the last batch of all the jobs contains no more than 10 tasks and half of the jobs 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 purge them quickly. In addition, there are cases where OMO yields even fewer number of map tasks in the last batch that Fair scheduler. It 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 from the experimental results. We will show how it helps improve the overall performance in the next subsection.

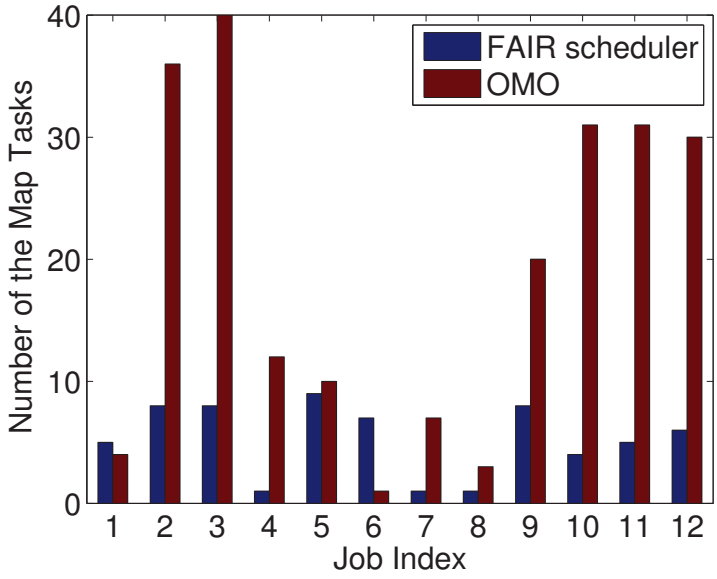

Figure 4.10: Last batch of map tasks

4.4.3 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

- FAIR SCHEDULER: We use the Hadoop's default slot configuration, i.e, each slave has 2 map slots and 2 reduce slots. 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: Our work FRESH also adopts dynamic slot configuration. The slowstart is set to 1.

In Summary, our results include following aspects:

- **Slot Allocation**: We illustrate the detailed slot 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.

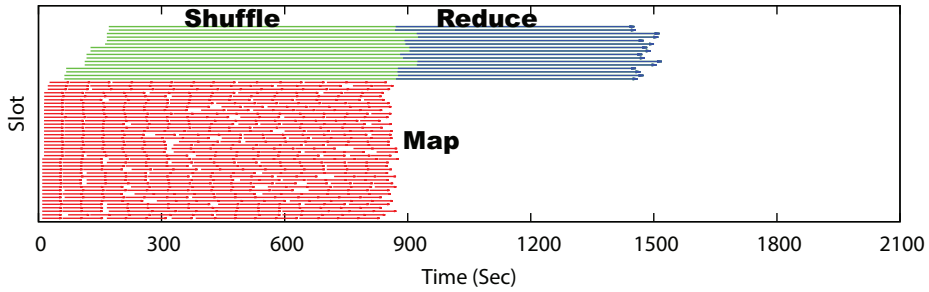

Figure 4.11: Slot allocation in the execution of 8 Terasort jobs with FAIR SCHEDULER and the default slowstart=0.05

4.4.3.1 Slot Allocation

First, we use TeraSort as an example to illustrate the slot allocation of OMO and other alternative schedulers in Hadoop (Fig. 4.11 - Fig. 4.14). In each test, we use 8 jobs with the Terasort benchmark. The input data size is 20 GB for each job. There are 160 GB data totally in each experiment. The X-axis is the execution time and

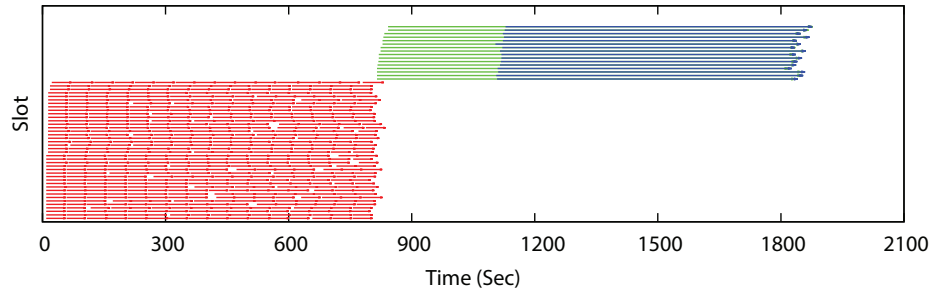

Figure 4.12: Slot allocation in the execution of 8 Terasort jobs with FAIR SCHEDULER and the default slowstart=1

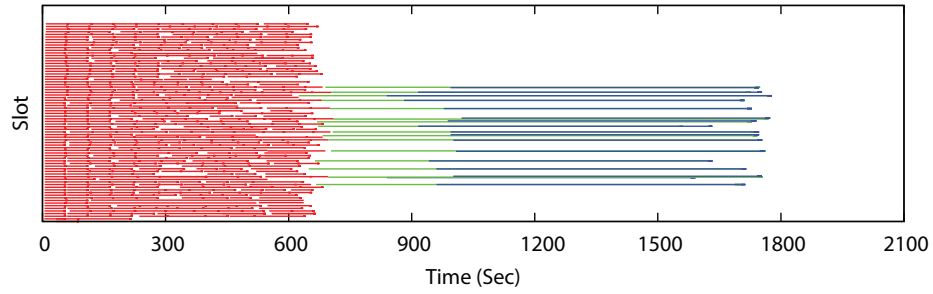

Figure 4.13: Slot allocation in the execution of 8 Terasort jobs with FRESH and the default slowstart=1

the Y-axis shows all the slots 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 span 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 slots 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*.

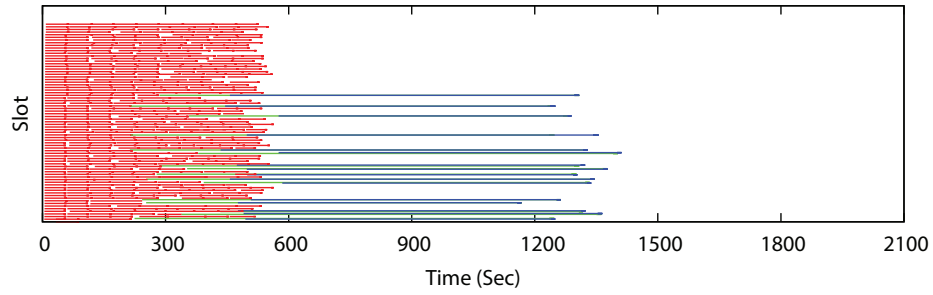

Figure 4.14: Slot allocation in the execution of 8 Terasort jobs with OMOand the default slowstart=1

4.4.3.2 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 eSplash, 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. 4.5. 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 [40]. 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 to H are for scalability experiments of eSplash.

Table 4.5: 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

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. 4.15 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.

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. 4.16 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*.

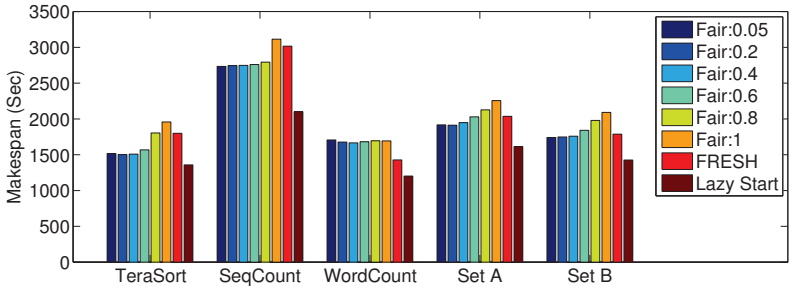

Figure 4.15: Execution time under FAIR SCHEDULER, *FRESH* and *Lazy Start* (with 20 slave nodes)

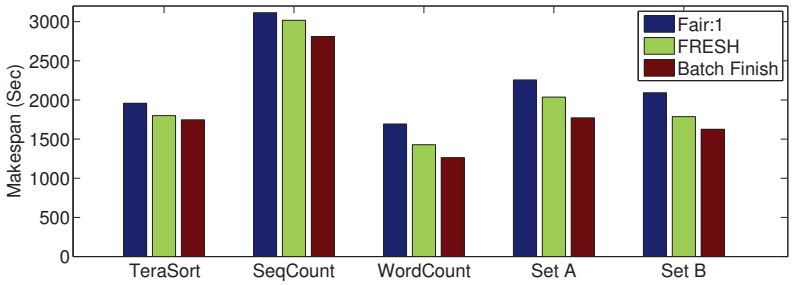

Figure 4.16: Execution time under FAIR SCHEDULER, *FRESH* and *Batch Finish* (with 20 slave nodes)

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.

