Abstract

Over the past decades, to deal with the rapidly growing data volumes in this big data era, data-parallel clusters were designed to process the data-parallel jobs, each of which runs on many machines in parallel by dividing the entire *job* into individual *tasks* that process different pieces of data. Many large organizations such as Facebook, Google and Yahoo! have deployed their data-parallel clusters to process thousands of jobs every day.

Many previous studies show that current production clusters process increasingly *diverse* jobs with various job characteristics (e.g., input data size, shuffle data size, and output data size). First, previous works have shown that there are a large number of shuffle-heavy jobs in current production workloads, which may result in network bottleneck in the clusters and hence greatly degrades the performance of the clusters. Second, the small jobs (i.e., jobs with small data to process) often dominate the workloads in production. However, current architectures and schedulers of data-parallel clusters were originally built to process *data-intensive jobs* that have large input datasets. The mismatch between the actual workloads and design objectives results in poor performance of the jobs in the clusters.

The key contribution of this dissertation is designing job schedulers in different architectures of data-parallel clusters to handle the diverse workloads. First, we design a Network-Aware job Scheduler (NAS) for data-parallel frameworks in traditional datacenter networks that can schedule tasks carefully to avoid and reduce the network congestion caused by the large amount of shuffle-heavy jobs.

Second, to build high-capacity, low-latency datacenter networks, researchers have proposed hybrid electrical/optical datacenter network (Hybrid-DCN) architectures, which augment the traditional electrical-packet switching (EPS) datacenter network with an on-demand *rack-to-rack* network using the optical circuit switch (OCS). In

order to utilize Hybrid-DCN efficiently, job schedulers for data-parallel frameworks must keep pace to meet the needs of such hybrid networks. Thus, we design a job scheduler called JobPacker that can efficiently exploit OCS in Hybrid-DCN to improve the job performance by finding the optimal tradeoff between the parallelism and traffic aggregation.

Third, recent works advocate hybrid scale-up/scale-out clusters (in short Hybrid clusters) to handle the workloads that consist of a majority of jobs with small input data sizes and a small number of jobs with large input data sizes. However, previous works did not solve the challenges for job placement and data placement in designing such a Hybrid cluster. We design the job placement and data placement strategies in Hybrid cluster to address the challenges, which can significantly improve the performance of workloads with a large amount of small jobs.

Finally, we demonstrate through trace-driven simulation and real cluster evaluation that our proposed schedulers improve the performance of diverse workloads in the data-parallel clusters in terms of throughput and job completion time.

To my family...

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank many people who have helped and supported me through this long Ph.D. journey.

First, I thank my advisor, Professor Haiying Shen, whose encouragement, guidance and support from the initial to the final level of my Ph.D. study enabled me to develop an understanding of the subject. Apart from research expertise, I also learn a lot from her about organizing and managing research activities such as project/grant proposals and international conferences. I am especially thankful for her guidance and valuable suggestions at every step of my career and life.

I acknowledge my committee members, Professor Andrew Grimshaw, Professor Mary Lou Soffa, Professor David Evans, Professor Zongli Lin for providing me valuable comments and suggestions in writing this dissertation.

I am grateful to my collaborators for sharing their time, ideas and enthusiasm. I had been fortunate to collaborate with each one of them at Clemson University, University of Virginia, Lehigh University, and etc.

I am proud of knowing many colleagues and friends in Pervasive Communications Laboratory. I am thankful to all of you for supporting my research and providing me suggestions. Thank you for the laughs and happy days in the office.

All my friends and family have been a source of inspiration in my life. I thank my family for their love and support at all times. None of my achievements would have been possible without their infinite support, and steely determination to give me the best possible education.

To my parents, Zhiwei Li and Ruimei Zhu, thank you for your love and all your advices through the years. I want to express my deepest gratitude to you for your financial support, raising me up, and encouraging me to complete this doctoral degree when I was depressed. Thank you for flying to United States every year to accompany me.

Last but not least, I especially thank my girlfriend, Ms. Yuhui Lyu, who accepted my long hours at research with open heart and mind and encouraged me at every step of my career and life. Without her understanding, support and having her beside me, this Ph.D. journey would have felt much longer and more stressful. I am lucky to have you in my life and can't wait to see our future life.

Chapter 1

Introduction

Data-parallelism is a form of parallelization that focuses on distributing the data across different nodes, which operate on the data in parallel. To deal with the rapidly growing data volumes in this big data era, data-parallel clusters were designed to process the data-parallel jobs, each of which runs on many machines in parallel by dividing the entire *job* into individual *tasks* that process different pieces of data.

Many large organizations such as Facebook, Google and Yahoo! have deployed their data-parallel clusters (e.g., MapReduce [47], Cosmos [27], and Spark [17]) to process thousands of jobs every day. In the data-parallel cluster of an organization, multiple users from different groups in the organization usually share the cluster and run a mix of long batch jobs and short interactive queries. The sharing feature enables the *statistical multiplexing*, which leads to lower cost since each group does not need to build its private cluster.

For example, in Facebook MapReduce cluster [174], event logs from Facebook's website are imported into a Hadoop cluster every hour. A variety of jobs (e.g., analyzing usage patterns, detecting spam, data mining and ad optimization) are run periodically to analyze the event logs from Facebook's website. In addition to "production" jobs that run periodically, there are some experimental jobs, such as machine learning computations and ad-hoc queries, whose running times range from several hours to 1-2 minutes.

With the exponential increase of the volumes of computation and data, improving the performance of data-parallel clusters (i.e., throughput of the cluster and average job completion time) gets increasingly important [174]. Researchers have been exploring different methods to improve the cluster performance, such as job scheduling [26, 62, 75, 86, 176], data placement [13, 15, 61, 109, 111, 112], intermediate data shuffling [42, 44, 155, 166] and improving small job performance [60].

1.1 Problems Due to Job Diversity

Although data-parallel frameworks and their schedulers were originally designed to process *data-intensive jobs* with large input datasets, many previous studies [3, 18, 32, 60, 94, 137] show that current production clusters process increasingly *diverse* jobs with various job characteristics (e.g., input data size, shuffle data size, and output data size). The mismatch between the actual workloads and design objectives results in poor performance of the jobs in the clusters.

First, previous work has shown that 60% and 20% of the jobs are shuffle-heavy jobs (i.e., jobs with a large shuffle data size) on the Yahoo! [32] and Facebook MapReduce clusters [172], respectively. A large amount of shuffle-heavy jobs in the workloads may result in network bottleneck in the clusters, which greatly degrades the performance of the clusters. This is because the shuffle phase in MapReduce is an all-Map-to-all-Reduce communication phase and the transfer of shuffle data is the dominant source of network traffic [3]. In addition, while there is full bisection bandwidth within a rack, traditional datacenter networks commonly use network topologies with link oversubscription ranging from 3:1 to 40:1 from the racks to the core [3, 47, 77, 172], as shown in Figure 1.1. What is worse is that more than 50% of the cross-rack

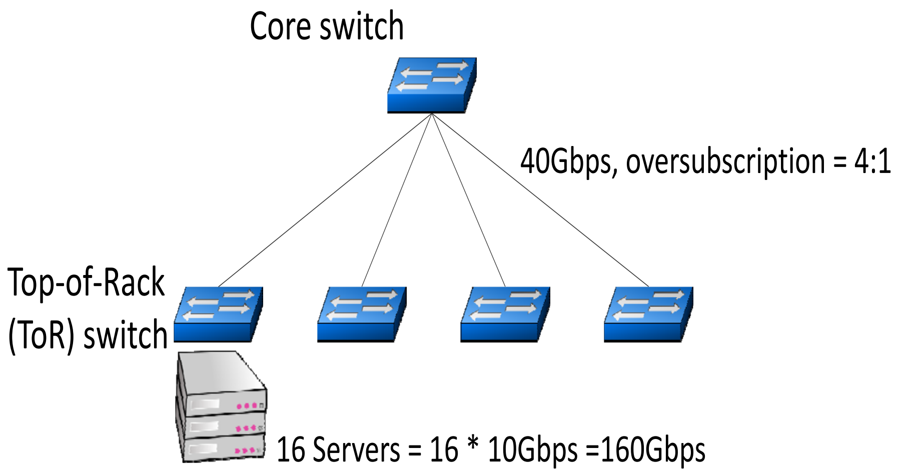

Core switch
40Gbps, oversubscription = 4:1
Top-of-Rack (ToR) switch
16 Servers = 16 * 10Gbps =160Gbps

Figure 1.1: Traditional datacenter network architecture.

bandwidth may be used for background data transfer [34, 94]. These two reasons further reduce the available cross-rack bandwidth for the shuffle-heavy jobs.

Second, the small jobs (i.e., jobs with small data to process) often dominate the workloads in production, although the data-parallel frameworks were originally built for large jobs (i.e., jobs with large data to process). For example, the production workloads in Microsoft and Yahoo! clusters have median job input size under 14GB [18, 60, 137] and 90% of jobs on a Facebook cluster have input size under 100GB [13]. Many previous studies [18, 60, 137] show that the small jobs in the production clusters often have poor response time, since these frameworks were built without having short jobs in mind. For example, one of their key design goals was scalability with respect to the job and cluster sizes. Obviously, large clusters of cheap servers are the most cost-effective way to process exabytes, petabytes, or multi-terabytes of data [18]. However, is it really the best option for workloads where the majority are small jobs?

1.2 Proposed Scheduling Techniques

Accordingly, we worked on three research thrusts to improve the performance of diverse workloads in data-parallel clusters. While we present our designs and results in the MapReduce setting [47] – one of the most widely used data-parallel frameworks, the designs and results generalize to any data flow based cluster computing system, like Dryad [90] and Spark [17].

1.2.1 Network-aware Scheduler in Traditional Datacenter Network Architecture

In this research thrust, we aim to design a network-aware scheduler for the data-parallel frameworks in conventional datacenter network architecture.

Although there are many shuffle-heavy jobs in current commercial production clusters, many schedulers [26, 62, 71, 91, 172] only focus on achieving fairness and increasing data locality in the map stage. For example, Capacity [26], Fair [62] and Dominant Resource Fairness [71] schedulers aim to achieve the fairness of resource allocation (e.g., CPU, memory, storage and bandwidth) among users or jobs. Delay scheduler in [91, 172] aims to improve the data locality (i.e., the node running a task has its required data) in the map stage of MapReduce.

ShuffleWatcher [3] has been proposed to reduce cross-rack shuffle data traffic and avoid network congestion. When a job is submitted, ShuffleWatcher pre-computes a placement of its map and reduce tasks leading to the minimum network traffic. Then ShuffleWatcher schedules the tasks of the job based on the pre-computed placement. When the cross-rack network is congested, ShuffleWatcher delays scheduling reduce tasks (and hence the shuffle data transfer) by assigning map tasks instead, and assigns a job's reduce tasks to a rack based on the amount of shuffle data on that rack.

Although ShuffleWatcher reduces the cross-rack traffic of the reduce tasks, it increases the cross-rack traffic to read map input data, as mentioned in [94].

Therefore, it is important to consider the shuffle phase in task scheduling to avoid and reduce network congestion without compromising cluster performance. Specifically, we identify three main challenges in designing such a scheduler below.

1. **Balancing network traffic temporally.** Previous study [120] shows that the cross-rack network bandwidth is not always fully utilized at all time. This motivates us to design a scheduler to *balance the network traffic temporally*, so that the cross-rack network is less likely to be congested. To balance the network traffic temporally, we can constrain the maximum total shuffle data generated in the cluster at a time. Thus, the first challenge is how to place tasks to constrain the maximum total shuffle data generated in the cluster at a time.

2. **Enforcing data locality for the shuffle data.** Shuffle-heavy jobs generally transfer a large amount of shuffle data across racks, resulting in high requirement of cross-rack network bandwidth. Many jobs executed simultaneously exacerbate the pressure of the need of network bandwidth. Unlike the CPU, memory, and disk resources that are easy to scale-up by deploying more hardware, network bandwidth is hard to scale-up with current hardware technology [3]. Current datacenter network architectures [5] typically provide cross-rack bandwidth and within-rack bandwidth per node with a ratio of 5:1 to 20:1 [47, 78, 172]. Poor scheduling of reduce tasks on different nodes may lead to cross-rack network congestion, which degrades the performance. Thus, it is crucial to place the tasks to enforce certain data locality for the shuffle data, i.e., reducing the amount of shuffle traffic that is transferred cross-rack.

3. **Reducing cross-rack network congestion.** The overlap between the map

and shuffle phases improves the performance. ShuffleWatcher sacrifices such overlap to reduce network congestion [3], which degrades the performance. Therefore, it is important to reduce congestion without sacrificing the intra-job concurrency to achieve better performance.

In this research thrust, we design a network-aware scheduler (NAS) that incorporates three mechanisms to handle these three challenges respectively. The three mechanisms are map task scheduling (MTS), congestion-avoidance reduce task scheduling (CA-RTS) and congestion-reduction reduce task scheduling (CR-RTS). NAS considers the shuffle traffic in both the map and reduce task scheduling.

1. **Map task scheduling (MTS).** MTS places the map tasks while constraining the size of the shuffle data transmitted from each node at a time under a certain threshold. More importantly, through constraining the shuffle data size on each node, MTS balances the network traffic temporally since it constrains the maximum total shuffle data being processed in the cluster, which avoids the cross-rack network congestion. Specifically, based on the predicted shuffle data size [160] of each map task, MTS calculates the total shuffle data size of all the map tasks in every node. Once a worker node requests for a map task, MTS checks the user list in the top-down manner until finding a map task with an output data size that can keep the updated total shuffle data size in the node no higher than the threshold, while still providing a certain degree of data-locality and fairness.

2. **Congestion-avoidance reduce task scheduling (CA-RTS).** CA-RTS aims to avoid the cross-rack network congestion while increasing cluster performance. It adaptively adjusts the map completion threshold of jobs based on their shuffle data sizes. Also, for each job, it distributes its reduce tasks based on the

distribution of its shuffle data among the racks in order to minimize cross-rack traffic.

3. **Congestion-reduction reduce task scheduling (CR-RTS).** Usually, both shuffle-heavy and shuffle-light jobs (i.e., jobs that generate very little shuffle data size) run at the same time. Considering that the shuffle-light jobs consume negligible cross-rack network bandwidth [3], rather than delaying the shuffle phase of all jobs as in ShuffleWatcher, CR-RTS does not delay the shuffle phases of shuffle-light jobs. As a result, CR-RTS reduces network congestion while reducing the sacrifices of overlap between the shuffle and map phases.

We implemented NAS in Hadoop on a cluster. Our trace-driven simulation and real cluster experiment demonstrate the superior performance of NAS on improving the throughput (up to 62%), reducing the average job completion time (up to 44%) and reducing the cross-rack traffic (up to 40%) compared with state-of-the-art schedulers.

1.2.2 Job Scheduler in Hybrid Electrical/Optical Datacenter Network Architecture

In this research thrust, we aim to design a job scheduler for the data-parallel frameworks in a recently proposed datacenter network architecture.

To supply sufficient network bandwidth for data transfer, we can increase network capacity. However, high-speed network interface cards (NICs) incur high capital expenditures (CapEx) and high operating expenditures (OpEx) due to power consumption. An answer to the problem of high CapEx and OpEx for high-speed networking lies in optical circuit switch (OCS), which can provide a bandwidth up to 100Gbps [63, 162]. OCS can connect any of its input port to any of its output port, but one input port can be connected to only one output port at a time. To change the input-to-output connection, one needs to *reconfigure* the OCS connection which results in a *reconfiguration delay* on the order of μs-to-ms. This reconfiguration delay is significantly higher than the latency of packet switching that is in the order of ns. Therefore, OCS can only be added as a complement to traditional electrical packet switch (EPS), but not a replacement.

Recently, several studies propose hybrid electrical/optical datacenter network (in short *Hybrid-DCN*) designs [30, 63, 134, 162], which augment the EPS datacenter network with an on-demand *rack-to-rack* network using the OCS. The top-of-rack (ToR) switches are connected with a core EPS and an OCS, forming packet-switched network and circuit-switched network, respectively. Typically, in Hybrid-DCN, each rack connects to one input port and one output port, which means that one rack can send data via OCS to only one other rack at a time. Due to such limitation, in Hybrid-DCN, OCS is used only for large data transfers (e.g., 1.125GB) between racks so that the overhead of μs-to-ms reconfiguration delay is negligible.

Current state-of-the-art schedulers (e.g., Fair [62] and Corral [94]) in data-parallel frameworks fail to leverage OCS to accelerate the data transfer, since they either spread the tasks of a job (e.g., map and reduce tasks in MapReduce) among racks which generates many small flows or schedule the tasks of a job to avoid using cross-rack traffic which cannot exploit OCS to accelerate the data transfer. Thus, new job schedulers for data-parallel frameworks are required to meet the need of Hybrid-DCN.

To take full advantage of Hybrid-DCN, we could aggregate the data to be transferred by placing the tasks of a job in only a few racks. However, it may sacrifice the basic principle of data-parallel frameworks – parallelism (i.e., the tasks of a job running concurrently), since each rack may have a limited number of containers available at a time. If a rack does not have sufficient available resources to run all the assigned tasks concurrently, it increases the latency of the job (i.e., the duration from the start of a job until its completion). Hence, there is a tradeoff between parallelism and traffic aggregation. In this work, we propose a job scheduler called JobPacker that aims to efficiently leverage OCS in Hybrid-DCN to improve job performance by balancing such tradeoff.

JobPacker attempts to aggregate the shuffle data transfers of a job in order to use OCS effectively. JobPacker consists of an offline scheduler (to schedule recurring jobs in the next unit period of time) and an online scheduler. The offline scheduler consists of a job profiler and a job manager.

- The job profiler exploits the fact that most jobs in a cluster are often recurring and have predictable job characteristics [2, 66, 94] to find all feasible (map-width, reduce-width) pairs (defined as the number of racks to run the map and reduce tasks) of each shuffle-heavy recurring job that can aggregate sufficient shuffle data to use OCS effectively while achieving sufficient parallelism.

- Then, the job manager finds the best (map-width, reduce-width) pair with the shortest completion time, and also generates a global schedule including which racks to run each recurring job and the sequence to run the map/reduce tasks of recurring jobs in each rack that yields the best performance (i.e., high throughput for batch jobs and short completion time for online jobs). The job manager also has a new sorting method to prioritize the recurring jobs in scheduling to prevent high resource contention while fully utilizing cluster resources.