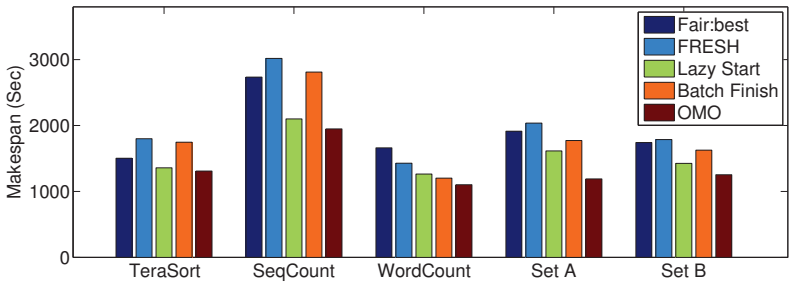

Figure 4.17: Execution time under FAIR SCHEDULER, *FRESH*, *Lazy Start*, *Batch Finish* and *OMO* (with 20 slave nodes)

The experiment results of three simple workloads and two sets of mixed workloads are shown in Fig. 4.17. *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. 4.18 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 phase and the shuffling phase are running concurrently and *Shuffle Only* represents the time span that the shuffling phase continues after the map phase has finished. By increasing map task slots 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*.

4.4.3.3 Comparison with YARN

In addition, Table. 4.6 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. 4.6. 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 4.6: Execution time of Terasort benchmark under YARN and OMO (with 20 slave nodes).

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

4.4.3.4 Scalability

Finally, we show the scalability of OMO with the experiments of different input data sizes, jobs 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. 4.19 (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.

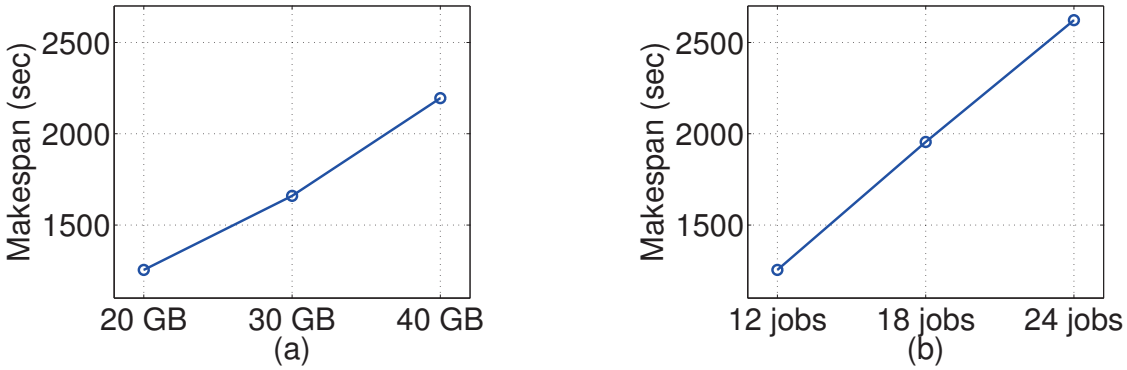

Figure 4.19: Execution time under *OMO* with: (a) different sizes of the input data and (b) different number of jobs

Then, we test the number of jobs scalability. we run the experiments with the same setting 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. 4.19 (b) shows the experiments results. The execution time in OMO grows linearly according to the number of jobs.

In the end, we test Set H on a large cluster with 40 slave nodes to show the scalability of OMO and the evaluation results are shown in Table 4.7. 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 4.7: 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.

4.5 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 [29] and Capacity Scheduler [28] are widely used to ensure each job to get a proper share of the available resources.

The Hadoop community released Next Generation MapReduce (YARN) [3]. 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. To improve the resource utilization in YARN, Haste [41] leverages the information of requested resources, resource capacities, and dependency

between tasks in the resource allocation. Opera [42] proposes a novel opportunistic and efficient resource allocation approach which breaks the barriers knowledge of actual runtime resource utilizations to re-assign opportunistic available resources to the pending tasks.

Some recent work [43, 44] proposed to use multiple scheduler to solve the scalability issue. Mesos [4] introduced a distributed two-level scheduling mechanism to share clusters and data efficiently between different platforms such as MapReduce, Dryad [45] and others. Our work can be integrated into these platforms as a single low-level scheduler.

4.6 Summary

This work 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. Two new techniques are included in OMO: lazy start of reduce tasks and batch finish of map tasks. Compared to the prior work, our solution considers more dynamic factors and predicts the resource availability when assigning the slots to jobs. 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.

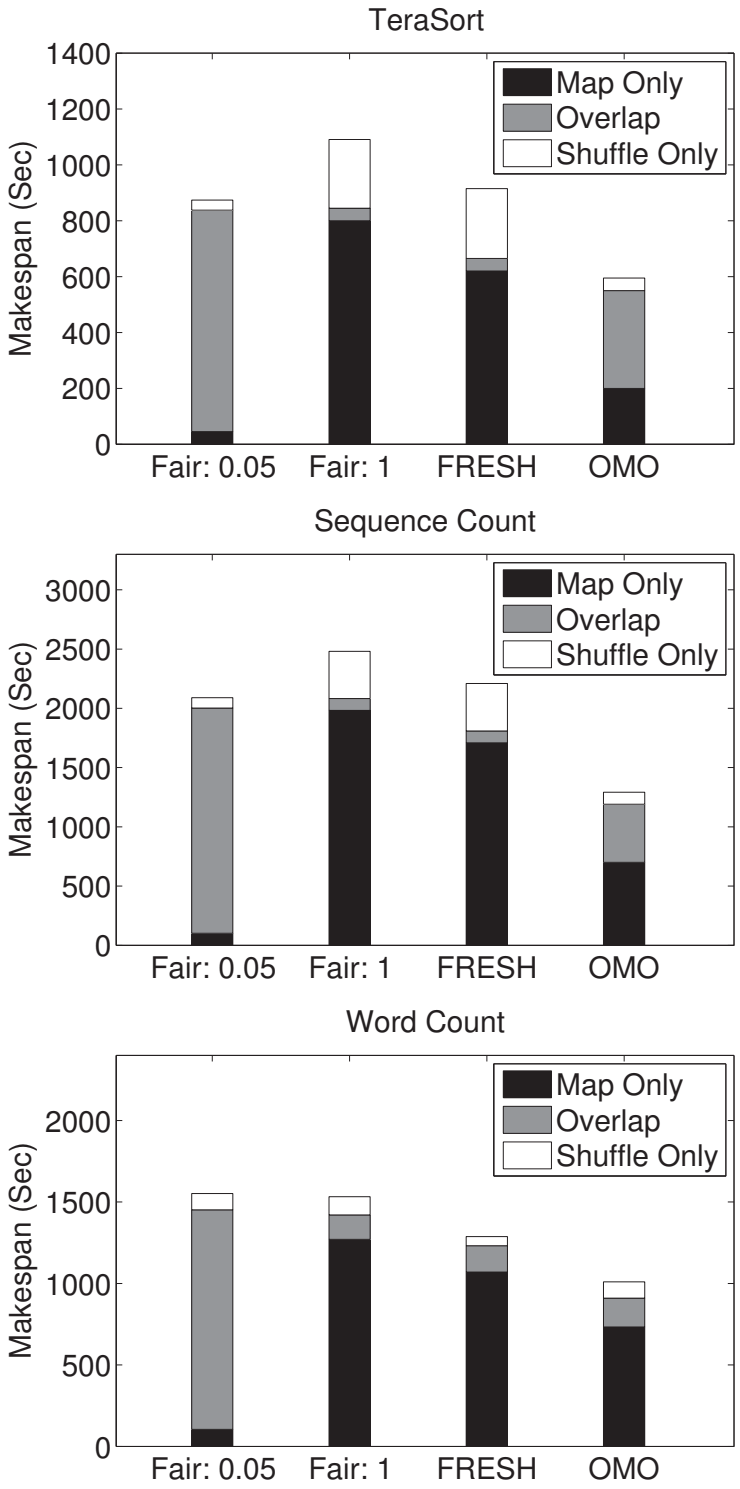

Figure 4.18: Execution time in map + shuffling phase of simple workloads.

CHAPTER 5

RESOURCE ALLOCATION WITH NODES FAILURE

In this chapter, we introduce our resource allocation schemes eSplash with node failures in the cluster. This work aims to mitigate the impact of node failures on the system performance by efficiently and effectively detecting stragglers and assigning speculative tasks in a heterogeneous cluster. Besides accurately and efficiently identifying stragglers, eSplash can assign speculative tasks to the appropriate nodes in order to improve the system performance. In the following part of this chapter, we elaborate our motivate and solution of eSplash. In addition, we compare eSplash with the latest default speculative mechanism in Hadoop YARN. From the experimental results, eSplash can distinctly reduce the increased makespan of batch jobs caused by the stragglers.

5.1 Background and Motivation

In any large-scale computing cluster, node failures are normality in practice. A usual omen is the straggling computing performance on the node. Speculative execution is a common and effective solution for mitigating the impact of node failures, e.g., speculation is a built-in component in Hadoop. Basically, once detecting a straggler node, the cluster will launch a redundant copy of the task running on the problematic node. Once either of them is finished, the other one will be killed. The intuition is to trade the resource efficiency with the reliability, especially if the delay of one task may further postpone the whole data processing. However, the traditional speculation does not work well in a heterogeneous cluster, which consists of nodes with different

hardware profiles. The heterogeneous setting has become a common environment in practice due to various reasons such as incremental hardware upgrade and diverse demands from different applications. Designing a speculation scheme in such a heterogeneous cluster, however, is challenging because it is very difficult to distinguish straggling nodes from naturally slow nodes.

In this work, we present eSplash, a Hadoop system with an efficient speculation scheme specifically designed for heterogeneous clusters. We identify the problems in the existing Hadoop system and develop the following major components: (1) Cluster all the nodes into different levels according to their computing performance; (2) Identify straggler nodes by monitoring the task's estimated finish time and progress rate; (3) Submit speculative request with parameters that guide the future execution. All the techniques presented in this work are implemented in Hadoop YARN system. We conduct extensive experiments for evaluation, and the results show that eSplash significantly improve the system performance.

5.1.1 Speculative Execution in Hadoop

Speculative execution is an important feature in a Hadoop system. It aims to identify the unstable slave nodes in the cluster and avoid the delay of the job execution caused by these nodes. In a large scale Hadoop system, each node's status and performance may not be consistent for a long-term process depending on a lot of hardware and software factors. It is possible that some nodes are prone to a failure, and their performance is degraded at the runtime. This laggard performance could be temporary, or eventually require restarting the Hadoop service or even a reboot. In the Hadoop system, the centralized ApplicationMaster monitors the execution of every task, and if it detects that a task is running slowly (more slowly than the other same type of tasks), it will start a redundant task, called speculative task, as an

alternative. When one of them is completed (either the original task or speculative task), the other task will be killed.

Particularly, Hadoop runs a background speculator service that maintains a statistics table to record all the execution times of the identical tasks, i.e., all the map or reduce tasks in a job. In other words, each MapReduce job has two entries in this statistics table, one for its map tasks and the other for its reduce task. The data in this table is updated upon the completion of each task. The speculator service will periodically check this table and the running tasks to find the candidate tasks for speculative execution. Specifically, it enumerates all the running tasks and estimate the finish time of each task t_i based on the elapsed time and the current progress as shown in Eq(5.1), where T_{now} is the current timestamp, $T_{start}(i)$ is the starting time of task t_i, and $PG(i)$ indicates t_i's current progress. In addition, the speculator service estimates the finish time of the alternative speculative execution in Eq(5.2. The execution time of the speculative task is estimated as the mean value of the historic execution times of the same type of tasks maintained in the statistic table. Fig. 5.1 shows an example of estimating Eq(5.1) and Eq(5.2) in the current Hadoop system. Apparently, if $EstEnd$ is greater than $EstRepEnd$, the task is expected to benefit from a speculative execution. When there are multiple candidate tasks for speculative execution, the speculator service will pick the one with the maximum value of $EstEnd - EstRepEnd$. Finally, the speculator service will create a new task attempt of the selected running task, and submit it to the pending task queue as a regular task.

$$EstEnd = \frac{T_{now} - T_{start}(i)}{PG(i)} + T_{start}(i) \tag{5.1}$$

$$EstRepEnd = mean(\ getTaskType(t_i)\ ) + T_{now} \tag{5.2}$$

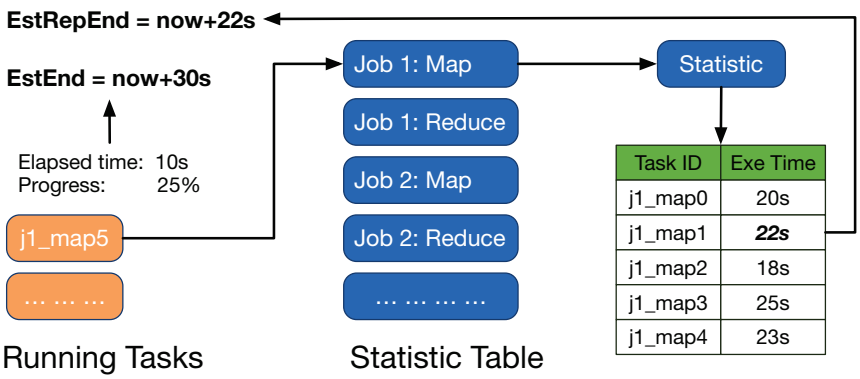

Figure 5.1: Hadoop records the historic execution times of each type of tasks in each job for determining the candidate tasks for speculative execution

5.1.2 Problems in a Heterogeneous System

The current speculative execution, however, is not effective in a heterogeneous Hadoop system, and could even lead to severe performance degradations. The main issue is that a heterogeneous cluster consists of 'slow' nodes and 'fast' nodes. The mean value of the execution times is no longer a good guideline to judge if a node is abnormally slow. In addition, it is difficult to estimate the execution time of each speculative task as it depends on what type of nodes the task will be running on. Specifically, because of the diverse processing performance across the cluster, there are the following two major problems for the speculative execution.

First, the decision of starting or not starting a speculative task for each running task may be wrong. The intuition of the current design is to detect the running tasks that are far behind the expected progress compared to other finished tasks of the same type. This intuition, however, does not hold in a heterogeneous system as a 'slow' node does need more time to finish a task than a 'fast' node. If the current statistical data are mainly from the same type of tasks finished on 'fast' nodes, the speculator service may consider the task running on a 'slow' node behind the schedule and start a redundant speculative task for it. However, this 'slow' node is behaving normally, and the scheduled speculative execution is unnecessary. On the other hand,

if a 'fast' node gets some problems and halts after executing a task for a while, the speculator service may consider its progress still in the normal range compared to the tasks finished on other nodes (especially on those 'slow' nodes). No speculative tasks will be created until this faulty 'fast' node has been hanging for a long time.

Second, the scheduled speculative tasks, when executed in the system, may not be as effective as we expect. The benefit of speculative execution is to mitigate the negative effects of problematic nodes in the system and avoid the delay caused by them. In a heterogeneous system, however, the execution time of the speculative task depends on the node that hosts its execution. If the task is assigned to a slower node, the execution time would be longer than the original task. What is even worse is that the speculative task might be assigned to the same node that hosts the original task because of the diverse resource capacities.

We conduct an experiment on a cluster of 4 nodes with identical hardware settings. However, when configuring the Hadoop service, we set the each node's capacity of vcores with different values as follows, slave1(32 vcores), slave2(16 vcores), slave3(8 vcores), slave4(4 vcores). As the number of physical cores in each node is fixed, the node set more vcores has worse performance for each vcore. In another words, for the tasks in the same type and from the same job, the node with more vcores configured takes more time to execute a task. In this case, slave1 can be consider as the slowest node.

In the experiment, we execute 5 MapReduce jobs each consisting of 38 map tasks and 10 reduce tasks. The statistics of the speculative execution is listed in the following Table 5.1. Out of the total of 240 original tasks, the speculator service has generated the redundant execution for 62 of them. All these 62 original tasks are initially assigned to the slowest node slave1. Among all the 62 speculative tasks, 25 of them are assigned back to slave1 and all of them get killed in the end.

	Task Type	slave1	slave2	slave3	slave4	Total
Finished	Map	0	19	5	5	28
	Reduce	0	0	1	0	1
Killed	Map	25	7	1	0	33
	Reduce	0	0	0	0	0

Table 5.1: Speculative executions in an experiment

Above all, we aim to develop an efficient speculation scheme eSplash for a heterogeneous cluster which can accurately and quickly detect straggler nodes, effectively avoid unnecessary speculative execution, submit speculative tasks to the most appropriate nodes.

5.2 Our Solution: eSplash

In this section, we present the details of the design of eSplash that aims to efficiently manage the speculation execution in a large scale heterogeneous computing system. It mainly includes the three components:

- **Classify cluster nodes**: To accommodate the heterogeneous environment, our solution classifies the cluster nodes into different groups depending on their computing capabilities. A centralized manager maintains the run-time performance statistics for each individual group. These per-group data will serve other components such as detecting straggler nodes and submitting speculative tasks. The classification of the nodes can be pre-configured by the administrator, or dynamically determined based on run-time performance.

- **Detect straggler nodes**: The straggler nodes are the 'slow' nodes compared to other nodes in the same group, and could be prone to a failure. The tasks running on a straggler node are candidates for speculative execution. In eSplash, we develop a scheme that accurately and quickly detects straggler nodes in a large cluster.

- **Submit speculative tasks**: This is the most important component in eSplash. Basically, we need to determine whether a speculative task is worthwhile. The decision is based on the comparison of the estimated complete time of the current task (running on a straggler node), and the estimated execution time of a new speculative task. However, it is difficult to achieve an accurate estimation in practice, and it is more challenging in a heterogeneous cluster because the execution time of the speculative task depends on the computing capability of the hosting node. Our design in this component considers the practical factors, derives accurate estimation, and provides associated parameters for each speculative task for its future execution.