- Based on the determined schedule, when jobs and their datasets are submitted, the online scheduler places input datasets and schedules the recurring jobs to racks accordingly. It schedules non-recurring (i.e., ad-hoc) jobs to the resources not assigned to the recurring jobs. As the recurring jobs can finish earlier by more efficiently utilizing OCS, it leaves more computing resources and network bandwidth to ad-hoc jobs to complete earlier [94].

We have evaluated JobPacker using large-scale simulation and small-scale emulation on GENI based on a Facebook trace [32]. The results show that JobPacker reduces the makespan of a batch of jobs (i.e., the time to finish all the jobs) up to 49% and the median job completion time up to 43%, compared to the state-of-the-art schedulers in Hybrid-DCN.

1.2.3 Job Placement and Data Placement in Hybrid Scale-up/out Cluster

In this research thrust, we aim to design job placement and data placement strategies for hybrid scale-up/out cluster to handle the diverse workloads for high performance.

Conventionally, the data-parallel clusters consist of a large number of scale-out machines. Recent studies [18, 105, 110] advocate to explore hybrid scale-up and scale-out heterogeneous clusters (in short Hybrid clusters) to handle current workloads, since previous studies [32, 60] show that a large amount of jobs (e.g., more than 80%) in current workloads only process small data size and have diverse job characteristics (e.g., shuffle data size). Here, scale-up is vertical scaling, which means adding more resources to the nodes of a system, typically the processors and RAM, and scale-out is horizontal scaling, which refers to adding more nodes with few processors and RAM to a system. Appuswamy *et al.* [18] evaluated the jobs with different characteristics on scale-up and scale-out machines and found that scale-up is significantly better in some cases, than scale-out. Hence, we are motivated to design a Hybrid cluster to handle the diverse workloads in the clusters for high performance.

Hybrid cluster is essentially a heterogeneous cluster. There have been plenty of efforts [4, 68, 105, 173] focusing on improving the performance in heterogeneous clusters. However, since these proposals do not consider the workloads with diverse job characteristics and do not take advantage of this feature, they are not suitable for the Hybrid cluster, which is designed to process such workloads. For example, since we intentionally introduce scale-up machines to deal with job diversity in Hybrid cluster, we expect the jobs that favor scale-up to run on scale-up machines, which is not considered in previous work [4, 68, 105, 173]. Li *et al.* [110] identified the challenges of job and data placement to design a Hybrid cluster and configured it with a remote file system to solve the challenges. However, the proposed solution with remote file system causes a large amount of remote data transfer.

In this work, we focus on the design of Hybrid cluster with conventional local file system (e.g., HDFS), where some scale-out machines are replaced by scale-up machines that have the same cost with the scale-out machines. In other words, we aim to design a Hybrid cluster to improve the performance of big data analytics with the same monetary cost, namely a more *cost-effective* cluster. First, we identify the key challenges in designing Hybrid clusters to improve the performance of big data analytic clusters. There are two main challenges – job placement challenges (J.1, J.2, and J.3) and data placement challenges (D.1 and D.2).

- **J.1** A proper job placement strategy is essential for the Hybrid cluster. The jobs with different job characteristics may benefit differently from scale-up and scale-out machines. Therefore, we need to adaptively place the jobs to scale-up or scale-out machines based on their job characteristics to achieve the most benefits for the jobs.

- **J.2** The job placement strategy should consider the load balancing. After we schedule the jobs to scale-up or scale-out machines based on their job characteristics, severe load imbalance may occur on different types of machines. For example, suppose a large amount of small jobs are submitted to Hybrid cluster simultaneously, while there are not many large jobs. If we still run the jobs on different machines based on their job characteristics, it leads to overload on the scale-up machines, while under-utilizing on the scale-out machines. Hence, the job placement strategy needs to adaptively schedule small jobs to the other type of machines to avoid overload.

- **J.3** It seems that it is straightforward to address J.2 challenge by moving tasks from scale-up (scale-out) to scale-out (scale-up) when one type of machines are under-utilized. However, this mechanism is not sufficient since the scale-up and scale-out machines have different capability of computing for the tasks, which is a typical problem in heterogeneous clusters. The different computing speed results in significant imbalance progress of tasks within a job, that is, fast machines complete the tasks faster and need to wait for the slow machines to complete the tasks of the same job. This leads to a non-negligible delay and significantly degrades the performance of the job [4, 173].

- **D.1** Data locality is an essential factor for high performance [172]. Since we adaptively place a job to scale-up or scale-out machines based on its job characteristics, in order to maintain data locality, we need to accordingly place the data of every job to the machines that the job is supposed to run on.

- **D.2** We cannot simply place the data of scale-up jobs on scale-up machines and the data of scale-out jobs on scale-out machines. This is because the jobs may be adjusted between scale-up and scale-out machines for load balancing according to J.2 challenge. If the adjustment of some jobs occurs, the data locality cannot maintain, which degrades the performance of Hybrid.

Then, we propose corresponding job placement and data placement strategies that can be easily implemented to handle the challenges.

- **Job placement strategy** In order to achieve the best performance on Hybrid cluster, we use a Support Vector Machine (SVM) model to classify the submitted jobs into two groups, scale-up jobs (i.e., small jobs) and scale-out jobs (i.e., large jobs), based on the characteristics of the jobs. We use the term *scale-up jobs* to refer to the jobs that are better to run on scale-up machines, and the term *scale-out jobs* to refer to the jobs that are better to run on scale-out machines. The scale-up jobs are scheduled on scale-up machines, while the scale-out jobs are scheduled on scale-out machines.

  This policy ensures performance improvements for both small and large jobs. The small jobs can benefit from the use of scale-up machines, while the large jobs obtain benefits because placing the small jobs on scale-up machines significantly mitigates the resource contention between large and small jobs on scale-out machines.

  Further, to balance the loads between scale-up and scale-out machines, we propose a job stealing strategy, which adaptively steals scale-up jobs to run on scale-out machines when the scale-out machines are under-utilized.

- **Data placement strategy** In order to solve the data placement challenges, we propose a replication-based placement strategy. We place the replicas of each data block on both scale-up and scale-out machines. Specifically, for a data block, the first and second replicas are placed on the scale-out machines, while the third replica of the data block is placed on the scale-up machines. The proposed data placement strategy can maintain high data locality for both scale-up and scale-out jobs. Furthermore, when the job stealing strategy is used, it does not decrease the data locality of the stolen jobs.

Finally, we implement a Hybrid cluster with the above two strategies, and evaluate its performance through real cluster run and large-scale trace-driven simulation.

Using the workload derived from Facebook [32], we show that with our proposed strategies, the Hybrid cluster can reduce the makespan of the workload up to 40% and the median job completion time up to 60%, compared to traditional scale-out clusters with state-of-the-art schedulers.

1.3 Contributions

My thesis statement is that we can improve the performance of current state-of-the-art schedulers (e.g., Fair and Delay schedulers in Hadoop) by

- balancing the network traffic temporally and enforcing the data locality for shuffle data,
- aggregating the data transfers to efficiently exploit optical circuit switch in hybrid electrical/optical datacenter network while still guaranteeing parallelism of the jobs,
- and adaptively scheduling a job to either scale-up machines or scale-out machines that benefit the job the most in hybrid scale-up/out cluster.

The main dissertation consists of three proposed schedulers to support the thesis. First, we discuss a network-aware scheduler that can shape the network traffic temporally and reduce network congestion in the traditional datacenter network. We evaluate the proposed scheduler through real cluster experiment and trace-driven simulation.

Second, we design a job scheduler that aims to take full advantage of OCS in Hybrid-DCN by finding the optimal tradeoff between traffic aggregation and parallelism. We evaluate the proposed scheduler with Hybrid-DCN through trace-driven simulation and emulation.

Third, we present job placement and data placement strategies in hybrid scale-up/out cluster to adaptively select scale-up or scale-out machines based on the job characteristics to achieve the most benefits for the jobs. We demonstrate the effectiveness of our proposed strategies in hybrid scale-up/out cluster through real cluster experiment and trace-driven simulation.

Together, the experiments demonstrate that the performance of current state-of-the-art schedulers (e.g., Fair and Delay schedulers) in data-parallel clusters with diverse workloads is improved by the proposed schedulers.

The primary contributions of the dissertation are summarized as follows.

- **Network-aware scheduler in traditional datacenter network.** We identify the challenges in designing a job scheduler that can shape the network traffic temporally and reduce cross-rack network congestion. First, we need to place the tasks to constrain the maximum total shuffle data generated in the cluster at a time so that the network traffic is balanced temporally. Second, we need to place the reduce tasks to enforce certain data locality for the shuffle data, which reduces the cross-rack shuffle traffic and avoids cross-rack network congestion.

  To handle these challenges, we design a Network-Aware job Scheduler (NAS) that consists of three main mechanisms: i) map task scheduling (MTS), ii) congestion-avoidance reduce task scheduling (CA-RTS) and iii) congestion-reduction reduce task scheduling (CR-RTS). NAS considers the shuffle phase in both the map and reduce task scheduling. MTS constrains the size of shuffle data on each node when scheduling the map tasks. Through this mechanism, MTS constrains the maximum total shuffle data being processed in the cluster at a time, which helps avoid the cross-rack network congestion. When the network is not congested, CA-RTS is used to distribute the reduce tasks for each job based on the distribution of its shuffle data among the racks in order to minimize cross-rack traffic. When the network is congested, CR-RTS is used to schedule reduce tasks that generate negligible shuffle traffic to reduce the congestion. Our trace-driven simulation and real cluster experiment demonstrate the superior performance of NAS on improving the throughput, reducing the average job completion time, and reducing the cross-rack traffic, compared with state-of-the-art schedulers.

- **Job scheduler in hybrid electrical/optical datacenter network.** We identify the key challenge of designing a job scheduler that can efficiently utilize OCS in Hybrid-DCN is exploring the tradeoff between the parallelism and traffic aggregation. We design JobPacker, a job scheduler for data-parallel frameworks in Hybrid-DCN that aims to take full advantage of the OCS to improve job performance by exploring such tradeoff.

  Since many jobs in production are recurring with predictable characteristics, JobPacker uses an offline scheduler to explore the tradeoff between parallelism and traffic aggregation, and generates a global schedule including which rack-s to run each recurring job and the sequence to run the map/reduce tasks of recurring jobs in each rack that yields the best performance. Then, an online scheduler schedules recurring jobs based on the generated offline schedule, and schedules non-recurring jobs to the idle resources that are not assigned to recurring jobs. Trace-driven simulation and emulation show that JobPacker reduces the makespan and the median completion time in Hybrid-DCN, compared to the state-of-the-art schedulers.

- **Job placement and data placement in hybrid scale-up/out cluster.** We identify the job placement and data placement challenges of using hybrid scale-up/out cluster to improve the performance of workloads with a large amount of small jobs. For job placement, we need to adaptively place the jobs to scale-up or scale-out machines based on their job characteristics to achieve the most benefits for the jobs. In addition, after we schedule the jobs to scale-up or scale-out machines based on their job characteristics, severe load imbalance may occur on different types of machines. For data placement, since we adaptively place a job to scale-up or scale-out machines based on its job characteristics, in order to maintain data locality, we need to accordingly place the data of every job to the machines that the job is supposed to run on.

We design job placement and data placement strategies in hybrid scale-up/out cluster to address these challenges. We use a Support Vector Machine (SVM) model to classify the submitted jobs into two groups, scale-up jobs (i.e., small jobs) and scale-out jobs (i.e., large jobs), based on the characteristics of the jobs. Further, to balance the loads between scale-up and scale-out machines, we propose a job stealing strategy, which adaptively steals scale-up jobs to run on scale-out machines when the scale-out machines are under-utilized. To handle data placement challenges, we propose a replication-based data placement strategy that places the replicas based on the types of the jobs. Real cluster experiment and trace-driven simulation show that with our proposed strategies, Hybrid cluster can reduce the makespan and median job completion time, compared to traditional scale-out cluster with state-of-the-art schedulers.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 first provides the introduction of MapReduce programming model and the Hadoop framework. Then, it presents the system model in this dissertation.

Chapters 3, 4, 5 propose three job schedulers in different architectures of data-parallel clusters. Chapters 3 and 4 both focus on designing job schedulers to address the network bottleneck problems caused by a large amount of shuffle-heavy jobs. However, the former one outlines a network-aware scheduler in traditional datacenter network in detail, while the latter one describes a job scheduler that can efficiently exploit OCS in Hybrid-DCN. Chapter 5 illustrates the challenges in designing a hybrid scale-up/out cluster and presents corresponding job scheduling and data placement strategies to handle these challenges.

Chapter 6 describes the framework of our built simulator used to evaluate our proposed job schedulers. Chapter 7 provides an overview of the related work. Finally, Chapter 8 concludes this dissertation with future remarks.

Chapter 2

Background and System Model

In this dissertation, we use MapReduce [47] as the example use case and Hadoop [84] as its implementation. While we present our designs and results in the MapReduce setting, they generalize to any data flow based cluster computing system, like Dryad [90] and Spark [17]. The locality and network issues we address are inherent in large-scale data-parallel computing.

In this chapter, we first provide a brief introduction of MapReduce [47] programming model and then describe the Hadoop architecture, which is a popular open-source implementation of MapReduce (Section 2.1).

2.1 MapReduce Programming Model

MapReduce [47] is a popular computing model for parallel data processing on large-scale datasets. A job in MapReduce consists of the map and reduce stages, each of which consists of multiple map and reduce tasks. When each of these tasks has all its input data ready, it is assigned to a *container* on a node to execute, as shown in Figure 2.1. Each containers contains certain amount of CPU and memory resource [94]. Each map task processes one input data block and generates the intermediate

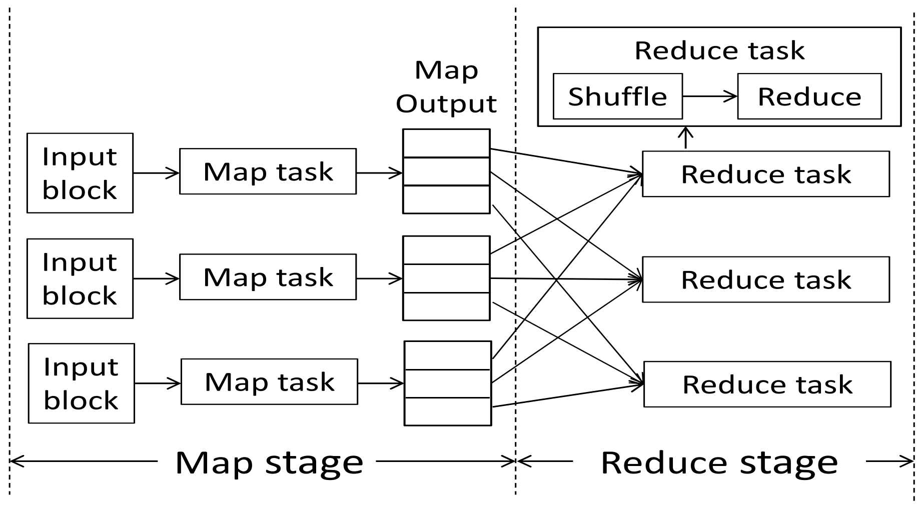


Figure 2.1: Map, shuffle and reduce phases in MapReduce [47].

key-value pairs (called map output data or shuffle data or reduce input data). Each reduce task consists of two phases: shuffle and reduce phases. In the shuffle phase, all data with the same key from different map tasks is assigned to the same reduce task, and the reduce task fetches the data through HTTP from the corresponding map tasks. Finally, each reduce task processes the input data and generates the final output.

2.2 Hadoop Overview

Hadoop [84] is a popular open-source implementation of MapReduce [47]. Yet Another Resource Negotiator (YARN) infrastructure is the resource management framework in Hadoop that provides the resources for the application running. YARN consists of one central ResourceManager (RM), a per-application ApplicationMaster (AM) and a per-node NodeManager (NM) in a cluster, as shown in Figure 2.2. As a master of the cluster, the RM primarily works together with AM and NM, and it knows the resource information (e.g., location and amount of resource) of the cluster. The RM has a Scheduler component, which determines how many and where to allocate the resource

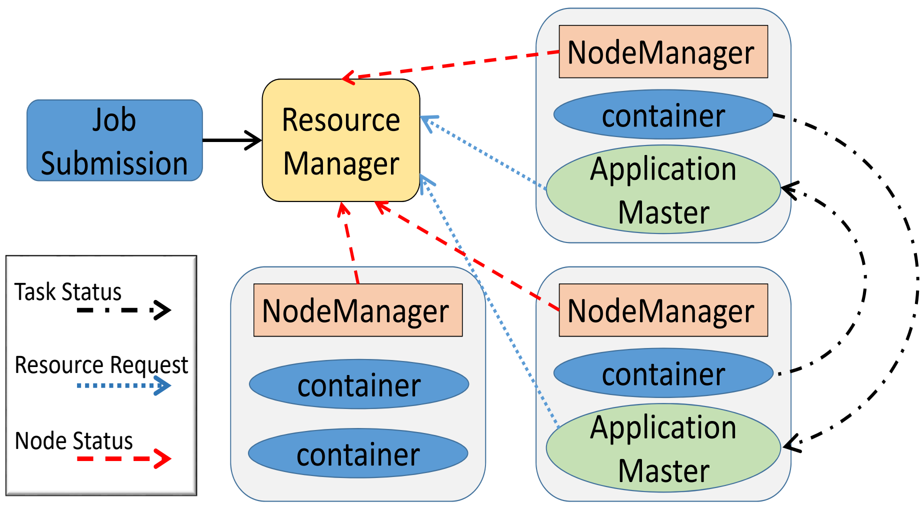


Figure 2.2: YARN framework.

containers to the applications. Each *container* incorporates certain amount of CPU and memory resources and each task in MapReduce is assigned a container. Currently, the Scheduler is pluggable to YARN. For example, Fair scheduler [62] and Capacity scheduler [26] are the most common schedulers. Each application in Hadoop has an associated AM, which is responsible for negotiating appropriate resource containers from RM and monitoring the progress of the application. The NM is YARN's per-node agent, and takes care of a compute node in a Hadoop cluster. The NM keeps track of the updates of its containers and *periodically* sends the update information (namely *heartbeat*) to the RM.

In Hadoop, when 5% (called map completion threshold) of the map tasks for a job have completed, the reduce tasks of the job can be scheduled to run on nodes. Only after a reduce task is scheduled, the shuffle phase can start. The reduce phase of the reduce tasks cannot start until all the map tasks of the job complete and the shuffle phase has transferred all the map output data needed by the reduce task to process. Overlapping the map and shuffle phases (i.e., intra-job concurrency) improves the performance in terms of throughput and execution time. Usually, the nodes that process reduce tasks (i.e., reducers) for a job are different from the nodes that process the map tasks (i.e., mappers) for the same job. As a result, almost all the shuffle data is transmitted to different nodes, generating a large amount of cross-node and even cross-rack network traffic. Part of the shuffle data may even be transmitted to different racks, generating a large amount of cross-rack network traffic.

Hadoop uses Hadoop Distributed File System (HDFS) as its primary distributed data storage system. Data is broken down into smaller blocks and stored in HDFS. To ensure fault tolerance, HDFS uses replication strategy. By default, the number of replicas is three for each block in HDFS. Conventionally, HDFS puts one replica in one node in one rack, another replica in a node in a different (remote) rack, and the third replica in a different node in the same remote rack. In other words, the three replicas are placed in two racks; one replica in a rack and two replicas in another rack.

2.3 System Model and Scheduling Problems

In this section, we define the model and present the terminology in this dissertation.

We consider a *job* consists of a certain number of *tasks*. Several previous studies [2, 50, 66, 73, 74, 94, 97] show that cluster workloads contain a large number of *recurring* jobs, whose *job characteristics*, including input/shuffle/output data sizes, job arrival time, the number of map/reduce tasks, and the map/reduce task duration, can be predicted with a small error (e.g., 6.5% [94]). Thus, for a *recurring* job, we assume that all the job characteristics above are known priori. For a *non-recurring* job, we assume that the input data size and the number of map/reduce tasks of the job are known priori.

We consider that a cluster consists of *nodes (servers)* that process tasks and *schedulers* that assign tasks to nodes. As a common assumption for data-parallel clusters in previous studies [3, 48, 94, 128], we assume that each node can run a *fixed* number of *containers*. Each container has certain amount of resources (e.g., CPU and memory) and can run one task at a time. In other words, each task is scheduled to one container on a node to run.

Recall that the job scheduling in Hadoop is performed by the ResourceManager, which manages the nodes in the cluster. Every few seconds (e.g., 1 second), the NodeManager on every node send a heartbeat to the ResourceManager, including the number of available containers on the node. The job schedulers assign tasks to the nodes while respecting the capacity constraints (i.e., the number of available containers) and task dependencies. The objectives of the schedulers often include maximizing throughput (or minimizing makespan, the time to finish a batch of jobs) and minimizing average job completion time (the time from when a job is submitted to the cluster until the last task of the job completes).

Regardless of the scheduling objectives, offline scheduling or online scheduling, many previous studies [73, 74, 76, 94, 128] indicated that no polynomial time approximation scheme is possible in scheduling the tasks in data-parallel clusters. Under the limitation of computational complexity, the job schedulers deployed in practice do not attempt to pack tasks. All known proposals rely on *heuristics* to design the job schedulers.

Chapter 3

Network-aware Scheduler in Traditional Datacenter Network Architecture

Carefully placing the map/reduce tasks can reduce cross-rack traffic and avoid cross-rack congestion. However, many schedulers [26, 62, 71, 91, 172] only focus on the scheduling of map stage. For example, Capacity [26], Fair [62] and Dominant Resource Fairness [71] aim to achieve the fairness of resource allocation (e.g., CPU, memory, storage and bandwidth) among users or jobs in the map stage. ShuffleWatcher [3] has proposed to assign the map and reduce tasks of a job onto a few racks to reduce cross-rack shuffle data traffic and avoid network congestion. Although ShuffleWatcher reduces the cross-rack traffic of the reduce tasks, it increases the cross-rack traffic to read map input data. Therefore, in this chapter, we propose a network-aware scheduler that handles the challenges of network problem of data-parallel frameworks – constraining the cross-node network load to shape the network traffic temporally, reducing cross-rack traffic and cross-rack congestion.

The remainder of this chapter is organized as follows. Section 3.1 identifies the challenges in the design of network-aware scheduler. We describe the main design of our scheduler in Section 3.2 and present our experiment evaluation in Section 3.3. Section 3.4 concludes this chapter with remarks on our future work.

3.1 Background

Several efforts [26, 62, 71] aim to achieve fairness among jobs or users for the map tasks. Fair scheduler [62] is most widely used in real clusters to achieve fairness among jobs, i.e., each job occupies approximately the same amount of resources. Dominant Resource Fairness scheduler [71] achieves a max-min fairness for multiple resources (e.g., CPU, memory and I/O). Delay scheduler [172] reduces network traffic by solving the tradeoff between fairness and map input data locality. However, the above schedulers mainly focus on the scheduling of map tasks but do not consider the shuffle phase, which is the major network traffic source in MapReduce clusters [3]. We focus on reducing the shuffle traffic and avoiding cross-rack network congestion to improve the cluster performance within certain data locality and fairness constraints.

A few previous studies consider the scheduling of reduce tasks to improve the cluster performance. Guo *et al.* [83] presented *ishuffle* that actively pushes map output data to nodes and flexibly schedules reduce tasks considering workload balance. Coupling scheduler [147] gradually launches reduce tasks based on the progress of map tasks rather than using a greedy algorithm to launch reduce tasks like Fair scheduler [62]. However, the above works do not reduce the cross-rack network traffic or avoid cross-rack network congestion. Tan *et al.* [148] formulated the reduce task scheduling that minimizes the shuffle data transfer cost to a classic stochastic assignment problem to find out the optimal reduce task placement. Jiang *et al.* [96] designed Symbiosis, which identifies and corrects unbalanced utilization of multiple resources during runtime to improve the resource utilization such as computing and network resources. Our work not only reduces the cross-rack traffic to avoid congestion but also handles cross-rack network congestion, which greatly improves the performance of MapReduce clusters.

ShuffleWatcher [3] reduces the cross-rack congestion by delaying all the reduce tasks and tries to place the map tasks into one or a fewer racks. However, it sacrifices the intra-job concurrency to achieve higher shuffle locality. Moreover, their algorithm increases the cross-rack traffic introduced by reading map input data. NAS improves both data-locality and intra-job concurrency with its three mechanisms.

Therefore, it is important to consider the shuffle phase in task scheduling to avoid and reduce network congestion without compromising cluster performance. Specifically, we identify three main challenges in designing such a scheduler below.

1. **Balancing network traffic temporally.** Previous study [120] shows that the cross-rack network bandwidth is not always fully utilized at all time. This motivates us to design a scheduler to *balance the network traffic temporally*, so that the cross-rack network is less likely to be congested. To balance the network traffic temporally, we can constrain the maximum total shuffle data generated in the cluster at a time. Thus, the first challenge is how to place tasks to constrain the maximum total shuffle data generated in the cluster at a time.

2. **Enforcing data locality for the shuffle data.** Shuffle-heavy jobs generally transfer a large amount of shuffle data across racks, resulting in high requirement of cross-rack network bandwidth. Many jobs executed simultaneously exacerbate the pressure of the need of network bandwidth. Unlike the CPU, memory, and disk resources that are easy to scale-up by deploying more hardware, network bandwidth is hard to scale-up with current hardware technology [3]. Current datacenter network architectures [5] typically provide cross-rack

bandwidth and within-rack bandwidth per node with a ratio of 5:1 to 20:1 [47, 78, 172]. Poor scheduling of reduce tasks on different nodes may lead to cross-rack network congestion, which degrades the performance. Thus, it is crucial to place the tasks to enforce certain data locality for the shuffle data, i.e., reducing the amount of shuffle traffic that is transferred cross-rack.

3. **Reducing cross-rack network congestion.** The overlap between the map and shuffle phases improves the performance. ShuffleWatcher sacrifices such overlap to avoid network congestion [3], which degrades the performance. Therefore, it is important to reduce congestion while reducing the sacrifices of intra-job concurrency to achieve better performance.

3.2 Design of NAS

In this section, we explain the methodology of our proposed scheduler. It consists of three main mechanisms to handle the challenges: map task placement (MTS), reduce task placement (CR-RTS) and network-aware shuffle delaying (CA-RTS). All the mechanisms use the shuffle data size predictor [160]. NAS considers the shuffle phase in both the map and reduce task scheduling.

3.2.1 Shuffle Data Size Predictor

Our mechanisms need to learn the shuffle data size beforehand to schedule the map and reduce tasks and distinguish shuffle-heavy and shuffle-light jobs. We utilize a predictor [160] to estimate the map output data size of each map task to be shuffled. The predictor leverages the fact that the map tasks from the same job have similar map output/input ratios. The map output/input ratio of a job is obtained from the completed map tasks for the same job. Then, the predictor extrapolates the map output data size of a map task in the job by:

$$MapOutput = (map\ output/input\ ratio) * MapInput \tag{3.1}$$

where *MapOutput* and *MapInput* are the output and input data size of the map task, respectively.

It is worth mentioning that the shuffle data size can be provided by the users, if it is known in advance, or obtained from previous runs. Many previous studies [7, 66, 94] indicate that most of the jobs in production clusters are recurring, whose characteristics can be estimated with a low error. For a newly submitted job without knowing its map output/input ratio, the map output/input ratio of the job can be initialized to 1 [3]. We call this kind of jobs as *unpredicted jobs*, otherwise, *predicted jobs*. Once a map task of the job is completed, this task's map output/input ratio can be calculated by $MapOutput/MapInput$. Then, the ratio of this job is updated by calculating the average of all the completed map tasks.

Based on the predicted shuffle data size of a job, we can classify the jobs to shuffle-heavy jobs, shuffle-medium jobs and shuffle-light jobs. For example, in the experiment in Section 3.3, we define shuffle-light, shuffle-medium and shuffle-heavy jobs as the jobs with shuffle data size smaller than 1MB, in the range of (1-100)MB and larger than 100MB, respectively.

3.2.2 Overview of NAS

We briefly introduce the overall procedure of the NAS scheduler, as shown in Algorithm 1. When a node requests for a map task, MTS is invoked to schedule the map task (lines 1-2). The scheduler keeps track of the network conditions of the cluster. When a node requests for a reduce task, the network condition is checked (lines 4 and 6). If the network is not congested (lines 4-5), CR-RTS is called to schedule the reduce task. Otherwise (lines 6-7), CR-RTS is called to schedule the reduce task.

Algorithm 1 Pseudocode of NAS scheduler, which is called when a worker node requests for a task.

```
Require: Current network condition of the rack of this worker node $NetState$
1: if request for a map task then
2:     call MTS
3: else if request for a reduce task then
4:     if $NetState < CogestionThreshold$ then
5:         call CA-RTS
6:     else
7:         call CR-RTS
8:     end if
9: end if
```

3.2.3 Map Task Scheduling (MTS)

MTS aims to balance the network load temporally and hence reduce the cross-rack congestion. Specifically, it constrains the shuffle data size generated on each node under its pre-determined threshold. We will explain how to determine the threshold later.

We exploit the algorithm in Delay scheduler [172] to sort the user, which attempts to achieve high data locality while maintaining fairness among users in resource sharing. Accordingly, MTS creates a user list based on fairness, where the users with less resource have higher priority to be allocated with resources. MTS first predicts map output data sizes of all the map tasks running on a worker node, and calculates its available space for map output based on the threshold. Next, from the user list, MTS finds a map task that has output size no larger than the available space (namely shuffle-qualified map task) and also meets the data-locality requirement (i.e., the data block of a task is stored on the same node where the task runs). MTS skips a user if the user does not have a qualified map task.

To achieve fairness between users to a certain degree, as in Delay scheduler [172], MTS sets a maximum skip count D^m. Once a user has been skipped for D^m times, the user's task can be scheduled without satisfying the data-locality or shuffle-qualified requirement. Skipping users will not greatly deviate the fairness requirement. This is because in a large cluster, thousands of tasks run in the cluster, and the containers that enable the tasks of the skipped user to meet the shuffle-qualified and data-locality requirements will be freed in a few seconds [172]. We will present this analysis later.

Algorithm 2 shows the pseudocode of the MTS mechanism. First, MTS searches the tasks of the first user in the user list and tries to find a map task that meets the shuffle-qualified and data-locality requirements. If the user has such a map task, MTS selects this map task (lines 3-4). If the user has several such map tasks, the map task for an unpredicted job has higher priority so that the job can become predicted earlier later on. Then, the map task whose map output data size is the closest to the available space is preferred so that the available shuffle data space can be fully utilized. Once the user's task is scheduled, its map skip counter is set back to 0. When MTS cannot find such a map task from the first user, if the map skip counter equals D^m, MTS identifies a shuffle-qualified map task without the data-locality in the first user (lines 7-12); otherwise, MTS skips the first user, increases its map skip counter by 1 (lines 17-18), and checks the second user in the same manner.

Without data-locality, we give a higher priority to the map tasks from small-input jobs than large-input jobs considering that large-input jobs have more input data blocks throughout the cluster and hence have a higher possibility to launch a local map task later on. Accordingly, we classify the jobs to different categories based on the input data size (first priority) and whether a job is a predicted job (second priority). Take two levels as an example, we categorize the jobs into four categories to select map tasks from as shown in lines 11-14. Note that the categorization of small-input and large-input jobs can be different from clusters to clusters. The cluster operators can define their own thresholds (the same as many other parameters in

Algorithm 2 Pseudocode for MTS.

```
Require: Initialize skip count of the $i^{th}$ user $D_i^m = 0$
         maximum number of skips $D^m$
 1: Calculate the available map output data size on the worker node.
 2: for user $i$ in the user list do
 3:   if the user has data-local and shuffle-qualified map task then
 4:     launch this map task on this node, set $D_i^m = 0$
 5:   else
 6:     if $D_i^m == D^m$ then
 7:       if we can find shuffle-qualified map tasks of this user then
 8:         launch a map task in the following order:
 9:             (1) map task from small-input unpredicted job
10:             (2) map task from small-input predicted job
11:             (3) map task from large-input unpredicted job
12:             (4) map task from large-input predicted job
13:       else
14:         launch a map task in the following order:
15:             (1) data-local map task
16:             (2) map task with the smallest map output data size
17:       end if
18:     else
19:       $D_i^m$++
20:     end if
21:   end if
22: end for
```

current Hadoop) to categorize the jobs. For example, in the experiment in Section 3.3, we classify the jobs with input data size smaller and larger than 10MB as small-input jobs and large-input jobs, respectively. In each category, we further evaluate the data transfer cost of each map task to determine the priority to select a map task. The data transfer cost is calculated by: $MapCost = \gamma * MapInputSize$, where γ is equal to 0 if the map input data is on the local node (data locality); equal to 1 if the map input data is on the local rack (rack locality); and equal to 2 if the map input data is on a remote rack (rack remote). If there are several map tasks with the same *MapCost*, we select the map task whose map output data size is the closest to the available space in order to fully utilize the available shuffle data space.

When there is no map task that is shuffle-qualified from all the users (lines 15-18), if there exist data-locality map tasks, MTS selects the one with the smallest shuffle data size since it exceeds *TrafficThreshold* the least; otherwise, MTS just selects the map task with the smallest shuffle data size among all map tasks (lines 17-19) in order to reduce map output data.

Node traffic threshold determination. Now, we explain how the threshold for the shuffle data size of each node is determined, denoted by *TrafficThreshold*. When all the containers in the cluster are assigned and freed one time, it is called one *wave* of map (reduce) tasks. All submitted map (reduce) tasks cannot be scheduled to the clusters simultaneously and hence they are scheduled through several continuous waves. The shuffle data transfers of the tasks in one wave are conducted in approximately the same time. We set a threshold on each node for two purposes.

- First, it avoids scheduling many map tasks that generate large shuffle data on each node, which balances the cross-node network load.
- Second, it avoids scheduling many map tasks that generate large shuffle data simultaneously in the cluster (i.e., in one wave), which potentially constrains

the network traffic generated in the cluster at a time and hence avoids cross-rack network congestion.

We assume that $\{J_1, J_2, ..., J_n\}$ are the n submitted jobs currently in the cluster. Job J_i has S_i shuffle data size and contains K_i map tasks. We assume that in the cluster, there are N nodes, each of which has m containers and hence there are Nm containers in total. The map tasks generate $\sum_{i=1}^{n} S_i$ shuffle data size in total in the cluster. Then, $\sum_{i=1}^{n} K_i$ map tasks are processed in $\sum_{i=1}^{n} K_i/Nm$ waves. We divide the total shuffle data of all the jobs evenly into several waves. The average size of shuffle data generated in each wave is:

$$\frac{\sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} K_i/Nm} = \frac{Nm\sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} K_i} \tag{3.2}$$

Keeping approximately the same amount of shuffle data in each wave in the cluster prevents scheduling many map tasks with large shuffle data sizes at the same time and hence avoids cross-rack congestion.

To achieve a balanced cross-node traffic, we set the threshold for the shuffle data size on each node *TrafficThreshold* as the average size of shuffle data generated on each node in each wave:

$$\frac{Nm\sum_{i=1}^{n} S_i}{N\sum_{i=1}^{n} K_i} = \frac{m\sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} K_i} \tag{3.3}$$

TrafficThreshold is updated periodically. The cluster operators can change *TrafficThreshold* dynamically (the same as many other parameters in current Hadoop) that serves their own clusters more accurately. For example, in some clusters, there are fewer shuffle-heavy jobs and then *TrafficThreshold* can be set to a smaller value.

Analysis of the map skip counter strategy. We analyze the probability of launching a map task with the data-locality and shuffle-qualified constraints. When a worker node requests for a map task, we assume that user i is the first one in the user list and it has m_i submitted jobs denoted by $\{J_1^i, J_2^i, ..., J_{m_i}^i\}$. Let p_J be the fraction of nodes that have job J's required data and q_J be the probability that the map tasks of job J are shuffle-qualified. Note that q_J is easy to adjust by the cluster operators by setting an appropriate *TrafficThreshold*. Then, the probability that user i cannot launch a map task that meets the data-locality and shuffle-qualified requirements after skipping D^m times is $\prod_{k=1}^{m_i}(1 - p_{J_k^i} q_{J_k^i})^{D^m}$. This probability decreases exponentially as D^m decreases. For example, assume that a user has 3 jobs, 10% of nodes have the jobs' input data (i.e., $p_j = 0.1$) and the probability that the map tasks of the jobs are shuffle-qualified is $q_J = 0.5$. Then, this user has a 78.5% probability to launch a map task within 10 skips and a 99.8% probability to launch a map task within 40 skips. In Facebook cluster [172], 27 containers are freed every second on average, which means that there is 99.8% probability to take less than 2 seconds for the user to launch a data-locality and shuffle-qualified map task.