5.2.1 Classify Cluster Nodes

In order to effectively identify the straggler node and launch speculative tasks, the cluster manager has to compare a node's run-time performance to other nodes with similar hardware that are executing the same task.

In our design, each node in the cluster is associated with a *level* indicating its computation performance. Specifically, we classify all the nodes into multiple groups according to their performance, and the level value is the index number the group the node belongs to. We define that a higher level value represents a stronger computation ability. In other words, a level i node will finish a task faster than a level j node for $i > j$.

The node classification can be pre-configured by the cluster manager based on each node's hardware profile, or dynamically adjusted based the nodes' run-time performance. In this subsection, we focus on the dynamic classification algorithm at the run-time.

Performance vector: In our design, the cluster master maintains a *performance vector* PV_i for each node i,

$$PV_i = \{e_1, e_2, \ldots, e_D\},$$

where D is the number of distinct types of tasks node i has finished, and each value in the vector is the execution time of each type of tasks. For example, if there are five concurrent MapReduce jobs running in the cluster with the Fair scheduler, after finishing at least one map tasks from each job, every node will have a performance vector of five values. If more than one tasks have been finished for a particular type of tasks, then the average execution time will be filled in the PV. Then we will cluster all the nodes based on their PVs.

Clustering algorithm: We consider each node's performance represented by PV_i is a data point in a D-dimensional space, and our problem becomes similar to the traditional clustering problem such as k-mean. However, in our setting, the resulting clusters(levels) indicate the performance and require a lexicographical order of the performance vectors. A node in a higher level is supposed to dominate any other nodes in lower levels for any type of tasks. Therefore, we present a new clustering algorithm based on the traditional k-mean algorithm.

Our solution consists of a grouping algorithm and a group-based k-mean clustering algorithm. The goal of the grouping algorithm is to merge individual PV data points into a set of groups that satisfy the lexicographic order. Then in our group-based k-mean algorithm, each group is the smallest unit to be assigned to a cluster, i.e., all the data points in a group will always stay in the same cluster.

The details are presented in Algorithm 5.2.1 and Algorithm 5.2.2. In Algorithm 5.2.1, we start with each data point as a group (line 1). Then the algorithm tries to merge the groups to enforce the lexicographic order. For each group g_i, we keep track of the minimum and maximum values of each dimension, recorded in min_i and max_i (lines 3–6). Then the algorithm compares every pair of groups, g_i and g_j, and merge them if they do not dominate each other. In line 8, we present the

condition for the merging operation. If there exist two dimensions, where each of the two groups performs better in one of them, then we have to merge these two groups. After forming a new group, we need to merge other groups that overlap with the new group ($[min, max]$ overlapping in any dimension), and update the min_i and max_i (lines 10–11). The resulting groups are non-overlapping, and keep the lexicographic order between any two of them.

Algorithm 5.2.1: Grouping Algorithm

1: **Initial grouping:** $\forall i, g_i = \{PV_i\}$
2: **Merging groups:** form final groups based the lexicographical order
3: **for** g_i **do**
4: $min_i(k) = min\{e_k \in PV_l | PV_l \in g_i\}$, $\forall k \in [1, D]$
5: $max_i(k) = max\{e_k \in PV_l | PV_l \in g_i\}$, $\forall k \in [1, D]$
6: **end for**
7: **for** any g_i and g_j **do**
8: **if** $\exists a, b$, $min_i(a) > max_j(a)$ and $min_j(b) > max_i(b)$
9: **then** merge g_i and g_j, $g_i \rightarrow g_i \cup g_j$
10: Merge all other overlapping groups into g_i
11: Update min_i and max_i
12: **end for**

Based on the result of the grouping algorithm, we develop the following clustering algorithm. The basic steps are similar to the traditional k-mean algorithm. However, after each data point calculates the distance to each cluster center (line 2), it does not select the closest cluster center to join. In our algorithm, the decision has to be made by the whole group, not each individual data point. In particular, we adopt a voting scheme in line 3, by counting the preferred cluster center of every group member. The most popular cluster center will become the group preferred cluster center. Then all the data points in the group will be assigned to that cluster center. The algorithm repeats this iterative process until there is no reassignment.

The following Fig. 5.2 shows a comparison of traditional clustering algorithm and our group-based clustering algorithm. The ground truth is that we configured 4 types of nodes, each with 20 nodes. We conduct experiments with two jobs, WordCount

Algorithm 5.2.2: Group-based k-mean Clustering Algorithm

1: Randomly select k cluster centers
2: For each PV_i, calculate the distance to each center, and pick the closest one as the preferred center
3: For each group g_i, check the preferred center selected by each member PV, and pick the center with the most votes as the group preferred one
4: Assign all the PV points in a group to the group preferred center
5: Recalculate the cluster centers and repeat the process until no PV/group is reassigned

and TeraSort, and measure the execution time of each job's map tasks on every node. Apparently, our algorithm accurately captures the pre-defined levels while the clustering result from the traditional k-mean is not feasible in our problem setting.

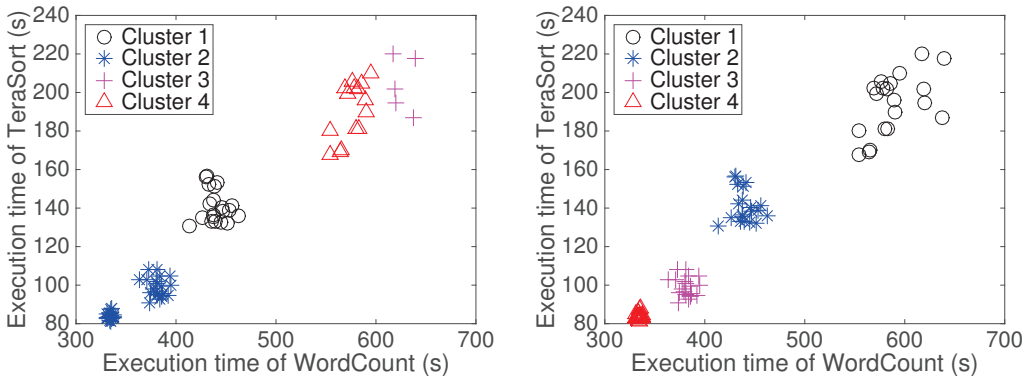

Figure 5.2: An example of clustering 80 nodes: we collect the execution time of two types of tasks (the map tasks in WordCount and TeraSort), thus each PV is a two dimensional data.

5.2.2 Detect Straggler Nodes

The traditional speculation scheme makes the decision based on per-task performance and is not suitable for a heterogeneous system because of the naturally varying performance across the cluster. In our design, detecting straggler nodes is the first step for speculative execution. Only the tasks on a straggler node are candidates for speculation.

Performance Statistics Table: With every node associated with a level value, eSplash maintains a per-level performance statistic table (ST), and uses the data in this table to detect the straggler nodes in the system. This table consists of $L \times D$ cells, where L is the number of levels in the system and D is the number of active task types. Each cell $ST(i, j)$ represents the performance statistics of task type j at a level i node,

$$ST(i, j) =< \mu \text{ (mean)}, \delta \text{ (variance)}, PR \text{ (progress rate)} >,$$

where μ and δ are regular statistics for the task execution time, and PR records the average progress increase of this type of tasks in the past epoch. The table data is updated once the master node receives the heartbeat messages from the slave nodes.
Straggler Value: Based on the information in table ST, eSplash detects the abnormally slow node by comparing the task performance on the node with the statistic data for the level it belongs to. In particular, we assign each node a *straggler value* (SV) to indicate how likely the node is a straggler. Once the value exceeds a threshold τ, the node is marked as a straggler. The straggler value of a node is updated with the task performance on the node, and the following two aspects are included:

- **Estimated execution time:** For each running task on the node, we estimate its execution time (denoted bt $EstT$) by dividing the elapsed time by the task progress. Then, we compare it with the mean and variance values stored in table ST. Assume the node is in level i, the following formula is applied to update SV,

$$SV \leftarrow SV + \sum_{\text{running task } j} \frac{EstT - \mu(i, type(j))}{\delta(i, type(j))},$$

  where $type(j)$ returns the task type index of j and we exclude the running tasks whose $EstT$ values are smaller than the recorded mean values.

- **Progress rate:** While the estimated execution time is a good indicator for the performance, sometimes it takes a relatively long time for the system to detect a straggler node. For example, if a node normally executes a task and gets stuck when the task is almost finished, its estimated execution time will stay in the normal range for quite a long time. Therefore, we include the second metric, progress rate, to help quickly identify a straggler node. Let PR_j represents the progress rate of task j running on the node, we use the following formula to update SV,

$$SV \leftarrow SV + \sum_{\text{running task } j} \frac{PR(i, type(j))}{PR_j},$$

Note that there are other approaches to aggregate these two metrics to calculate SV, and their weights can be adjusted with coefficient parameters. The current design in eSplash is an empirical setting and out intuition is to pay more attention on the deficit of the progress rate.