3.2.4 Congestion-avoidance Reduce Task Scheduling (CA-RTS)

In this section, we introduce the CA-RTS mechanism, which aims to avoid traffic congestion and reduce the cross-rack network traffic in reduce task scheduling. CA-RTS incorporates a cross-rack traffic reduction method and an adaptive map completion threshold method.

We define a threshold of desired upper bound of network utilization *Cogestion-Threshold* (e.g., 90% of cross-rack bandwidth is used). As in [3], we utilize some network monitor tools (e.g., NetHogs) to monitor the cross-rack network load in the cluster. When a worker node requests for the next reduce task to process, if *Coges-*

tionThreshold is not yet reached, it means that the cross-rack network is not congested and CA-RTS is used for reduce task scheduling. Otherwise, CR-RTS (Section 3.2.5) is used for reduce task scheduling.

Cross-rack traffic reduction method. It is indicated in [14] that for a job that has evenly distributed map output data on several racks, the best placement of reduce tasks to avoid cross-rack congestion on one rack is to evenly distribute the reduce tasks among these racks. Therefore, for each job, keeping the distribution of its reduce tasks the same as the distribution of its shuffle data among the racks can minimize cross-rack traffic and hence avoid cross-rack congestion because placing more reduce tasks of a job on a rack may congest its downlink, while placing fewer reduce tasks of this job on a rack may congest its uplink. That it, for a job, if $x\%$ (called *MapOuputPortion*) of its total map output data is generated in rack R_i, scheduling $x\%$ of its total reduce tasks (denoted by *TotalReduceNum*) in rack R_i can minimize the cross-track network traffic for shuffle data transfer of the job. We define:

$$ReduceNum = TotalReduceNum * MapOutputPortion. \tag{3.4}$$

Algorithm 3 Pseudocode for CA-RTS.

```
1: Select a user from the user list based on fairness.
2: Launch reduce task from a job that satisfies map completion threshold in the
   following order (a job with delayed or $MapProgressRate = 100\%$ has higher
   priority in the same category):
3:         (1) Shuffle-heavy jobs whose $ReduceNum$ is not reached,
4:         (2) Shuffle-medium jobs whose $ReduceNum$ is not reached
5:         (3) Shuffle-light jobs whose $ReduceNum$ not reached
6:         (4) Shuffle-light jobs whose $ReduceNum$ is reached
7:         (5) Shuffle-medium jobs whose $ReduceNum$ is reached
8:         (6) Shuffle-heavy jobs whose $ReduceNum$ is reached
```

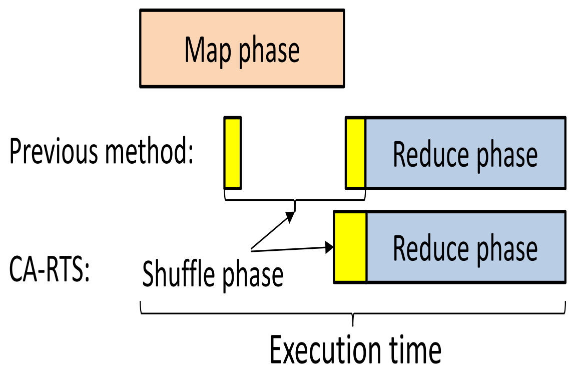
When CA-RTS handles a reduce task request from a worker node on a rack R_i, it first predicts the shuffle data size (*ShuffleSize*) (as introduced in Section 3.2.1), and then calculates *MapOuputPortion* and *ReduceNum* of each job on rack R_i.

Algorithm 3 shows the pseudocode of CA-RTS. From the first user in the user list (line 1), CA-RTS selects the reduce tasks from the jobs that run fewer reduce tasks than *ReduceNum* on rack R_i (lines 3-5). In addition, CA-RTS also considers i) whether it is delayed in CR-RTS, ii) whether the percentage of completed map tasks of a job, *MapProgressRate*=100%, and iii) *ShuffleSize* to achieve high performance. CA-RTS gives a higher priority to the reduce tasks marked as "delayed" by CR-RTS in order not to delay some reduce tasks for too long. Next, CA-RTS gives a higher priority to the reduce tasks of the jobs with fully completed map tasks (i.e., *MapProgressRate*=100%) in order to start them as early as possible. Finally, CA-RTS prefers the reduce tasks from the jobs with larger shuffle data sizes in order to fully utilize available bandwidth when the cross-rack network is not congested.

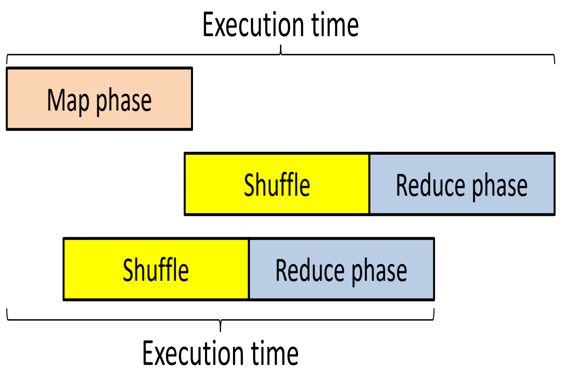
If CA-RTS cannot find the reduce tasks from the jobs that have the number of reduce tasks less than *ReduceNum* on rack R_i, CA-RTS then gives a higher priority to the reduce tasks from the jobs with smaller shuffle data size because such tasks cause a smaller amount of cross-rack traffic (lines 6-8). As a result, CA-RTS reduces the cross-rack traffic generated from shuffle data transfer.

Adaptive map completion threshold method. In MapReduce, when the percentage of completed map tasks of a job reaches the map completion threshold (e.g., 5%), the reduce tasks of the job can be scheduled, and only after all shuffle data of the job is transferred, the reduce tasks can start running. Transferring a smaller amount of shuffle data takes a shorter time period and vice versa.

As shown in Figure 3.1(a), for a shuffle-light job, the shuffle data is transferred in a short time and early scheduling of reduce tasks (the top scheduling in Figure 3.1(a)) may cause two problems. First, it may increase the network bandwidth competition with shuffle-heavy jobs though its delayed transfer won't affect the job running performance. Second, it occupies the computing resource even no shuffle data is transferred



(a) Shuffle-light jobs



(b) Shuffle-heavy jobs

Figure 3.1: Demonstration of shuffle-light and shuffle-heavy jobs.

since the transfer is done in a short time. On the other hand, a shuffle-heavy job takes a long time for the shuffle data to transfer. A later data transferring start time may increase their execution time and reduces the overlap between the map and shuffle phases (the top scheduling in Figure 3.1(b)).

Therefore, the data transfer and reduce task scheduling for shuffle-light jobs can start later, while the data transfer for shuffle-heavy jobs can start earlier, as shown in the bottom scheduling manners in Figures 3.1(a) and 3.1(b). Based on this rationale, we propose our adaptive map completion threshold method based on shuffle data size to avoid traffic congestion and fully utilize resource. That is, if a job has a larger shuffle data size, it has a smaller threshold and vice versa. Assume the maximum and minimum shuffle data size of the jobs in the cluster is S_{max} and S_{min}, respectively. We set the smallest and the largest threshold to T_{min} and T_{max}, which are the thresholds for the jobs with the largest and the smallest shuffle data size, respectively. If a job has a shuffle data size of S, its threshold equals:

$$Threshold = \frac{T_{min} - T_{max}}{S_{max} - S_{min}} * S + \frac{T_{max}S_{max} - T_{min}S_{min}}{S_{max} - S_{min}}. \tag{3.5}$$

As a result, jobs with a smaller shuffle data size have higher thresholds and vice versa. This prevents shuffle-light jobs from competing bandwidth with shuffle-heavy and shuffle-medium jobs while enabling them to finish shuffle data transfer when map tasks complete. Also, it enables to more fully utilize the resources by releasing the occupied but idle containers of reduce tasks from shuffle-light jobs.

3.2.5 Congestion-reduction Reduce Task Scheduling (CR-RTS)

In this section, we introduce the CR-RTS mechanism, which aims to mitigate the cross-rack network congestion caused by shuffle data transfer.

If *CogestionThreshold* is reached, the bandwidth is highly utilized. Delaying scheduling all reduce tasks to reduce the congestion sacrifices intra-job concurrency and compromises performance. To reduce the network congestion while maintaining the overlap between the map and shuffle phases, CR-RTS schedules reduce tasks and map tasks that will not generate a large amount of shuffle data traffic. Specifically, CR-RTS has three strategies. First, it selects the shuffle-light jobs to schedule and delays scheduling the reduce tasks of shuffle-heavy and shuffle-medium jobs until the network is not congested. Second, CR-RTS stops scheduling the map tasks of shuffle-heavy and shuffle-medium jobs. Then, the map completion threshold cannot be reached and the shuffle data of these jobs will not be transferred.

Algorithm 4 Pseudocode for CR-RTS.

```
Require: Initialize skip count of the $i^{th}$ user $D_i^r = 0$
         maximum number of skips $D^r$
 1: for user $i$ in the user list do
 2:   if $D_i^r < D^r$ then
 3:     if this user has shuffle-light jobs then
 4:       Select a reduce task from shuffle-light jobs, set $D_i^r = 0$
 5:     else
 6:       $D_i^r$++ and skip this user
 7:     end if
 8:   else
 9:     Select a reduce task from any jobs
10:   end if
11: end for
```

Algorithm 4 shows the pseudocode of CR-RTS. Again, there is a sorted user list created based on Delay scheduler. CR-RTS checks the users in the user list in the top-down manner. From the first user, CR-RTS tries to find a reduce task of shuffle-light job to schedule and delays the reduce tasks of shuffle-medium and shuffle-heavy jobs (lines 1-8). If the first user does not have a reduce task from shuffle-light jobs, CR-RTS searches the next user until it finds a matched reduce task. The reduce skip counter is handled in the same manner as the map skip counter. For the reduce tasks of shuffle-heavy and shuffle-medium jobs, each reduce task has a delay tag. CR-RTS changes the delay flag to "delayed". These delayed tasks will have a higher priority to be scheduled when the network is not congested as explained in Section 3.2.4. Further, CR-RTS notifies MTS not to schedule shuffle-heavy and shuffle-medium map tasks until the network is not congested.

Analysis of the reduce skip counter strategy. We assume that user i has m_i submitted jobs. Let f_J denote the probability that job J is a shuffle-light job. Therefore, the probability that user i does not have a shuffle-light job is $(1 - f_J)^{m_i}$. Thus, the probability that top u users in the user list do not have a shuffle-light job is $(1 - f_J)^{m_i u}$. Take the Facebook trace [32] as an example. According to our definition of shuffle-light jobs in Section 3.3, we find that 68.7% of the jobs (f_J=0.687) are shuffle-light jobs. Assume that each user has only one job ($m_i = 1$). Therefore, skipping 3 users ($u = 3$) has a 97% probability of launching a shuffle-light job and skipping 5 users ($u = 5$) has a 99.7% probability of launching a shuffle-light job. When the cross-track network is congested, there should be a large amount of users and jobs in the cluster. Hence, it is very likely to launch a shuffle-light job. Then, CR-RTS needs to skip only a few users or even no users if a user has several jobs, which maintains a high fairness.

3.2.6 Complexity of NAS

Similar to current schedulers [62, 172], NAS has very simple computations such as finding shuffle-qualified and data-local map tasks, which are in $\mathcal{O}(n)$ complexity (n is the number of jobs in the list). These computations are quite simple and generate negligible overheads. Also, the monitor is a low-overhead monitor. Therefore, the NAS has at least the same scalability as the state-of-the-art schedulers [62, 172].

3.3 Performance Evaluation

In this section, we evaluate NAS in comparison with other schedulers through trace-driven simulation. We also implemented our scheduler in Hadoop on a real cluster for performance evaluation.

3.3.1 Facebook Trace and Experimental Environment

Trace-driven simulation. We built an event-based simulator (details in Chapter 6) as in [48, 94, 128] to evaluate the performance. We used the Facebook day-long workload FB-2010 trace [32] in our simulation. The trace provides detailed information of 24442 jobs. We considered the small-input jobs and large-input jobs as the jobs with input data size smaller and larger than 10MB, respectively. We considered the shuffle-light jobs, shuffle-medium jobs and shuffle-heavy jobs as the jobs with shuffle data size smaller than 1MB, in the range of (1-100)MB, and larger than 100MB, respectively. Figure 3.2 shows the percentage of jobs of each type in the workload.

In the simulation, we set the number of users to 200 and the number of nodes to 600 in the cluster, which are consistent with the Facebook cluster reported in [32, 172]. The job arrival time strictly follows the trace. Since there is no user information in the trace, we assigned each job randomly to a user. In the 600-node cluster,

Job type	Percentage
Small-input	50.02%
Large-input	49.98%
Shuffle-light	68.70%
Shuffle-medium	12.58%
Shuffle-heavy	18.82%

Figure 3.2: Distribution of each job type in the simulation.

we assume that there are 30 racks, each of which has 20 nodes. Each node has 6 containers [172]. The block size was set to 128MB [172]. The replication factor was set to 3. In a commercial cluster, it is common that the cross-rack bandwidth for each node is much lower than the within-rack bandwidth for each node [16, 47, 172]. Like [3], we set the cross-rack bandwidth to 1Gbps. We set the within-rack bandwidth for each node to 250Mbps, so that the ratio of within-rack and cross-rack bandwidth for each node follows 5:1. Typical oversubscription ratio ranges from 5:1 to 20:1 [16, 47, 172] and with a higher oversubscription ratio than the setting, NAS can achieve more performance improvement than our reported experimental results.

We compared NAS with Fair scheduler (Fair) [62], Delay scheduler (Delay) [172] and ShuffleWatcher based on the Delay scheduler (SW-delay) [3]. Fair is the default and the state-of-the-art scheduler for Hadoop. It achieves fairness among jobs, i.e., each job occupies approximately the same amount of resources. Delay is built upon the Fair scheduler and is another default scheduler in Hadoop. It achieves high data locality of map tasks by delaying the jobs that cannot launch a local map task. ShuffleWatcher can be based on either Fair or Delay. We simulated ShuffleWatcher on top of Delay since it achieves the best throughput in [3]. For both ShuffleWatcher and NAS, we set the network congestion threshold to 80%. We set the maximum map (reduce) skip count $D^m = D^r = 135$ and set $T_{min} = 0.2$, $T_{max} = 0.5$. For Fair, Delay and SW-delay, they do not have the adaptively map completion threshold method, and their map completion threshold was set to 0.2, i.e., when 20% of the map tasks of a job complete, its reduce tasks can be scheduled.

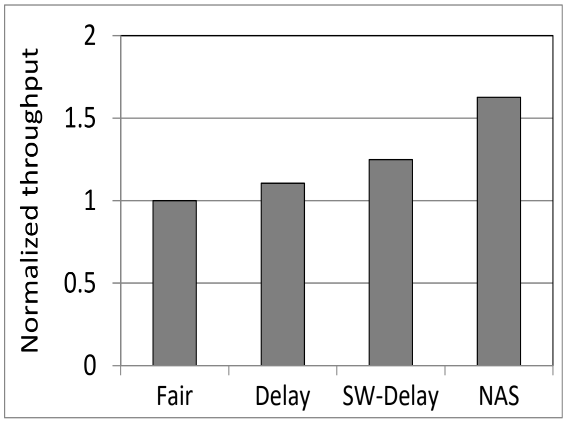
Real cluster experiment. We generated a workload consisting of 200 jobs using the Facebook workload synthesized execution framework [32] and the distribution of job types is the same as shown in Figure 3.2. As in [172], the job arrival time of these 200 jobs follows an exponential distribution with a mean of 14 seconds, which makes the process of all submissions 45 minutes long. We implemented NAS in Hadoop and conducted the evaluation on a 40-node cluster in CloudLab [41]. The 40 nodes were organized in 8 racks with interconnection of 1Gbps Ethernet. Each rack contains 5 nodes and each node has 1Gbps Ethernet interconnect, resulting in a 5:1 oversubscription ratio. The number of containers on each node was set to 16.

In order to implement NAS in Hadoop, we modified the source codes including ResourceManager, ApplicationMaster, RMAppManager, AMLauncher, and etc. We compared NAS with Fair, Delay, and SW-delay schedulers. As in [172], rather than using the maximum skip count D^m and D^r, we set a maximum wait time of 5 seconds, i.e., a user cannot be skipped more than 5 seconds to launch tasks. Other settings are the same as the simulation and other configuration parameters of Hadoop are the same for all the methods.

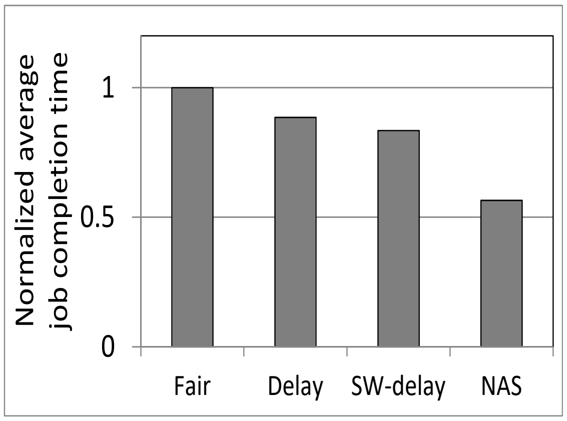
3.3.2 Real Cluster Results

In this section, we will present the results of 20 runs in the real cluster experiments. In order to show the results more clearly, *we normalize the experimental results by the results of Fair scheduler.*

We first compared the throughput of different schedulers, which is calculated by the number of jobs (i.e., 24442) divided by the total time to run all the jobs. Figure 3.3(a) shows the normalized throughput of different schedulers in the real cluster experiment. We see that NAS achieves improvement over Fair, Delay and SW-delay



(a) Throughput.



(b) Average job completion time.

Figure 3.3: Results of real cluster experiments.

with 62.5%, 47.0% and 30.2% higher throughput.

We then measured the average job completion time of all the jobs, which is calculated by the sum of job completion time of all the jobs divided by the total number of jobs. Figure 3.3(b) shows the normalized average job completion time of different schedulers in the real cluster experiment. We see that the average job completion time of NAS is 44.6%, 36.1%, and 32.3% shorter than Fair, Delay and SW-delay, respectively.

SW-delay and NAS outperform the Fair and Delay scheduler because they reduce the cross-rack shuffle network traffic, which greatly expedites shuffle data transfer. NAS produces higher throughput and lower average job completion time than SW-delay. This is because i) NAS balances the shuffle data transfer load on each node, while SW-delay does not, ii) NAS pro-actively avoids cross-rack congestion by adjusting the map completion threshold for different jobs based on their shuffle data sizes, and iii) SW-delay sacrifices the map and reduce phase overlap to achieve higher shuffle locality (i.e., most shuffle data of a reduce task is located on the same rack where this reduce task is run). When the network is saturated, SW-delay delays all the reduce tasks including the shuffle-light jobs, while NAS does not delay the shuffle-light jobs,

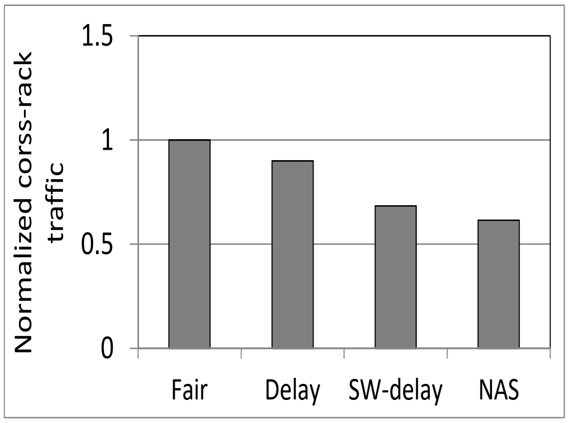


Figure 3.4: Cross-rack traffic.

which are the majority of the jobs in the workload (i.e., 60%).

Figure 3.4 shows the normalized cross-rack shuffle data traffic in the real cluster experiment, which is measured by the total amount of data transferred cross-rack in the cluster. We see that SW-delay and NAS produce less cross-rack traffic than Fair and Delay. SW-delay and NAS try to reduce cross-rack shuffle data traffic upon the cross-rack network congestion while Delay and Fair do not address cross-rack network congestion caused by shuffle data transfer.

Figure 3.5 shows the total number of occurrences of cross-rack congestions in the real cluster experiment. We see that NAS and SW-delay generate fewer cross-rack congestions than Fair and Delay. This is because when the network is close to saturation, SW-delay delays the scheduling of all reduce tasks and NAS delays the scheduling of reduce tasks from shuffle-medium and shuffle-heavy jobs to reduce congestion, while Fair and Delay do not have mechanisms to deal with the network congestion. This figure indicates the effectiveness of NAS on reducing cross-rack congestion.

We further evaluate how NAS improves the performance of jobs with various shuffle data size. Figure 3.6 shows the throughput improvement (i.e., $\frac{with\ NAS}{without\ NAS}$) for shuffle-light, shuffle-medium, and shuffle-heavy jobs in the real cluster experiment. We

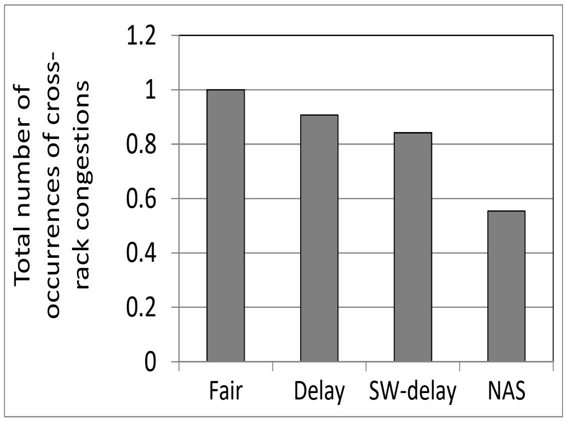


Figure 3.5: The number of occurrences of cross-rack congestions.

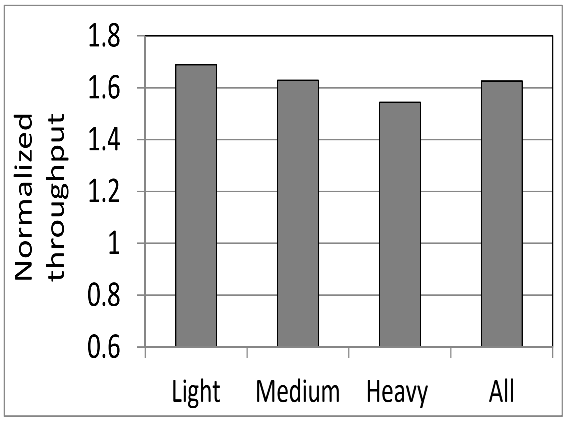


Figure 3.6: Throughput improvement for different jobs.

see that in the real cluster experiments, shuffle-light jobs achieve the highest throughput improvement, followed by shuffle-medium and then shuffle-heavy jobs. Without NAS, the performance of shuffle-light jobs are severely degraded by shuffle-heavy jobs when the network is congested, since the shuffle-heavy jobs occupy the container resources and do not release the container resources for a long time. NAS significantly reduces the network congestion and hence reduces the impact from shuffle-heavy jobs on shuffle-light jobs, which results in the most throughput improvement for shuffle-light jobs.

We also investigated how NAS performs as the map completion thresholds T_{min} and T_{max} change. Figure 3.7 shows the throughput of NAS for varying T_{min} and

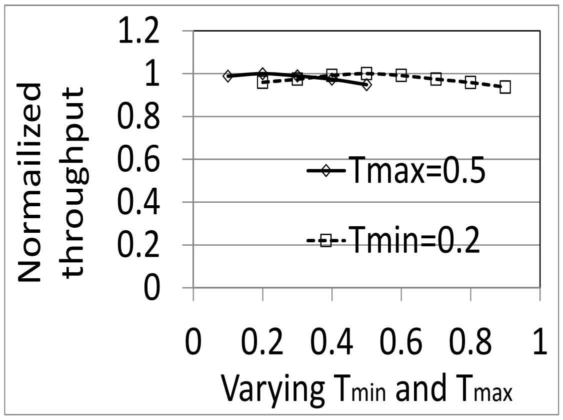


Figure 3.7: Throughput of NAS for varying T_{min} and T_{max}.

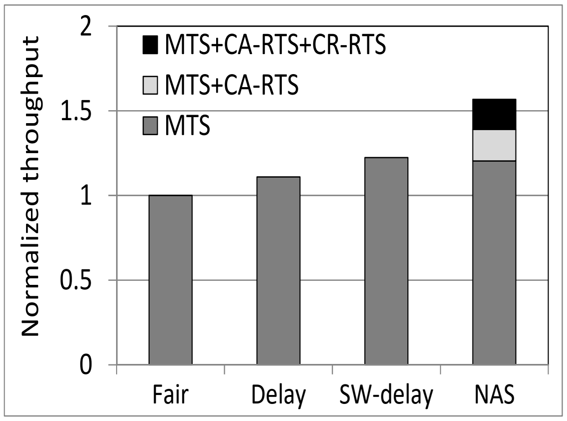
T_{max} in the real cluster experiment. For the curve $T_{min} = 0.2$, it means that T_{min} is fixed to 0.2 and T_{max} is varied from 0.2 to 0.9. For the curve $T_{max} = 0.5$, T_{max} is fixed to 0.5 and T_{min} is varied from 0.1 to 0.5. We normalized the results to the result when $T_{min} = 0.2$ and $T_{max} = 0.5$. We see that when T_{min}/T_{max} is too small or too large, the throughput is decreased. This is because (i) a too small value of T_{min}/T_{max} results in an early start of reduce tasks and hence waste of container resources; and (ii) a too large value of T_{min}/T_{max} results in a smaller overlap of map and reduce phase and hence longer job completion time. With the adaptive map completion threshold method, our scheduler can improve the throughput by up to 5%, comparing to the schedulers without this adaptive method (i.e., $T_{min} = T_{max}$). Therefore, it is important to select the correct T_{min} and T_{max} for a cluster based on its workloads. For a given cluster, its workloads generally remain similar [94], so the parameters do not need to be always changed once they are determined.

3.3.3 Trace-driven Simulation Results

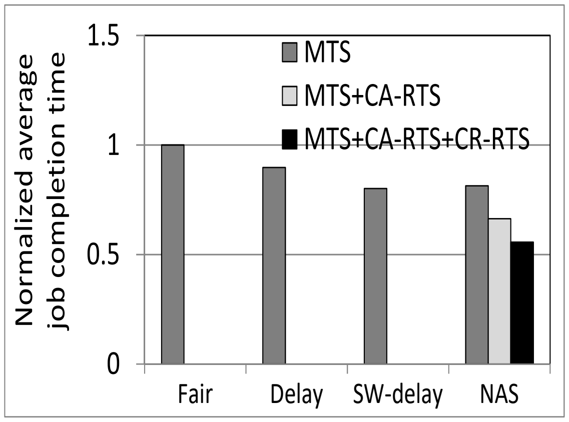
In this section, we will present the results of 20 runs in the trace-driven simulation. In order to show the results more clearly, *we normalize the experimental results by the results of Fair scheduler*. In the simulation, we also show the performance of

MTS, MTS+CA-RTS and MTS+CA-RTS+CR-RTS (i.e., NAS) in order to show the effectiveness of different mechanisms in NAS.

Figure 3.8(a) shows the normalized throughput of different schedulers in the simulation. We see that NAS achieves improvement over Fair, Delay, and SW-delay with 56.9%, 41.2%, 28.0% higher throughput. Figure 3.8(b) shows the normalized average job completion time of different schedulers. We see that the average job completion time of NAS is shorter than Fair, Delay and SW-delay by 44.3%, 38.0%, and 30.5%, respectively. Both the results of throughput and average job completion time are consist with the results in the real cluster experiments due to the same reasons.



(a) Throughput.



(b) Average job completion time.

Figure 3.8: Results in the simulation.

In addition, from Figures 3.8(a) and 3.8(b), we see that MTS, CA-RTS, and CR-RTS all show great impact on the improvement of throughput and completion time in NAS. The experimental results indicate that NAS outperforms other schedulers on improving the throughput and average job completion time, which demonstrates the effectiveness of the mechanisms in NAS.

Figure 3.9 shows the normalized cross-rack traffic in the simulation, which is consistent with the results in the real cluster experiments due to the same reasons. From Figure 3.9, we see that MTS achieves similar cross-rack traffic as Delay, since

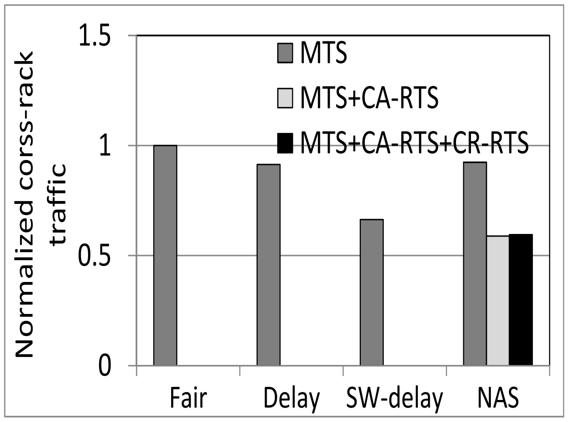


Figure 3.9: Cross-rack traffic.

both MTS and Delay attempt to guarantee data locality for the map tasks. Using CA-RTS with MTS, the cross-rack traffic is decreased, since CA-RTS places the reduce tasks proportional to the map output distribution to minimize the cross-rack traffic. The additional use of CR-RTS does not further decrease the cross-rack traffic. This is because CR-RTS does not reduce cross-rack traffic in the system and it actually shapes the cross-rack traffic (i.e., delays the transfer of heavy shuffle data until the network is uncongested) to reduce cross-rack congestion, which improves the throughput and average job completion time, as shown in Figures 3.8(a) and 3.8(b).

In order to show the degree of the overlap sacrifice because of the delay scheduling for the reduce tasks, we draw Figure 3.10, which shows the average *MapProgressRate* for all the jobs when the first reduce tasks of these jobs are scheduled. The *MapProgressRate* of a job equals the fraction of completed map tasks of the job. A lower *MapProgressRate* means more overlap between the map and shuffle phases. We see that the results follows SW-delay>NAS>Delay≈Fair. Since both NAS and SW-delay delay the reduce task scheduling when the network is congested, their overlaps between map and reduce phases are smaller than those of Fair and Delay. Moreover, NAS has a lower average *MapProgressRate* than SW-delay. This is because when the network is congested, SW-delay delays all the reduce tasks, while NAS keeps

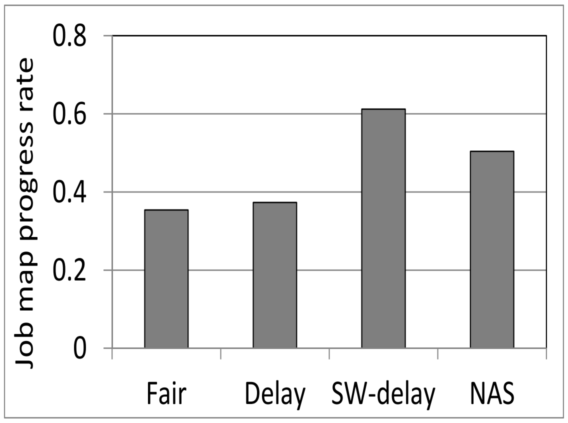


Figure 3.10: Job map progress rate when the first reduce task is scheduled.

assigning shuffle-light jobs rather than delaying them, resulting in a lower average *MapProgressRate*.

Figure 3.11 shows the percentage of nodes that have map output data size higher than *TrafficThreshold* (i.e., congested nodes). We see that NAS has a very small percentage of congested nodes, while Fair, Delay and SW-delay have relatively higher percentage of congested nodes. In NAS, MTS tries to constrain each node's map output data below *TrafficThreshold*. Some nodes become congested because their map skip counter reaches the maximum value and then their map tasks are scheduled without the shuffle-qualified or data-locality constraint. The figure shows that this situation happens only a few times, which means that the cross-node traffic are constrained below *TrafficThreshold* most of the time. In Fair and Delay schedulers, the scheduling of map tasks is only based on fairness without considering cross-node traffic, leading to a large number of congested nodes. SW-delay generates even more congested nodes than Fair and Delay because SW-delay schedules map tasks on the containers when the network is congested, which may result in reading remote input data and hence more congested network. This figure demonstrates the effectiveness of MTS in NAS on constraining cross-node traffic.

Figure 3.12 shows the total number of occurrences of cross-rack congestions in the

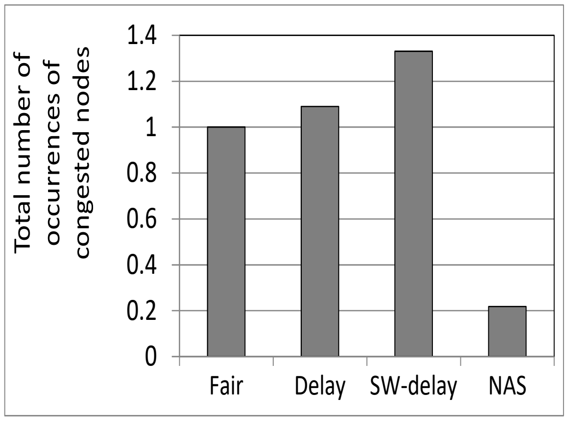


Figure 3.11: The number of occurrences of cross-node congestions.

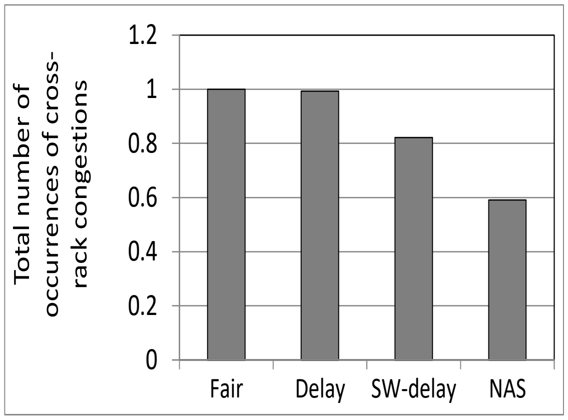


Figure 3.12: The number of occurrences of cross-rack congestions.

simulation. We see that NAS and SW-delay generate fewer cross-rack congestions than Fair and Delay. This is because when the network is close to saturation, SW-delay delays the scheduling of all reduce tasks and NAS delays the scheduling of reduce tasks from shuffle-medium and shuffle-heavy jobs to reduce congestion, while Fair and Delay do not have mechanisms to deal with the network congestion. This figure indicates the effectiveness of NAS on reducing cross-rack congestion.

Recall that setting of *TrafficThreshold* in MTS and the adaptive map completion threshold method in CA-RTS help avoid the cross-rack network congestion. To verify this, we tested the performance of NAS without the setting of *TrafficThreshold* in MTS (*w/o TT*), NAS without the adaptive map completion threshold method (*w/o*

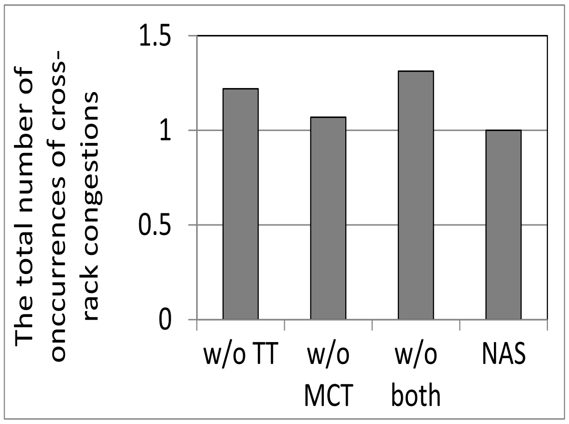


Figure 3.13: The total number of occurrences of cross-rack congestions breaking down by different mechanisms.

MCT) and NAS without the both methods (*w/o both*). Figure 3.13 shows the total number of occurrences of cross-rack congestions during the entire experiment time of these methods compared with NAS (normalized by NAS). The total numbers of occurrences of *w/o TT*, *w/o MCT*, *w/o both*, and NAS are 1.22, 1.07, 1.31, and 1, respectively. This result demonstrates that the setting of *TrafficThreshold* and the adaptive map completion threshold method can help avoid cross-rack network congestion.