5.2.3 Submit Speculative Tasks

Once straggler nodes are identified, all the active tasks running on those nodes are candidates for speculative execution. The goal of this module is to select one candidate task and submit a speculative task for it. In the traditional speculation scheme in Hadoop, we need to estimate the finish time of the speculative task and compare to the currently running task to determine if it is worthwhile. In a heterogeneous system, however, the finish time of the speculative task depends on which (level of) node will host the execution. For example, given a candidate task, it is possible that running a speculative task on a high level node will be faster, but running it on a low level node will be even slower than the current task. Therefore, when making the decision for speculative execution, we have to consider the level of the prospective hosting node of the speculative task. In addition, when submitting the speculation request, the level constraint should be specified.

In eSplash, we require every speculation request to be associated with a value of the minimum level ($minL$) to execute the speculative task. Only the nodes in the minimum level or higher level are eligible to host the speculative task. The following Algorithm 5.2.3 is developed in eSplash to determine the value of $minL$. When examining an active task running on a node at level i, the possible values for

Algorithm 5.2.3: Determine $minL$ for a speculation request

1: Given a task running on a straggler node at level i
2: **for** $l = i$ to HL **do**
3: $\quad ExpT_l \leftarrow \sum_{m \in [l, HL]} Pr(m) \cdot EstT(m)$
4: **end for**
5: $minL = min\{ExpT_l, \forall l \in [i, HL]\}$

$minL$ range from i to HL which represents the highest level in the system. In the algorithm, we enumerate all the possible values, and then decide the best choice. Assume the $minL$ is set to be l, the algorithm calculates an expected execution time of the speculative task in line 3. The speculative task could be executed on a node at level l or above. We use $EstT(m)$ to indicate the execution time if the speculative task is hosted on a level m node, and $Pr(m)$ is probability of this case. The value of $EstT(m)$ can be set as the mean value in the performance statistic table, and $Pr(m)$ is derived as follows:

$$Pr(m) = \frac{C(m)}{\sum_{j \in [l, HL]} C(j)},$$

where $C(j)$ is the number of containers on all the nodes at level j that can serve this type of task. The value of $C(j)$ can be pre-computed according to the resource capacity on all the level j nodes and the resource demands of the task. Eventually, in line 5, $minL$ is set to the value that yields the minimum expected execution time.

Finally, after every candidate task derives a $minL$ value with its speculative request, we need to pick one candidate and submit its request. Following the Hadoop workflow, this process will be repeated periodically, but every time only one speculative request can be submitted. We inherit the Hadoop's design, and use a *speculative*

value to indicate the priority of each candidate task. However, this speculative value is re-defined as follows:

$$\frac{\sum_{m \in [minL, HL]} C(m) \cdot (EstEnd - T_{now} - EstT(m))}{\sum_{m \in [1, HL]} C(m)},$$

where $EstEnd$ is the estimated finish time of the original task, and T_{now} is the current timestamp. Essentially, this speculative value is the expected benefit (execution time reduction) the task can obtain. Therefore, the task with the highest speculative value will be selected for speculation execution.

5.2.4 Other Enhancements when Executing Speculative Tasks

In eSplash, we developed a couple of other enhancements to improve the performance. First, the marked straggler nodes are excluded from hosting any speculative task. Second, if a speculative task waits for a certain amount of time in the queue of the pending task, we re-evaluate its estimated finish time and compare to the original task to see if it is still worth a speculative execution. Due to the page limit, the details are omitted in this work, but these enhancements are also evaluated in our experiments.

5.3 Performance Evaluation

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

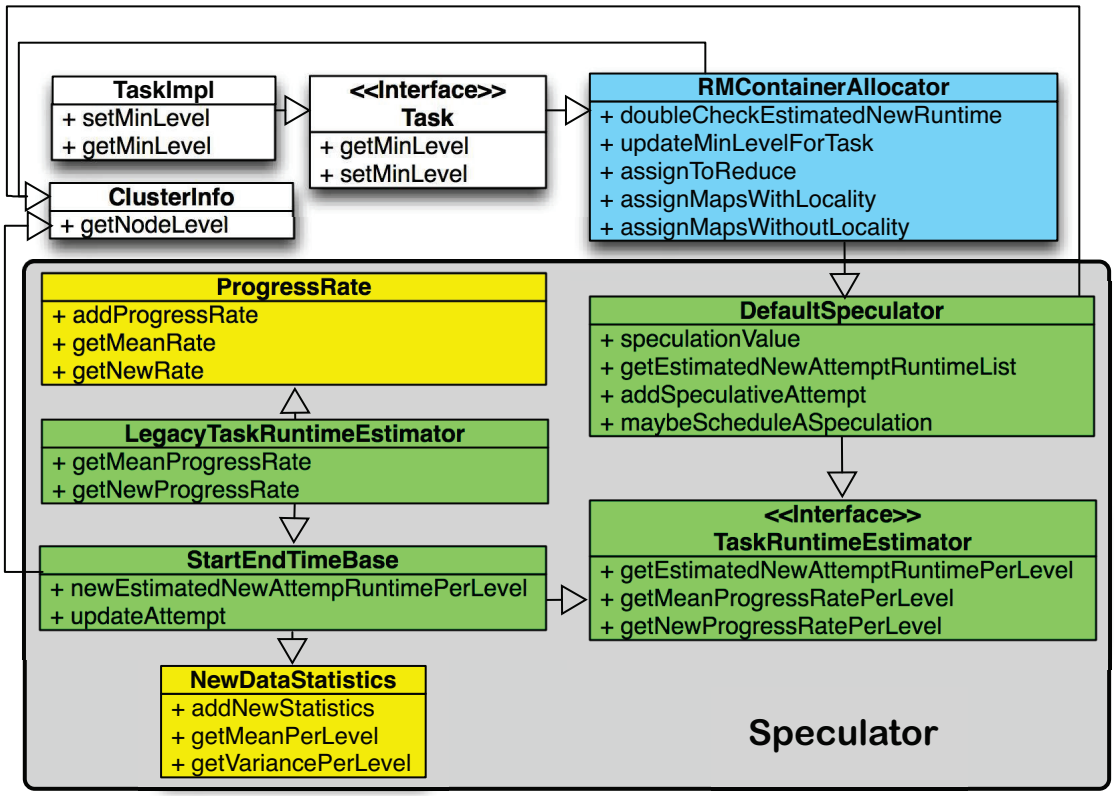

Figure 5.4: System implementation

To support our solution, we implemented our new scheduler ESPLASH on Hadoop YARN version 2.7.1 by creating a new **Speculator** component and modifying **RMContainerAllocator** component (Container Allocator) in *MRAppMaster* (MapReduce Application Master). Fig. 5.4 shows the details of the system implementation.

5.3.1 System Implementation

First, to determine the speculative tasks, we create the new *Speculator* component. Its architecture follows *Speculate* component in native Hadoop. In *Speculator*, all modules in green exist in native Hadoop and we create new methods in them. And the modules in yellow are newly created by us. *NewDataStatistics* statistics the average execution time of each completed map/reduce task on each node level. According to such statistics, *StartEndTimeBase* estimates the execution time of every

running task on the slave nodes with different levels. In addition, the new component *ProgressRate* monitors the progress variety of every running task in real time. *LegacyTaskRuntimeEstimator* is the subclass of *StartEndTimeBase* and it collects the information of the progress rate per running task from *ProgressRate*. *TaskRuntimeEstimator* is the interface for *DefaultSpeculator* to get all statistics above. Based on the estimated execution time and progress rate of every running task, *DefaultSpeculator* firstly determines the candidate tasks for speculation execution. Then it quantifies the speculative value of each candidate task and marks the minimum node level where the speculation can be executed. Finally, it selects the candidate task with the highest speculative value and creates a speculative task for it.

Second, to allocate the appropriate containers for the speculative tasks, we modify the *RMContainerAllocator* component. Firstly, it re-calculates the estimated execution time of the original tasks with pending speculative tasks to check whether it is still worthy to execute these speculative tasks. Secondly, it removes the unnecessary pending tasks and updates their minimum node levels. In the end, it checks the node levels that all available containers belong to and assigns the most appropriate container to each pending speculative task.

In addition, we modified *ClusterInfo* to set the node level for each slave node and the *TaskImpl*/*Task* in *job* to mark the minimum node level that every task can be executed on.

5.3.2 Testbed Setup and Workloads

All the experiments are conducted on NSF CloudLab platform at the University of Utah [39]. In each server, there are 8 ARMv8 cores at 2.4GHz, 64 GB memory and 120 GB storage. We launched a cluster with 9 servers: 1 master node and 8 slave nodes. We create 4 node levels and assign two slave nodes in each level. To create the heterogeneous environment, we classify node levels by specifying different

capacities of servers in different levels. Specifically, we configure 32 vcores in each slave node of level 1, 16 vcores of level 2, 8 vcores of level 3 and 4 vcores of level 4. As the number of physical cores is fixed in each server, the one configured by more vcores has worse performance for each vcore. So the slave nodes in level 4 achieve the best performance in executing a map/reduce task and the nodes in level 1 present the worst performance.