The performance of NAS may vary if the number of jobs per user changes. Figure 3.14 shows the normalized throughput of NAS with different number of users in the cluster comparing with Fair with 200 users. Note that when the number of users decreases, the number of jobs per user increases. We see that NAS achieves higher throughput with fewer users. This is because each user has more jobs when the number of users decreases, which provides more choices for scheduling the appropriate task to the container and hence decreases the needs to skip a user in task searching in MTS and CR-RTS.

We further evaluate how NAS improves the performance of jobs with different shuffle data size. Figure 3.15 shows the throughput improvement (i.e., $\frac{with\ NAS}{without\ NAS}$)

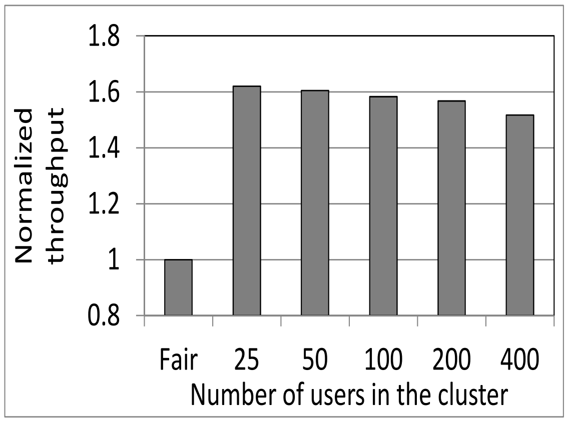


Figure 3.14: Normalized throughput for different number of users.

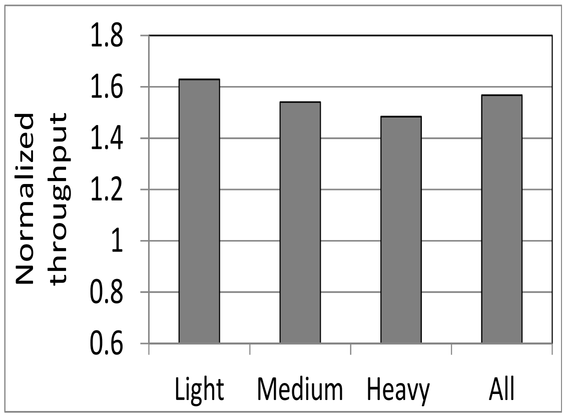


Figure 3.15: Throughput improvement for different jobs.

for shuffle-light, shuffle-medium, and shuffle-heavy jobs in the simulation. We see that in the simulation, shuffle-light jobs achieve the highest throughput improvement, followed by shuffle-medium and then shuffle-heavy jobs. The reasons are the same as the reasons of Figure 3.6 in the real cluster experiment.

We also investigated how NAS performs as the map completion thresholds T_{min} and T_{max} change. Figure 3.16 shows the throughput of NAS for varying T_{min} and T_{max} in the simulation. For the curve $T_{min} = 0.2$, it means that T_{min} is fixed to 0.2 and T_{max} is varied from 0.2 to 0.9. For the curve $T_{max} = 0.5$, T_{max} is fixed to 0.5 and T_{min} is varied from 0.1 to 0.5. We normalized the results to the result when $T_{min} = 0.2$ and $T_{max} = 0.5$. We see that when T_{min}/T_{max} is too small or too large,

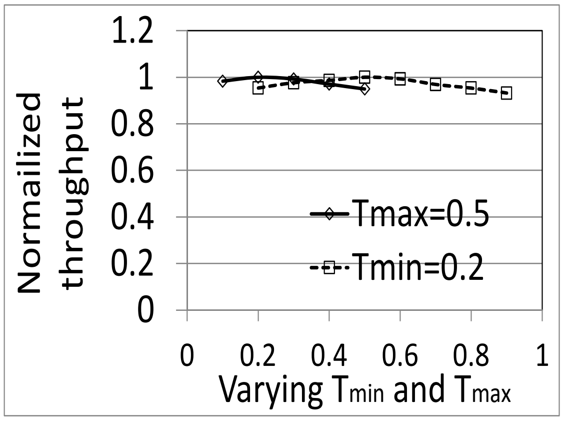


Figure 3.16: Throughput of NAS for varying T_{min} and T_{max}.

the throughput is decreased. The reasons are the same as the reasons of Figure 3.7 in the real cluster experiment.

3.4 Summary

Shuffle data transfer is the dominant source of cross-node/rack network traffic, which greatly affects the performance of MapReduce clusters. However, few previous schedulers handle the network traffic caused in the shuffle phase. Therefore, we presented a new network-aware MapReduce scheduler (NAS). NAS consists of three mechanisms: Map Task Scheduling (MTS), Congestion-reduction Reduce Task Scheduling (CR-RTS) and Congestion-avoidance Reduce Task Scheduling (CA-RTS). These three mechanisms jointly work to constrain the cross-node network traffic and reduce cross-rack network traffic. We implemented NAS in Hadoop on a supercomputing cluster. Through large-scale trace-driven simulation based on the Facebook workload and real Hadoop cluster experiment, we showed that NAS greatly improves cluster throughput and reduces average job completion time compared with the Fair, Delay and ShuffleWatcher schedulers. In the future, we will extend NAS to schedule the jobs considering the dependency between jobs in which, a job's output is the input of another job. For example, we will consider the placement of dependent jobs to reduce the network traffic generated from input data reading among the dependent jobs.

Chapter 4

Job Scheduler in Hybrid Electrical/Optical Datacenter Network Architecture

Recently, several previous studies propose hybrid electrical/optical datacenter network (in short *Hybrid-DCN*) designs [30, 63, 134, 162], which augment the traditional EPS datacenter network with an on-demand *rack-to-rack* network using the OCS. In Hybrid-DCN, OCS is used only for large data transfers between racks so that the overhead of μs-to-ms switch-reconfiguration time is negligible. In order to utilize Hybrid-DCN efficiently, job schedulers for data-parallel frameworks must keep pace to meet the needs of such hybrid networks. In this chapter, we aim to design a job scheduler that can efficiently leverage OCS in Hybrid-DCN to improve job performance.

The remainder of this chapter is organized as follows. Section 4.1 introduces the background in this chapter, including properties of OCS and Hybrid-DCN, desired properties of Hybrid-DCN and opportunities. Section 4.2 describes the main design of JobPacker. Section 4.3 presents the performance evaluation. Section 4.4 concludes

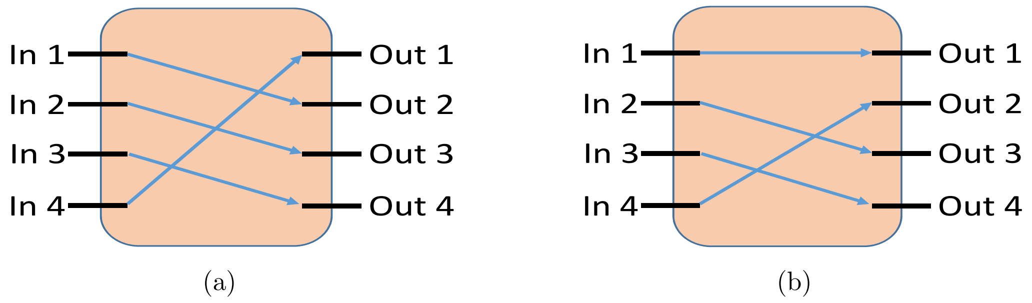


Figure 4.1: An example of 4*4 OCS. (a) Input port 1 can only send data via OCS to output port 2. (b) To send data from input port 1 to output port 1, we need to reconfigure OCS.

this chapter with remarks on our future work.

4.1 Background

4.1.1 Optical Circuit Switches

OCS can provide many substantial advantages over traditional EPS [30, 63, 162]. First, OCS does not require transceivers that convert between light and electricity, which provides a significant cost saving [63]. Second, OCS can provide up to 100Gbp-s per port with much less power than EPS [63, 89]. An OCS consumes about 240 mW/port, while a 10GigE Ethernet EPS consumes 12.5 W/port. The slow reconfiguration is the only problem of OCS. Figure 4.1 shows an example of 4*4 OCS and OCS reconfiguration to build the connection between an input port and an output port. OCS can connect any input port to any output port, but one input port can be connected to only one output port at a time. In other words, no input port can be connected to multiple output ports and no output port can be connected to multiple input ports. However, the reconfiguration results in a non-negligible delay δ (e.g., 10ms for 3D-MEMS [89]). During the reconfiguration, no traffic is allowed to send via OCS, which leads to non-trivial penalty for the flows with a small size.

4.1.2 Hybrid Electrical/Optical Datacenter Network Architecture

Despite the slow reconfiguration of OCS, previous studies [30, 63, 113, 162, 163] demonstrate that employing OCS in datacenter networks can deliver high bandwidth and significant cost reduction compared to the traditional packet-switched datacenters.

In this dissertation, we use the Hybrid-DCN architecture shown in Figure 4.2, which is the same as Helios [63] and c-Through [162]. The top-of-rack (ToR) switches are connected with a core EPS and an OCS, forming packet-switched network and circuit-switched network, respectively. The link rate between ToR and core EPS is bw_e, while the link rate between ToR and OCS is bw_o. We assume there are R racks in the cluster and one rack can send data via OCS to only another rack at a time, as in Helios [63] and c-Through [162]. We consider that the OCS has R input ports and R output ports, and each rack connects to one input port and one output port. Since OCS can connect one input port to only one output port at a time, it means that one rack can send data via OCS to only one other rack at a time. To ensure a high utilization of OCS (e.g., 90%), the average duration of each configuration should be at least 9δ (e.g., 90ms) [89, 113], which means that OCS can only transfer elephant flows (e.g., the flow with traffic greater than 100Gbps*90ms=1.125GB), while the mice flows (non-elephant flows) are still sent via the EPS to meet certain latency requirements. We assume that only the flows with size larger than the elephant flow threshold (e.g., 1.125GB) are sent via OCS; otherwise, it communicates through EPS. We define shuffle-heavy jobs as the jobs with shuffle data size no less than the elephant flow threshold.

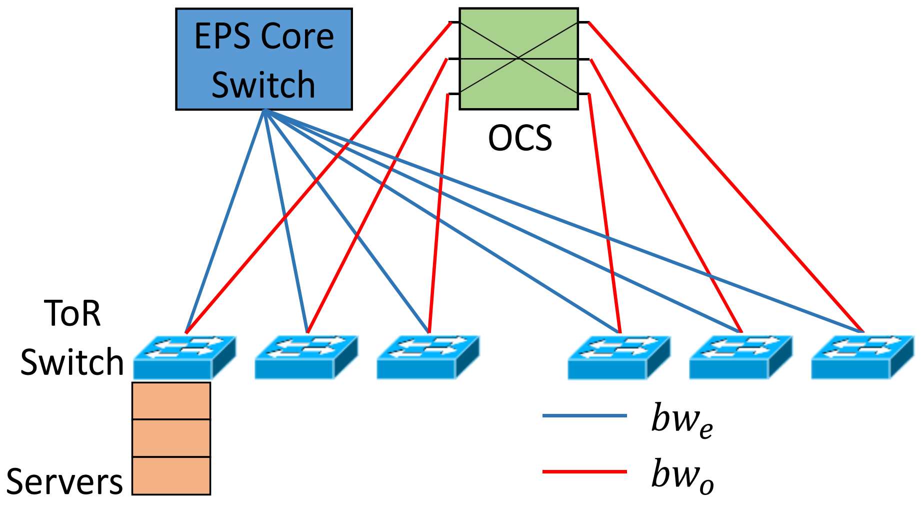


Figure 4.2: Architecture of Hybrid-DCN. The link rate between ToR and core EPS is bw_e, while the link rate between ToR and OCS is bw_o.

4.1.3 Traffic Matrices

We express traffic demand between each rack by a matrix M of size $R*R$. We denote the traffic sent via OCS as an $R*R$ matrix M_o; the remaining traffic is sent via EPS. The OCS reconfiguration process dynamically estimates the traffic demands, builds the matrices, and accordingly computes and establishes the port connections for data transfer. In this paper, as in [30, 63, 162], the OCS in Hybrid-DCN is reconfigured periodically with a fixed *reconfiguration interval* (e.g., $1s$).

To fully take advantage of OCS for high performance, the desired properties of the matrices M and M_o for data-parallel frameworks are listed as follows, in which the balance-skewness is identified by us.

- **Skewness [113].** The demand from any rack is high to only a few other racks and is low to the remaining racks, forming a skew demand matrix M. As a result, the high-demand entries in M can be well served by the OCS, while the low-demand entries can be served by the EPS.

- **Sparsity [113].** The OCS demand matrix M_o should be sparse (with only a

$$\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\ \hline
1 & 20 & 20 & 20 & 20 \\
2 & 2.5 & 2.5 & 2.5 & 2.5 \\
3 & 1.25 & 1.25 & 1.25 & 1.25 \\
4 & 1.25 & 1.25 & 1.25 & 1.25 \\ \hline
\end{array}
\qquad
\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\ \hline
1 & 6.25 & 6.25 & 6.25 & 6.25 \\
2 & 6.25 & 6.25 & 6.25 & 6.25 \\
3 & 6.25 & 6.25 & 6.25 & 6.25 \\
4 & 6.25 & 6.25 & 6.25 & 6.25 \\ \hline
\end{array}$$

(a) (b)

Figure 4.3: Balance-skewness of demand matrix.

few non-zero entries), since a rack can send data via OCS to only one other rack at a time.

- **Balance-skewness.** The shuffle traffic of a shuffle-heavy job is balanced between racks to reduce the durations of shuffle data transfer, as illustrated in Figure 4.3. Suppose that a shuffle-heavy job with 20 map tasks, 4 reduce tasks and 100 units of shuffle data runs in a 4-rack cluster. Suppose flows with no smaller than 1 unit are elephant flows and the speed of OCS is 1 per unit time. Assume each map task generates the same size of shuffle data and each reduce task processes the same size of data [94]. (a) This is the demand matrix when racks 1,2,3,4 process 16,2,1,1 map tasks and 1,1,1,1 reduce task, respectively. The time to complete the data transfer of this matrix is 20+20+20=60. (b) This is the demand matrix when racks 1,2,3,4 process 5,5,5,5 map tasks and 1,1,1,1 reduce task, respectively. The time to complete the data transfer of this matrix is 6.25+6.25+6.25=18.75.

In spite of the advantages of Hybrid-DCN, the state-of-the-art job schedulers in YARN are not suitable for Hybrid-DCN because these schedulers cannot produce demand matrices with the above desired properties. For example, in Fair scheduler [62] and Delay scheduler [172], the default input data placement strategy is to randomly place the data blocks across the entire cluster. Then, the map tasks will be spread across the entire cluster because a map task is assigned to the node that stores its input data block in order to achieve high map data locality. ShuffleWatcher [3] aims to evenly distribute the shuffle network traffic spatially (among different racks) and temporally (during different time periods). These schedulers generate many mice flows, which make the traffic demand matrices non-skew and non-sparse. Corral [94] places all the map and reduce tasks of a job in the same set of racks to avoid cross-rack shuffle data transfer, which may produce high competition of containers in the set of racks and hence sacrifice parallelism. In additional, Corral cannot take full advantage of OCS in Hybrid-DCN since Corral attempts to avoid cross-rack traffic. Therefore, a new job scheduler is needed for YARN in Hybrid-DCN that can take full advantage of the high-bandwidth OCS to achieve better job performance.

4.1.4 Opportunity

Several previous studies [2, 50, 66, 73, 74, 94, 97] show that cluster workloads contain a large number of recurring jobs, whose *job characteristics*, including input/shuffle/output data sizes, job arrival time, the number of map/reduce tasks, and the map/reduce task duration, can be predicted with a small error (e.g., 6.5% [94]). The predictability characteristics allow us to determine which racks to place the job input datasets and run the tasks for the recurring jobs *before* the input datasets and the jobs are submitted to the cluster. Many practical scenarios allow us to place data beforehand [94]. For example, in the cloud environment such as Microsoft Azure [117] and AWS [11], data is often stored in a dedicated storage cluster (e.g., Microsoft Azure Storage [118] and Amazon S3 [10]). To run MapReduce on the cloud, the data is fetched from the storage cluster. At this step, the data can be placed on the pre-determined racks.

4.2 Design of JobPacker

4.2.1 System Architecture

JobPacker has a shuffle data aggregation scheme that facilitates to use OCS. In addition, as shown in Figure 4.4, JobPacker consists of an offline scheduler and an online scheduler. The offline scheduler is responsible for deciding the schedule for the recurring jobs in the next unit period and has two main components – job profiler and job manager. The job profiler explores the tradeoff between parallelism and traffic aggregation, and returns all feasible *map-width* and *reduce-width* pairs of each shuffle-heavy recurring job (i.e., number of racks to run the map and reduce tasks) that can leverage OCS effectively while achieving sufficient parallelism. Then, the job manager finds out the best (map-width, reduce-width) for the shortest completion time of each shuffle-heavy recurring job, and also generates the global schedule including which racks to run the map/reduce tasks of each recurring job, and the sequence to run the map/reduce tasks of recurring jobs in each rack that yields the best performance (i.e., high throughput for batch jobs and short completion time for online jobs). For example, if the map tasks of job i, the reduce tasks of job j, and the map tasks of job k are assigned to a rack, the sequence on this rack is $Seq = \{map_i, reduce_j, map_k\}$, where map_j and $reduce_j$ means any map task and any reduce task of job j, respectively.

Based on offline schedule generated from job manager, the online scheduler guides the data placement and task placement of the job. The ad-hoc jobs are then scheduled based on previous scheduling scheme (e.g., Fair [62] and use the idle resources that are not assigned to recurring jobs.