Our workloads for evaluation consider general Hadoop benchmarks with large datasets as the input. In particular, we use two datasets in our experiments including 20 GB wiki category links data and 20 GB synthetic data. The wiki data includes wiki page categories information, and the synthetic data is generated by the tool TeraGen in Hadoop. We choose the following four Hadoop benchmarks from Hadoop examples library to evaluate the performance: (1) *Terasort*: Sort (key,value) tuples on the key with the synthetic data as input. (2) *Word Count*: Count the occurrences of each word with a list of Wikipedia documents as input. (3) *Grep*: Take a list of Wikipedia documents as input and search for a pattern in the files. (4) *Wordmean*: Count the average length of the words with a list of Wikipedia documents as input.

5.3.3 Performance Evaluation

Given a batch of MapReduce jobs, our performance metrics are the increased makespan with stragglers and the accumulated wasted time of killed speculative tasks. We mainly compare eSplash to the native speculation scheduler in YARN (*LATE* [35]) and the one with speculation disabled (*Non-specu*). We have conducted two categories of tests with different workloads: *simple workloads* consist of the same type of jobs and *mixed workloads* represent a set of hybrid jobs. For each test of simple workloads, we generate 8 jobs of the same benchmarks. For testing mixed workloads, we mix all four benchmarks above and generate 2 jobs for each benchmark. For each job of both simple and mixed workloads, the input data is 20 GB. There are 80 map

tasks and 10 reduce tasks created by each job and each task requires 1 vcore and 2 GB memory. In the rest of this subsection, we separately present the evaluation results in the heterogeneous environment: (1) without stragglers, (2) with stragglers which can be recovered, and (3) with stragglers which cannot be recovered.

5.3.3.1 Performance without Stragglers

For our first experiment, we test both single and mixed workloads in the heterogeneous cluster without any stragglers. The first graph of Fig. 5.3 shows the number of speculative tasks created during the experiments. The black parts represent all killed speculative tasks and the white parts show all successful ones. According to the principle of speculation execution, speculative tasks are killed because of their original tasks are finished earlier than the speculative ones. Ideally, there should be no speculative tasks created during the experiments. However, under *LATE*, there are 28 speculative tasks averagely created in the experiments of simple workloads and 62 ones in mixed workloads. The original tasks of such speculative ones are all from the slave nodes on node level 1. As there is no mechanism in assigning speculative tasks to appropriate slave nodes in *LATE*, on average, 64.8% of the speculative tasks are assigned back to the 'slow' nodes on node level 1 and killed when their original tasks are finished. We notice that there are still about 5 speculative tasks created in each experiment under eSplash. After tracing the logs of such tasks, we found the reason. In eSplash, we revoke the function in native YARN to report the estimated execution time of each running task. In very low chance, it may report one extreme high value and eSplash mistakenly create a speculative task based on such value. We will try to fix this issue in the future work. But still, eSplash reduces 80% - 91.9% unnecessary speculative tasks over *LATE*. In addition, the average accumulated execution time of killed speculative tasks in each experiment is 3536 seconds in

LATE and 343 seconds in eSplash. eSplash decreased 90.3% wasted time cost in killed speculative tasks over *LATE*.

Fig. 5.5 shows the makespan performance of eSplash, *LATE* and *Non-specu*. Although 64.8% speculative tasks are killed in *LATE*, the remaining successful speculative tasks help to speed up the finish of all jobs. There is no significant difference in makespan between *LATE* and *Non-specu*. On average, eSplash improves 5.89% and 4.58% of the performance on makespan compared to *Non-specu* and *LATE*.

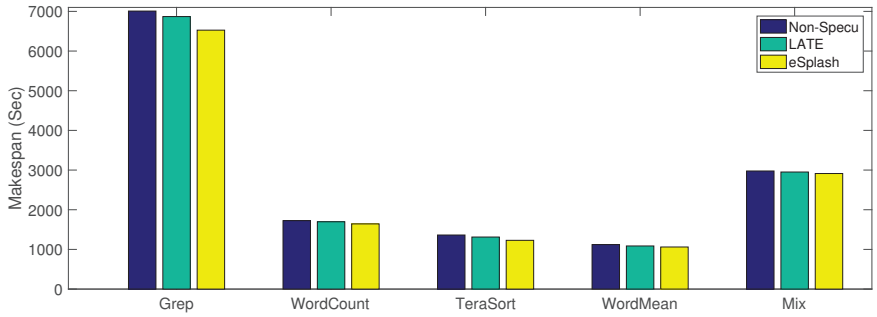

Figure 5.5: Makespan without stragglers

5.3.3.2 Performance with Stragglers Which Can Be Recovered

To evaluate the speculative execution with stragglers in the cluster, we manually slow down a slave node by running four CPU-intensive processes (the factorial of the integer 10,000) and four disk-intensive processes (*dd* tasks writing large files in a loop). Such processes last 1000 seconds during jobs execution and then the straggler will be recovered to normal performance. We run experiments separately with the straggler in 'slow' node (Level 1) and in 'fast' node (Level 4). The second and third graph of Fig. 5.3 shows the statistics of speculative tasks created during the experiments with a straggler on node level 1 and on node level 4. *Killed in Straggler* represents the killed speculative tasks which are assigned to the straggler node, *Killed in slow node* represents the killed speculative tasks which are assigned to the slave nodes in the same or lower node level compared to their original tasks. From the test results,

all killed speculative tasks in *LATE* are either assigned to the straggler itself or to a slower slave node. While triggering a straggler in a slow node (on node level 1), 61.9% averagely of speculative tasks are killed in *LATE*. Among all these killed speculative tasks, 50.4% of them are assigned back to the straggler. When the straggler is a fast node (on node level 4), on average 87.1% of speculative tasks are killed in *LATE* and 47.7% of the killed ones are assigned to the straggler. Meantime, there is no speculative task assigned to the straggler or the slower slave nodes in eSplash.

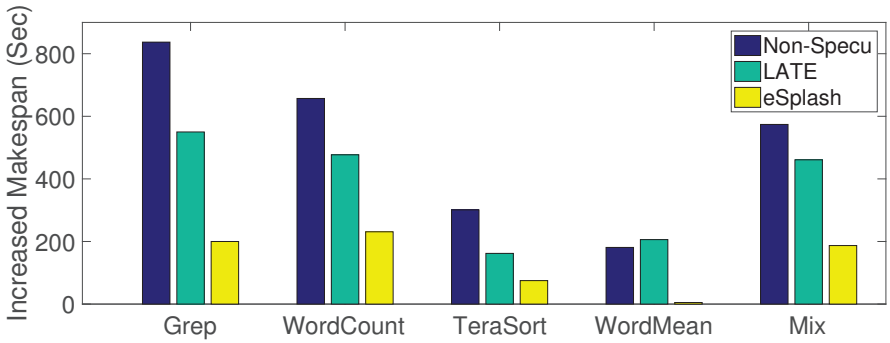

Figure 5.6: Increased makespan with a straggler on node level 1

Fig. 5.6 and Fig. 5.7 shows the increased makespan in both experiments. Generally, since *Non-specu* cannot address the situation with stragglers in the cluster, it represents the worst performance in the increased makespan. However, in the test with the straggler in the 'fast' node, as *LATE* assigned 13 out of 19 speculative tasks to the slave nodes on node level 1, the increased makespan of *LATE* is even 47.9% more than the one in *Non-specu*. eSplash improves the performance of the increased makespan significantly. With the straggler in the 'slow' node, averagely, eSplash improves the increased makespan by 76.7% and 65.7% compared to *Non-specu* and *LATE*. With the straggler in the 'fast' node, on average, the increased makespan under eSplash is 69.4% and 66.7% shorter than the ones under *Non-specu* and *LATE*. From the test results, *LATE* cannot efficiently deal with the stragglers in the heterogeneous cluster.

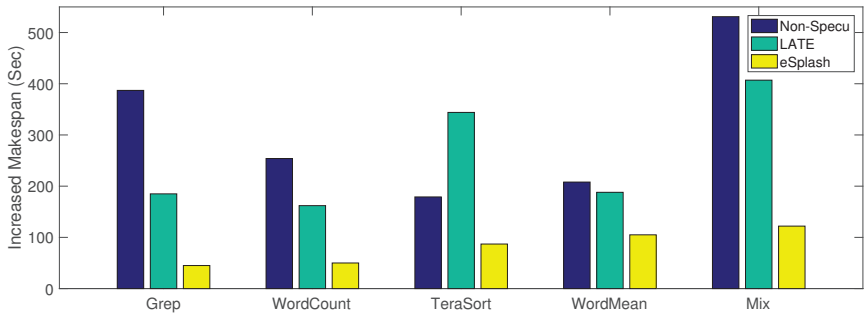

Figure 5.7: Increased makespan with a straggler on node level 4

5.3.3.3 Performance with Stragglers Which Cannot Be Recovered

In practice, abnormally slow execution of a server is a sign of a system failure. Rebooting the whole system is a common operation to handle such failure. So we design an experiment to check whether speculation executions can deal with this issue. In the experiment, we set a slave node in the node level 4 to be a straggler and slow down it by the same processes above. After running these processes for 1000 seconds, we manually shut down the processes of NodeManager and DataNode of the slave node and restart them after 600 seconds (to simulate the rebooting of the straggler node). We run the same experiment under *Non-specu*, *LATE* and eSplash. Fig. 5.8 illustrates the increased makespan under each mechanism. As the straggler is considered as normal node and speculative tasks from other 'slow' nodes are assigned to the straggler, for the benchmarks Wordcount and Wordmean, the increased makespan under *LATE* is even larger than the one under *Non-specu*. Under eSplash, as all the tasks running on the straggler have created speculative tasks on other 'faster' nodes, nearly no time is wasted to recover the failed tasks on the straggler when it's shut down. eSplash shows the best performance on the increased makespan which is averagely 42.4% shorter than *Non-specu* and 45.8% shorter than *LATE*.

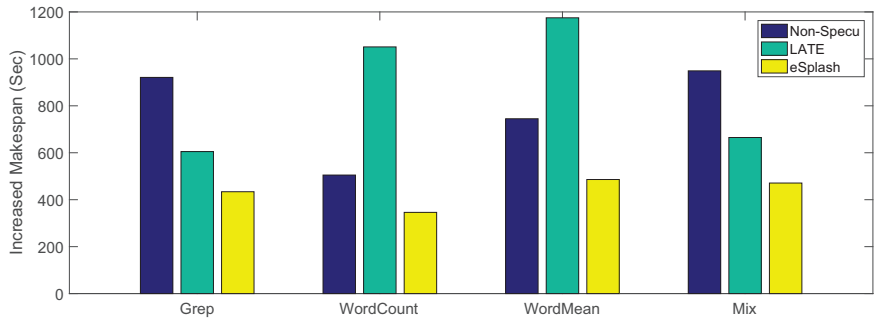

Figure 5.8: Increased makespan with a straggler which will be restarted

5.4 Related Work

MapReduce is a programming model and an associated implementation for processing and generating large data sets [46] [47]. J.D., etc. provide a detailed demonstration of MapReduce, especially a Straggler Backup Policy [46] [47] where master schedules backup executions of the remaining in-progress tasks and the task is marked as completed either the primary or the backup execution completes. Preceding policy inspires Speculative Execution [48] where task predicted to be slow can launch a redundant copy to re-execute.

However, due to system perturbations and equipment partial upgrade, most clusters in industry are not homogeneous anymore. Regarding to this truth, a large volume of work aiming at improving performance of Hadoop in a heterogeneous cluster has been done recently. B.R., etc [49] provide guidelines on how to overcome bottlenecks of heterogeneous clusters. Based on previous suggestions, a hybrid solution [50] of FIFO, FairSharing [51] and COSHH [52] is introduced based on job classification. What's more, J.X. etc. [53] place data on different nodes to assure a balanced load aiming at improving performance by optimizing data locality. S.G. etc. [54] propose a ThroughputScheduler which dynamically selects nodes by optimally matching job requirements to node capabilities. Targeting on the problem that speculation

mechanism degrades predictability of a cluster, Hopper [55] finds a balance between scheduling decision and speculation, also retrieves outstanding result after testing on Hadoop, Spark [6] and Sparrow [43], both centralized and decentralized schedulers.

Not only about scheduler, some other work also seeks the opportunity to increase the usage of storages in a heterogeneous cluster. For instance, N.S.I. and X.L. [56] propose new hybrid design and data placement policies to accelerate HDFS. Similarly, Cura [57] optimizes global resource utilization by configuring MapReduce jobs from the view of a service provider.

Nevertheless, previous work neglects that many unnecessary speculative tasks generated by slow nodes is one of the most important reasons for traditional Speculative Execution strategy incapable of adapting heterogeneous environment. LATE Scheduler [35] is created to solve the previous problem by only speculatively execute a copy of task that will finish farthest among all currently running tasks. Unfortunately the estimation of tasks' remaining execution time in LATE is not accurate enough, especially it is unable to make self-adjustment to adapt the system. Inspired by preceding work, we create eSplash, which labels nodes into levels for system to accurately and quickly identify straggling nodes. Our eSplash not only shows high performance on YARN [3], Next Generation MapReduce of Hadoop, but also can be integrated into other cloud computing systems.

5.5 Summary

This work studies the speculative execution in a large-scale heterogeneous computing cluster. Our goal is to mitigate the impact of node failures in the cluster. We develop a new speculation scheme eSplash which can efficiently and quickly identify the stragglers and submit the speculative tasks to the most appropriate nodes and avoid resource waste on the unnecessary speculation execution. We have implemented our solution on the Hadoop YARN platform, and conducted extensive experiments

with various workloads. The results show a significant improvement on distinguishing the stragglers, assigning speculative tasks, and reducing the impact of stragglers on the makespan compared to a conventional YARN system.

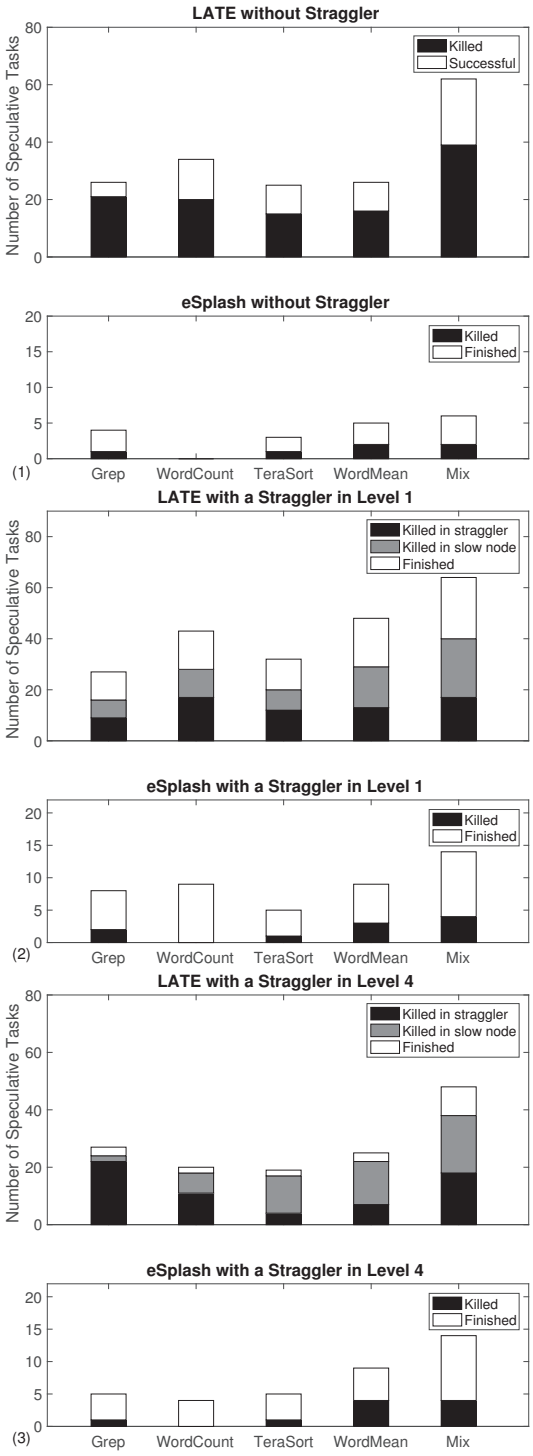

Figure 5.3: The Speculative tasks created under both *LATE* and *eSplash*: (1) without Straggler, (2) with one straggler on node level 1, (3) with one straggler on node level 4

CHAPTER 6

CONCLUSION

6.1 Dissertation Summary

In the era of big data, we envision that large-scale big data computing systems become ubiquitous to serve a variety of applications and customers. It is motivated by two factors. First, big data processing has shown the potential of benefiting many applications and services ranging from financial service, health applications, customer analysis, to social network applications. With more and more daily generated data, powerful processing ability will become the focus for research and development. Second, with the rise of cloud computing, it is now inexpensive and convenient for regular customers to rent a large cluster for data processing. Therefore, how to improve the performance in terms of execution times is the top issue on the list, especially when we imagine that a cluster computing system will often serve a large volume of jobs in a batch. In this dissertation, we mainly investigate the common characteristics of large-scale big data computing systems and aim to improve their efficiency and performance through more effective resource management and scheduling.