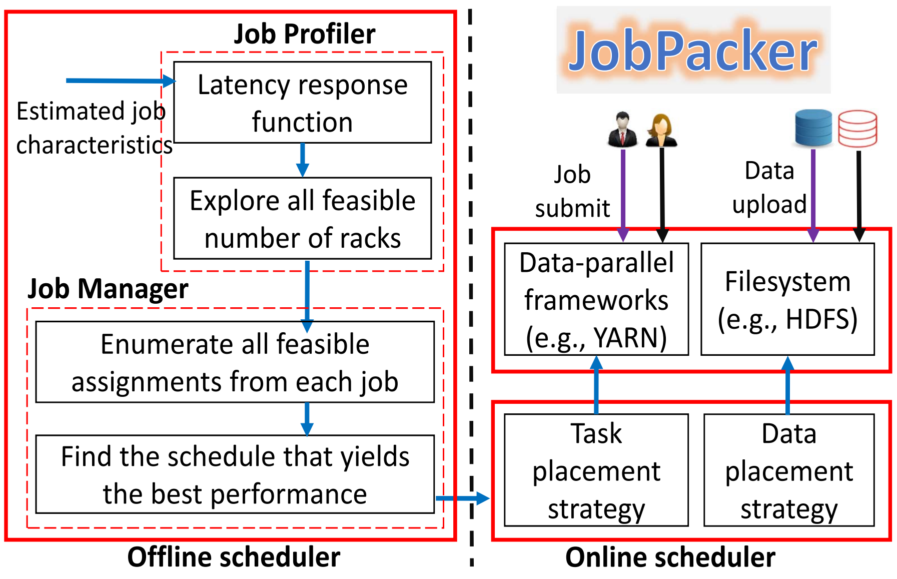

Job Profiler
Estimated job characteristics
Latency response function
Explore all feasible number of racks
Job Manager
Enumerate all feasible assignments from each job
Find the schedule that yields the best performance
Offline scheduler
JobPacker
Job submit
Data upload
Data-parallel frameworks (e.g., YARN)
Filesystem (e.g., HDFS)
Task placement strategy
Data placement strategy
Online scheduler

Figure 4.4: System architecture of JobPacker.

4.2.2 Shuffle Data Aggregation

Currently, the reduce task is associated with its shuffle and the shuffle starts fetching data once the corresponding reduce task is scheduled [159]. However, this default scheme does not facilitate shuffle traffic aggregation. Hence, we propose not to start the shuffle immediately after its corresponding reduce task is scheduled to a container. In order to aggregate the shuffle data transfers of a job, we force the shuffle to start until more reduce tasks from the same job are assigned to containers and the size of aggregated shuffle data of the reduce tasks reaches the elephant flow threshold. Then, we can use high-bandwidth OCS for shuffle data transfer to reduce the transfer delay of low-bandwidth EPS. If the size of aggregated shuffle data cannot reach the elephant flow threshold for a job (i.e., non-shuffle-heavy jobs), the shuffle data is transferred through EPS, which will not take a long time due to the small data size.

Figure 4.5 illustrates the shuffle data transfer in shuffle-heavy jobs in previous schedulers and in JobPacker. The shuffle data aggregation in JobPacker brings about two advantages. First, it enables the use of high-bandwidth OCS to accelerate the

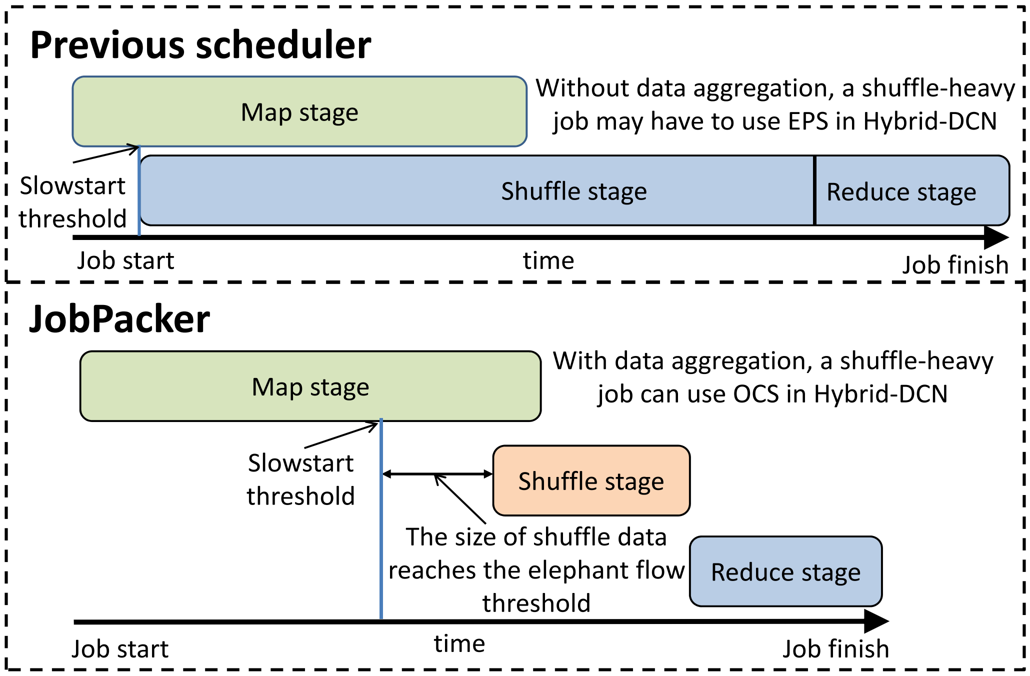

Previous scheduler
Map stage
Without data aggregation, a shuffle-heavy job may have to use EPS in Hybrid-DCN
Slowstart threshold
Shuffle stage
Reduce stage
Job start
time
Job finish
JobPacker
Map stage
With data aggregation, a shuffle-heavy job can use OCS in Hybrid-DCN
Slowstart threshold
Shuffle stage
The size of shuffle data reaches the elephant flow threshold
Reduce stage
Job start
time
Job finish

Figure 4.5: Illustration of shuffle-heavy jobs to take advantage of OCS.

shuffle data transfer, which shortens the shuffle duration significantly for the job. On the contrary, without shuffle data aggregation, a shuffle-heavy job may have to use the low-bandwidth EPS in Hybrid-DCN since the sizes of its traffic flows are small. The shuffle data aggregation does not degrade the performance of non-shuffle-heavy jobs much, as their shuffle data is relatively small so their shuffle duration is relatively short.

Second, in YARN, the slowstart threshold is set to a small value (default 5% [3]) to achieve high intra-job concurrency for high performance. However, in this case, the reduce tasks occupy the containers when they are doing nothing but transferring shuffle data, which wastes precious resources. With the data aggregation scheme in Hybrid-DCN, as shown in Figure 4.5, JobPacker can increase the slowstart threshold to prevent the reduce tasks from occupying resources for too long without compromising job performance, since fast data transfers through high-bandwidth OCS offsets the influence of intra-job concurrency reduction for shuffle-heavy jobs. For non-shuffle-heavy jobs, since transferring small-size data can be completed in short time duration, they do not need a low slowstart threshold for high intra-job concurrency. The slowstart threshold should be determined depending on the workloads in the system (i.e., whether reduce tasks can occupy containers for a long time without compromising other tasks' performance). If the system is lightly loaded, the slowstart threshold can be smaller for higher intra-job concurrency for high performance.

4.2.3 Offline Scheduler

We use r_j^{map} and r_j^{red} to denote the number of racks that are assigned to run job j's map and reduce tasks, respectively. We evenly distribute the map and reduce tasks among the r_j^{map} and r_j^{red} racks to achieve the balance-skewness property.

4.2.3.1 Job Profiler

We use a *latency response function* (LRF) [94] to model the latency for every job j. LRF takes the number of racks allocated to job j as input and predicts the latency of job j. LRF assumes that the map, shuffle and reduce stages run sequentially for simplicity though the shuffle stage overlaps with the map stage. This assumption matches JobPacker since it reduces the overlap (as shown in Figure 4.5). LRF also assumes that the map and reduce tasks of job j are scheduled on the same number of racks (i.e., $r_j^{map} = r_j^{red}$), which is not always correct in practice. In this paper, we remove this assumption to improve LRF. The latency of a job is calculated by:

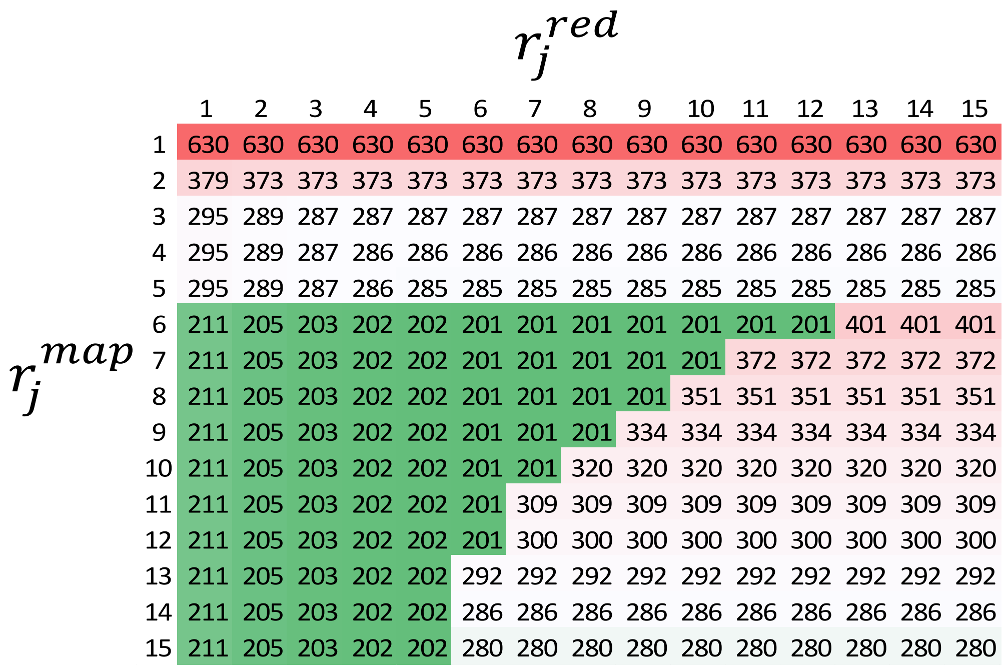
$$L_j(r_j^{map}, r_j^{red}) = l_j^{map}(r_j^{map}) + l_j^{shu}(r_j^{map}, r_j^{red}) + l_j^{red}(r_j^{red}), \tag{4.1}$$

where $l_j^{map}(r_j^{map})$, $l_j^{shu}(r_j^{map}, r_j^{red})$ and $l_j^{red}(r_j^{red})$ denote the latency for each of the three stages. Please refer to [94] for the details of how to compute $l_j^{map}(r_j^{map})$ and $l_j^{red}(r_j^{red})$ from the estimated job characteristics (input/shuffle/output data sizes and the number of tasks). $l_j^{shu}(r_j^{map}, r_j^{red})=\frac{D_j(r_j^{map}, r_j^{red})}{BW}$, where $D_j(r_j^{map}, r_j^{red}) = \frac{D_j^s}{r_j^{map} \cdot r_j^{red}} \cdot (r_j^{red}-1)$ is cross-rack shuffle data size, BW is the bandwidth, and D_j^s is the shuffle data size of job j. To determine bandwidth BW, we need to determine whether OCS or EPS is used in the shuffle stage of job j. Since shuffle data is sent from all map tasks to all reduce tasks, we check if the shuffle data size of job j divided by $r_j^{map} * r_j^{red}$ (i.e., $\frac{D_j^s}{r_j^{map} \cdot r_j^{red}}$) is greater than the elephant flow threshold. If yes, OCS is used; otherwise, EPS is used.

The job scheduler needs to carefully determine r_j^{map} and r_j^{red} for each shuffle-heavy job to achieve an optimal balance between parallelism and traffic aggregation, which yields relatively low job latency. For each value assignment of r_j^{map} and r_j^{red}, we can compute the latency of job j based on Equ. (4.1).

Figure 4.6 shows the latencies of an example shuffle-heavy job under different assignment combinations on a 15-rack cluster, where each rack has 600 containers. The example job consists of 3472 map tasks and 169 reduce tasks. The row and column represent r_j^{map} and r_j^{red}. Each entry is the latency when the map and reduce tasks are evenly distribute to r_j^{map} and r_j^{red} racks.

We see that as r_j^{map} increases from 1 to 5, the latency of the job drops significantly due to higher parallelism. The number of map tasks for this job is 3472, which is greater than the total number of containers in 5 racks ($5 * 600 = 3000$). Since the job has 169 reduce tasks, running the reduce tasks on one rack (i.e., $r_j^{red} = 1$) is sufficient for all the reduce tasks to run concurrently, i.e., achieving parallelism. Additionally, the latencies in the *green* zone are considerably lower than the latencies in the other zones due to two reasons. First, the assignment combinations in this zone do not sacrifice the parallelism. Second, OCS is used for shuffle data transfer in the green zone. As a result, the green zone illustrates all feasible (map-width, reduce-width) pairs for job j that can leverage the OCS to achieve a good tradeoff between parallelism and traffic aggregation.



r_j^{red}

r_j^{map}	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	630	630	630	630	630	630	630	630	630	630	630	630	630	630	630
2	379	373	373	373	373	373	373	373	373	373	373	373	373	373	373
3	295	289	287	287	287	287	287	287	287	287	287	287	287	287	287
4	295	289	287	286	286	286	286	286	286	286	286	286	286	286	286
5	295	289	287	286	285	285	285	285	285	285	285	285	285	285	285
6	211	205	203	202	202	201	201	201	201	201	201	201	401	401	401
7	211	205	203	202	202	201	201	201	201	201	372	372	372	372	372
8	211	205	203	202	202	201	201	201	201	351	351	351	351	351	351
9	211	205	203	202	202	201	201	201	334	334	334	334	334	334	334
10	211	205	203	202	202	201	201	320	320	320	320	320	320	320	320
11	211	205	203	202	202	201	309	309	309	309	309	309	309	309	309
12	211	205	203	202	202	201	300	300	300	300	300	300	300	300	300
13	211	205	203	202	202	292	292	292	292	292	292	292	292	292	292
14	211	205	203	202	202	286	286	286	286	286	286	286	286	286	286
15	211	205	203	202	202	280	280	280	280	280	280	280	280	280	280

Figure 4.6: Latencies of an example job under different assignments.

4.2.3.2 Job Manager

As [94], we consider two scenarios of job submission: *batch* and *online* scenarios. In the *batch* scenario, all jobs are submitted at the same time, and the goal is is to makespan, i.e., the time to finish all the jobs in the batch. In the *online* scenario, jobs are submitted at different times and the goal is to minimize the average job completion time, i.e., the average time from the arrival of a job until its completion.

For both batch and online scenarios, we can model the job scheduling as an optimization problem accordingly to achieve the goal. Nevertheless, the optimization problems for both scenarios are analogical to complex directed-acyclic-graph (DAG) structured job scheduling problem [94], which is well-known as NP-hard [122, 153]. Hence, we propose a heuristic method to solve the scheduling problem.

Batch scenario We define a shuffle-heavy job's width as the maximum value of $r_j^{map} + r_j^{red}$ among all of its feasible (map-width, reduce-width) pairs. The width of a non-shuffle-heavy job is defined as the total number of map and reduce tasks divided by the number of containers on a rack. First, we need to determine the priority of each job in scheduling. We could use the algorithm in Corral [94] that sorts the batch of jobs in descending order of job width. This widest-job-first algorithm avoids the case that the widest job cannot find enough racks to run all of its tasks concurrently and needs to wait for the job that is allocated to only a few racks to complete, which wastes the resources [94]. However, using this sorting algorithm, the *extremely* shuffle-heavy jobs are more likely to have very high priorities as these jobs most probably have extremely huge input data size (see explanation in Section 4.2.4) and hence have more tasks, which requires more racks. Then, it may lead to an extremely high network load and computing load (i.e., demand for a larger number of containers) at the beginning and a light network and computing load later. This resource utilization pattern is not desired [3], because all these extremely shuffle-heavy jobs compete for the precious resource simultaneously at the beginning, which degrades the performance significantly.

In order to solve this problem, we propose to evenly divide a batch's workload (for shuffle data transfer and map/reduce tasks) to B sub-batches, so that each sub-batch's workload will not create resource competition while fully utilizing cluster resource, as shown in Figure 4.7. The number of sub-batches B is a tunable parameter based on the entire cluster capacity and the resource demands of jobs.

To divide into B sub-batches, we use Tetris [76], which chooses a job to assign to a server with available capacity in order to increase the resource utilization of each server considering multi-resources (e.g., CPU, memory, bandwidth). Basically, based on the available resources on a server, Tetris gives a score[1] to each job and then greedily picks the job with the highest score to run on the server. We treat each sub-batch as a server and treat the shuffle data size, the number of map tasks

[1]For example, a server has (0.2,0.3,0.5) available resources. If a job consumes (0.1,0.2,0.3), the score of this job is the dot product of the two resource vectors, i.e., 0.1*0.2+0.2*0.3+0.3*0.5=0.23.

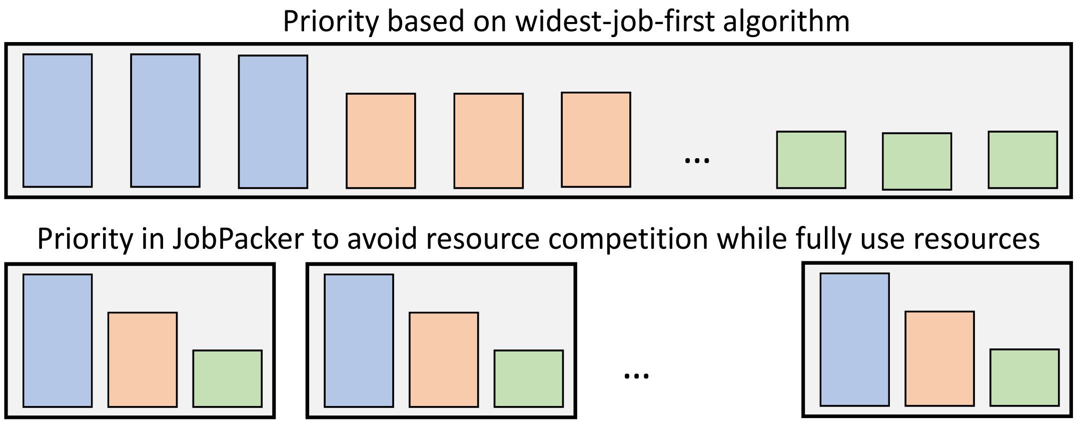

Priority based on widest-job-first algorithm
...
Priority in JobPacker to avoid resource competition while fully use resources
...