To achieve this target, three of our resource management schemes are introduced in this dissertation. First, we develop a fair and efficient resource allocation and scheduling for clusters called FRESH which can dynamically allocate resource based on the workloads of every stage of each job. FRESH not only reduces the makespan but also guarantees the fairness of a batch of jobs. Another main contribution of this dissertation lies in a new scheduling strategy OMO which is based on the dependency between stages. Unlike the traditional job scheduling problem, the later stage of a job

usually starts before the former stage is finished to "shuffle" the intermediate data. Motivated by this feature, OMO dynamically adjusts the start of the later stages of different applications to reduce the makespan of jobs and improve the resource utilization of the system. In addition, we present an efficient resource management scheme, eSplash, to deal with the node failures in the cluster. Based on monitoring the performance of every node and the execution of every task in the cluster, eSplash can detect the abnormal nodes quickly and accurately, create duplicated tasks on them, and assign these tasks to the most appropriate nodes. Therefore, eSplash can efficiently mitigate the impact of abnormal nodes on the system performances.

All our resource management schemes and algorithms are implemented in Hadoop MapReduce and Hadoop YARN, and evaluated in the large clusters on Amazon AWS EC2 and NSF CloudLab with multiple standard benchmarks. Our works achieve 16% to 67% improvement on system performance compared to the default scheduling policies and speculative mechanism in Hadoop MapReduce and Hadoop YARN.

6.2 Future Work

As resource management is critical for the big data computing systems, in the future, we would like to explore more resource allocation and scheduling algorithms to improve the efficiency of the cluster computing platforms. More dynamic factors will be investigated and implemented in the resource management. For example, data locality is important for the execution of tasks. Tasks with input data stored locally perform better execution time. Data blocks in the existing systems are randomly deployed in the distributed systems. We plan to develop an efficient and dynamic data block placement strategy that can improve the percentage of tasks with local data in the cluster. Furthermore, as new big data computing systems are recently developed to support more complex applications such as machine learning and graph processing, we also plan to investigate features and properties of the new systems

and develop new efficient resource management schemes and scheduling algorithms to improve the system performance.

REFERENCE LIST

[1] Idc reports. https://www.emc.com/collateral/analyst-reports/idc-digital-universe-2014.pdf.

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

[3] Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[4] Benjamin Hindman, Andy Konwinski, Matei Zaharia, et al. Mesos: A platform for fine-grained resource sharing in the data center. NSDI, pages 22–22. USENIX, 2011.

[5] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1357–1369, New York, NY, USA, 2015. ACM.

[6] Apache Spark. http://spark.apache.org.

[7] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[8] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[9] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[10] A Storm is coming: more details and plans for release. `https://blog.twitter.com/2011/a-storm-is-coming-more-details-and-plans-for-release`.

[11] Apache Mahout. `http://mahout.apache.org/`.

[12] Jiayin Wang, Yi Yao, Ying Mao, Bo Sheng, and Ningfang Mi. Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters. In *CLOUD*, 2014.

[13] Jiayin Wang, Yi Yao, Ying Mao, Bo Sheng, and Ningfang Mi. Optimize mapreduce overlap with a good start(reduce) and a good finish(map). In *IPCCC*, Dec 2015.

[14] Jiayin Wang, Teng Wang, Zhengyu Yang, Ningfang Mi, and Sheng Bo. eSplash: Efficient Speculation in Large Scale Heterogeneous Computing Systems. In *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[15] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, 2010.

[16] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance. In *MASCOTS*, Aug 2012.

[17] Michael Isard, Vijayan Prabhakaran, Jon Currey, et al. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, pages 261–276, 2009.

[18] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC*, pages 235–244, 2011.

[19] Jorda Polo, David Carrera, Yolanda Becerra, et al. Performance-driven task co-scheduling for mapreduce environments. In *NOMS*, pages 373–380, 2010.

[20] Next generation mapreduce scheduler. `http://goo.gl/GACMM"`.

[21] Xiao Wei Wang, Jie Zhang, Hua Ming Liao, and Li Zha. Dynamic split model of resource utilization in mapreduce. In *DataCloud-SC*, pages 21–30, 2011.

[22] Jordà Polo, Claris Castillo, David Carrera, et al. Resource-aware adaptive scheduling for mapreduce clusters. In *Middleware*, 2011.

[23] B. Sharma, R. Prabhakar, S. Lim, M.T. Kandemir, and C.R. Das. Mrorchestrator: A fine-grained resource orchestration framework for mapreduce clusters. In *CLOUD*, pages 1–8, June 2012.

[24] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared systems. Technical report, Digital Equipment Corporation, December 1984.

[25] Amazon EC2 Instances. `http://aws.amazon.com/ec2/instance-types/`.

[26] Purdue mapreduce benchmarks suite. `http://web.ics.purdue.edu/~fahmad/benchmarks.htm`.

[27] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954.

[28] Capacity scheduler. `http://hadoop.apache.org/common/docs/r1.0.0/capacity_scheduler.html`.

[29] Fair scheduler. `http://hadoop.apache.org/common/docs/r1.0.0/fair_scheduler.html`.

[30] Jian Tan, Xiaoqiao Meng, and Li Zhang. Performance analysis of coupling scheduler for mapreduce/hadoop. In *INFOCOM*, pages 2586–2590, March 2012.

[31] Jian Tan, Xiaoqiao Meng, and Li Zhang. Delay tails in mapreduce scheduling. In *SIGMETRICS*, pages 5–16, 2012.

[32] Jian Tan, Xiaoqiao Meng, and Li Zhang. Coupling task progress for mapreduce resource-aware scheduling. In *INFOCOM*, pages 1618–1626, April 2013.

[33] Weina Wang, Kai Zhu, Lei Ying, Jian Tan, and Li Zhang. Map task scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality. In *INFOCOM*, pages 1609–1617, 2013.

[34] Yanfei Guo, Jia Rao, and Xiaobo Zhou. ishuffle: Improving hadoop performance with shuffle-on-write. In *ICAC*, pages 107–117, 2013.

[35] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, et al. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 29–42, 2008.

[36] Yi Yao, Jianzhe Tai, Bo Sheng, and Ningfang Mi. Scheduling heterogeneous mapreduce jobs for efficiency improvement in enterprise clusters. In *IM*, pages 872–875.

[37] Yi Yao, Jiayin Wang, Bo Sheng, et al. Using a tunable knob for reducing makespan of mapreduce jobs in a hadoop cluster. In *CLOUD*, pages 1–8, June 2013.

[38] Y. Yao, J. Wang, B. Sheng, C. Tan, and N. Mi. Self-adjusting slot configurations for homogeneous and heterogeneous hadoop clusters. *IEEE Transactions on Cloud Computing*, PP(99):1–1, 2015.

[39] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX*, 39(6), December 2014.

[40] S. Alspaugh Y. Chen and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. VLDB, pages 1802–1813, 2012.

[41] Yi Yao, Jiayin Wang, Jason Sheng, Bo aMulti-resource Packing for Cluster Schedulersnd Lin, and Ningfang Mi. Haste: Hadoop yarn scheduling based on task-dependency and resource-demand. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing*, CLOUD '14, pages 184–191, Washington, DC, USA, 2014. IEEE Computer Society.

[42] Y. Yao, H. Gao, J. Wang, N. Mi, and B. Sheng. Opera: Opportunistic and efficient resource allocation in hadoop yarn by harnessing idle resources. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, Aug 2016.

[43] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, pages 69–84, 2013.

[44] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, et al. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*, pages 351–364, 2013.

[45] Michael Isard, Mihai Budiu, Yuan Yu, et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[46] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: A flexible data processing tool. *Commun. ACM*, 53(1):72–77, January 2010.

[48] Tom White. Speculative execution. In *Hadoop: The Definitive Guide*, chapter 6. O'Reilly Media, Inc., 2012.

[49] B.Thirumala Rao, N.V.Sridevi, V.Krishna Reddy, and L.S.S.Reddy. Performance issues of heterogeneous hadoop clusters in cloud computing. 07 2012.

[50] Aysan Rasooli and Douglas G. Down. A hybrid scheduling approach for scalable heterogeneous hadoop systems. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1284–1291, Washington, DC, USA, 2012. IEEE Computer Society.

[51] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, et al. Job scheduling for multi-user mapreduce clusters. Technical report, EECS Department, University of California, Berkeley, Apr 2009.

[52] Aysan Rasooli Oskooei and Douglas G. Down. Coshh: A classification and optimization based scheduler for heterogeneous hadoop systems. *Future Generation Comp. Syst.*, 36:1–15, 2014.

[53] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, J. Majors, A. Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–9, April 2010.

[54] Shekhar Gupta, Christian Fritz, Bob Price, Roger Hoover, Johan Dekleer, and Cees Witteveen. Throughputscheduler: Learning to schedule on heterogeneous hadoop clusters. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 159–165, San Jose, CA, 2013. USENIX.

[55] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 379–392, New York, NY, USA, 2015. ACM.

[56] N. S. Islam, X. Lu, M. Wasi ur Rahman, D. Shankar, and D. K. Panda. Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 101–110, May 2015.

[57] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bryan Langston. Cura: A cost-optimized model for mapreduce in a cloud. In *IPDPS*, 2013.