Figure 4.7: Priority determination based on sub-batches.

and the number of reduce tasks of each job as its demand on multi-resources. The capacity of each sub-batch is the capacity of the cluster on different dimensions (total cross-rack bandwidth, the number of containers). Then the batch division problem is interpreted as the job-to-server packing problem. The output includes B sub-batches, and the resource demands on each resource from all sub-batch are similar. The resource demands of a sub-batch on different resources equal the sum of shuffle data sizes, the sum of the number of map tasks and the sum of the number of reduce tasks of all the jobs in the sub-batch.

In each sub-batch, we sort the jobs in the descending order of width. The jobs with the same width are further sorted in the decreasing order of job latency, because the longest-latency-job-first first algorithm is effective for makespan minimization [72, 94]. After sorting, we combine all the sub-batches in a random order. Finally all the jobs form a list for sequential offline scheduling as shown in Figure 4.7.

During the offline scheduling, we keep track of the time T_{ik} when the container k on rack i completes the current task and requests the next task. We compute the time needed by the map, shuffle and reduce stage using the method in Equ. (4.1).

For each job j from the sorted list, we check whether it is shuffle-heavy or not and conduct the scheduling as follows. We assign the tasks of a shuffle-heavy job to the best (map-width, reduce-width) pair among all feasible pairs that yields the best performance, while assigning the tasks of a non-shuffle-heavy job to any containers that are available. We use N_j^{map} and N_j^{red} to denote the number of map tasks and reduce tasks of job j.

Non-shuffle-heavy jobs We pick the first N_j^{map} available containers based on the next available time T_{ik} of each container. We assign these containers to the map tasks and update the T_{ik}. Next, we pick the first N_j^{red} available containers based on T_{ik} to run the reduce tasks of job j. The reduce stage start time S_j^{red} is computed by adding the completion time of the last map task (cmp_j^{map}) and the shuffle stage latency $l_j^{shu}(r_j^{map}, r_j^{red})$. Finally, the job manager updates the sequences of the racks that run job j's map and reduce tasks correspondingly, and records the set of racks that run job j's map tasks (R_j^{map}), which will be used to guide the input data placement in online scheduling.

Shuffle-heavy jobs We enumerate each (map-width, reduce-width) pair among all feasible pairs, and find the pair that yields the earliest completion time. For each rack i, we find out the time when $\lceil N_j^{map}/r_j^{map} \rceil$ containers are available. Then we find r_j^{map} the earliest such available racks to run its map tasks, and update the corresponding T_{ik} of each assigned container. We use the same way above to compute the completion time of the last map task cmp_j^{map}. Similarly, we find r_j^{red} the earliest available racks that have $\lceil N_j^{red}/r_j^{red} \rceil$ available containers. Then we can compute the job completion time in this iteration. After we finish all interations, we find out the (map-width, reduce-width) that yields the earliest job completion for job j, and which racks to run job j's map and reduce tasks. Finally, the job manager updates the sequences of those racks correspondingly, and records the set of racks that run job j's map tasks (R_j^{map}).

As a result, in the offline schedule, a rack has large data transfer to only a few racks and has small data transfer to the other racks, which satisfies the skewness and sparsity desired properties. Also, since the tasks of each shuffle-heavy job are evenly distributed among the set of racks, the balance-skewness is achieved.

Online scenario The objective in the online scenario is to minimize the average job completion time. We sort the jobs in an increasing order of their predicted job arrival times. When the jobs are submitted at the same time, we use the sorting algorithm for sorting jobs in each sub-batch. Other steps are the same as those in the batch scenario.

Over-provisioning Recall that we estimate the resource demand (i.e., the number of containers needed) of each recurring job with a small error. However, in the offline scheduler, we *intentionally* assign more containers to each job than the estimation by a certain ratio (i.e., *over-provisioning ratio*) due to two reasons. First, we take the estimation variation into account. Thus, the recurring jobs can have sufficient containers during the actual job execution. Second, we attempt to leave ad-hoc jobs sufficient containers to run in the cluster. This strategy will not waste resource because during actual job execution, when the recurring jobs do not need as many containers as planned in the offline schedule, the unused containers can be used by the ad-hoc jobs or other recurring jobs assigned to the same racks. The cluster operators can adaptively determine the over-provisioning ratio based on the estimation variance and the percent of ad-hoc jobs in their clusters to achieve better performance. If a cluster has higher estimation variance or fewer recurring jobs, we can set a higher over-provisioning ratio; otherwise, it can be zero.

Summary The job manager in JobPacker returns a sequence for each rack, which includes the recurring jobs' map or reduce tasks to run on this rack. Besides the sequence for each rack, for each job j, the job manager outputs R_j^{map} to guide the placement of its input data. The outputs of job manager are then passed to the online scheduler, which will be introduced in Section 4.2.4.

4.2.4 Online Scheduler

The online scheduler executes the generated offline schedule. When the input dataset of a recurring job j is uploaded to the cluster, JobPacker places one replica of each data chunk of job j in a randomly chosen rack from R_j^{map} to achieve data locality. The second and the third replicas of all data chunks are randomly placed on the other racks. This data placement strategy still obeys the default data placement in HDFS that places the replicas of each data chunk in two random racks.

We try to determine if we can judge whether an ad-hoc job is a shuffle-heavy job based on its input data size in order to avoid scheduling shuffle-heavy jobs together in a rack to achieve the sparsity property. Using the two Facebook workloads in 2009 (FB2009-1, FB2009-2) and two Facebook workloads in 2010 (FB2010-1, FB2010-2) from [32], we plot Figure 4.8 that shows the cumulative distribution function (CDF) of input data size of all the shuffle-heavy jobs (shuffle data size greater than 1.125GB). We observe that most shuffle-heavy jobs have an input data size larger than 1GB. Based on the above observation, when the input data of a job is submitted to the cluster, if the input data size is greater than a threshold (e.g., 1GB), we empirically treat the job as a shuffle-heavy job. Note that non-shuffle-heavy jobs may be sometimes over-estimated as shuffle-heavy jobs. However, over-estimation is better than under-estimation, as under-estimation may inappropriately place some shuffle-heavy jobs, which generates unwanted demand matrices.

To determine the priorities of ad-hoc jobs, we use the default user-specified scheduler, such as Fair [62]. Recall that in the offline schedule, each rack is assigned with a sequence of recurring jobs. During scheduling, when a rack has a container available, the online scheduler tries to follow the offline schedule (which is only for recurring jobs). If there are recurring jobs, the online scheduler selects a map/reduce task of the first job in the sequence. Only when there is no recurring job assigned to this

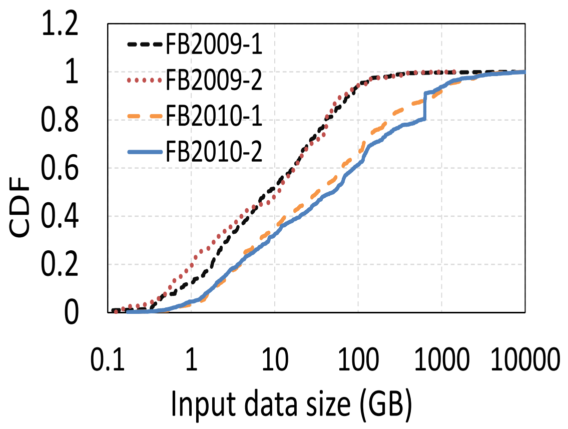


Figure 4.8: Workload analysis of shuffle-heavy jobs.

rack, an ad-hoc job will be scheduled to the container. In this step, the scheduler tries to schedule shuffle-heavy job first while avoiding scheduling shuffle-heavy jobs together in a rack in order to achieve the sparsity property. Specifically, the scheduler checks whether there are any tasks of shuffle-heavy jobs *currently running* in the rack. If yes, the scheduler selects a task from the ad-hoc non-shuffle-heavy job with the highest priority. Otherwise, the scheduler gives high priority to the task from the ad-hoc shuffle-heavy job with the highest priority if there are any in the queue. In the case of failure of a rack, JobPacker ignores the guidance from the offline scheduler and schedules the jobs assigned to this rack based on the default scheduler.

4.2.5 Discussion

In online scheduler, for each scheduling decision, JobPacker performs simple examinations (e.g., sequence and priority of jobs), which is quite similar to the Fair scheduler [62]. Hence, the computation overhead in online scheduler is no more than Fair, indicating the excellent scalability of JobPacker.

We also tried to aggregate the shuffle data transfers of ad-hoc jobs in the online scheduler. Specifically, while the scheduling for the recurring jobs remains the same in the online scheduler, we further attempted to place the tasks of ad-hoc jobs on only a few racks. We first analyzed the ratio between the best map-width and the number of map tasks for each recurring job, and we took the average of the ratios of all the recurring jobs. Then, when the input data of an ad-hoc job is uploaded to the cluster, we computed the number of map tasks (input data size divided by block size), and derived the map-width W_{map} for the ad-hoc job by using the number of map tasks and the average ratio above. Next, we placed the first replica of all the data chunks in W_{map} racks (randomly selected), and place the second and third replicas in W_{map} racks, which are randomly selected from the racks that do not have the first replica. When placing the map tasks, the map tasks are restricted to run on W_{map} racks that are available the earliest and have its data. The reduce-width W_{reduce} of each ad-hoc job is estimated using similar method and the reduce tasks are restricted to run on any W_{reduce} racks that are available the earliest.

However, we find that such shuffle data aggregation of ad-hoc jobs degrades the performance of ad-hoc and recurring jobs (see Figure 4.18 in Section 4.3.4). This is because it is hard to determine how many racks and which racks to run ad-hoc jobs without knowing the shuffle data size priori. Thus, we do not attempt to aggregate shuffle data transfers of the ad-hoc jobs currently. Nevertheless, the ad-hoc jobs still achieve significant improvement with JobPacker, as the recurring jobs can take advantage of OCS, the network load on EPS are significantly reduced, which benefits the ad-hoc jobs. We leave the exploration on how to place the tasks of ad-hoc jobs to aggregate their shuffle data transfers as a future work.

4.3 Performance Evaluation

4.3.1 Traces and Settings

Workload traces We evaluated JobPacker assuming that all the jobs are recurring first, and then using the workload with a mix of both ad-hoc and recurring jobs. We also conducted the sensitivity analysis of different settings. The workload trace we used was from the SWIM Facebook workloads [32]. Since this workload trace misses important information such as task running time, we first replayed all the jobs in the trace (using the tools provided in the same project [32]) one by one on a single-node Hadoop YARN cluster and then recorded the necessary information for every job. We used this recorded log as the workload trace for simulation and emulation.

Simulation In order to evaluate the performance of JobPacker in a large scale, we built a flow-based event simulator (details in Chapter 6) to replay the workload trace. In the simulation, there are 600 servers, organized into 20 racks with 30 servers each. Each server can run up to 20 tasks and has 10Gbps network bandwidth. The Hybrid-DCN topology is the same as in Figure 4.2. The link rate between the ToR switch and core EPS is 30Gbps, which yields a 10:1 oversubscription ratio. The ToR switch and OCS are always connected with 100Gbps link. We ran 1000 jobs selected from the workload. The elephant flow threshold was set to 1.125GB, which is inferred empirically from previous studies [63, 89, 113] to achieve high OCS utilization. As in [63, 162], we used Edmonds' algorithm [58] to compute the optimal input-to-output configuration for OCS in every reconfiguration.

Emulation on GENI We also conducted an emulation on GENI [69]. We built a testbed with 10 servers on GENI, each emulating a virtual rack. We assumed that each virtual rack can run up to 10 emulated tasks. Due to the bandwidth availability, the link rate between virtual racks via OCS is 1Gbps, while the link rate between virtual racks via EPS is 0.1Gbps. We limited the bandwidth for each task to 0.1Gbps, yielding an oversubscription ratio of 10:1. To emulate the Hybrid-DCN network, we used *tc* command to control the link reconfigurations. We also emulate the Hadoop framework to send the shuffle data on each virtual rack. We ran 200 jobs chosen from the workload trace. We shrank the transfer data size of each task, and the elephant flow threshold by a factor of 100, which equals the network bandwidth shrinking factor in GENI. The emulation allows us to evaluate the performance of JobPacker under real network environment.

4.3.1.1 Baselines

We compared JobPacker with Hybrid-DCN with three other systems.

(1) Fair scheduler [62] with Hybrid-DCN (F-Hybrid). It uses the Fair scheduler to schedule the jobs in Hybrid-DCN. Fair is the most widely used scheduler in current production clusters [62], and it assigns resources to jobs so that each job roughly receives an equal share of resources (containers) over time.

(2) Corral [94] with Hybrid-DCN (Corral). Corral places the map and reduce tasks of the same job on the same set of racks to reduce the cross-rack shuffle data transfer.

(3) Fair scheduler with traditional packet-switched datacenter network (F-EPS). It uses the Fair scheduler to schedule the jobs in a packet-switched network. In this system, the ToR switches are connected through an EPS core switch and their link rate is 100Gbps (1Gbps on GENI). This link rate is as high as the link rate of OCS though the system does not have OCS. However, the high-bandwidth EPS in this architecture leads to up to a factor of 9 higher CapEx and OpEx [63, 162], compared to the Hybrid-DCN.

4.3.2 Trave-driven Simulation Results

In this section, we present the experimental results in the simulation. We considered that all the 1000 jobs are recurring and can be predicted with zero error. All the experiments were run for 20 times and the average results are reported. The OCS was reconfigured every 1 second [162] and the reconfiguration delay was 10ms. The slowstart threshold was 0.7. The over-provisioning ratio was set to 0, as all the jobs are recurring.

4.3.2.1 Batch Scenario

In the batch scenario, the 1000 jobs were divided into 5 sub-batches. Figure 4.9 shows the makespan of different methods in the simulation. All the results are normalized by the results of F-Hybrid. We see that JobPacker outperforms F-Hybrid by 35% in the simulation. This is because Fair scheduler does not aggregate the network traffic to take full advantage of OCS in Hybrid-DCN. On the contrary, JobPacker attempts to aggregate the shuffle data transfer by placing the map and reduce tasks in a few racks without sacrificing the parallelism. Compared with Corral, JobPacker reduces the makespan by 23.5%. JobPacker is better than Corral in Hybrid-DCN because Corral attempts to place both map and reduce tasks on the same set of racks to reduce shuffle data transfer, which may result in high resource contention on the set of racks and hence it may take a long time for each job to get resources. Besides, Corral does not have the traffic aggregation mechanisms in JobPacker (e.g., postponing the shuffle data transfer) to trigger the use of OCS, which leads to a low utilization of OCS. On the contrary, JobPacker increases the utilization of OCS and allows the map and reduce tasks to be scheduled on different sets of racks. We also see that JobPacker achieves a comparable performance as F-EPS (less than 5% difference). However, as mentioned above, F-EPS generates significantly higher CapEx and OpEx. The results above demonstrate the effectiveness of JobPacker on improving the performance of data-parallel frameworks in Hybrid-DCN.

We then measured the percentage of total traffic sent via OCS and EPS, respectively. Figure 4.10 shows the percentage of traffic sent by OCS and EPS in the simulation. Since F-EPS does not have OCS, we do not show its results. We see that the OCS has a much higher utilization in JobPacker in the batch scenario (98.3%),

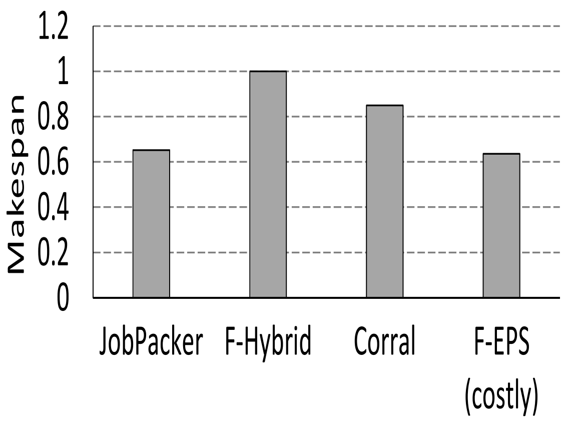


Figure 4.9: Makespan results in the batch scenario.

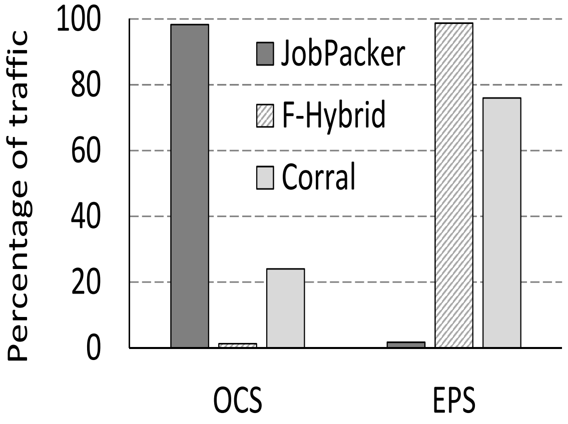


Figure 4.10: Percentage of traffic through OCS and EPS in the batch scenario.

compared with F-Hybrid (1.3%) and Corral (23.9%). This is because JobPacker aggregates the shuffle data of shuffle-heavy jobs to trigger the use of OCS to accelerate the shuffle data transfer. On the contrary, F-Hybrid spreads the shuffle data across racks, and hence is less likely to leverage OCS. Corral attempts to avoid cross-rack traffic and does not postpone the shuffle data to trigger the use of OCS. Hence, Corral has much lower OCS utilization than JobPacker. The result demonstrates the outstanding performance of JobPacker on taking advantage of OCS.

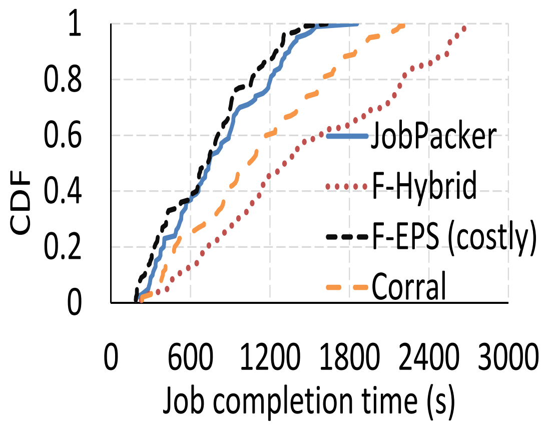


Figure 4.11: CDF of job completion times in the online scenario.

4.3.2.2 Online Scenario

In this section, the jobs arrived uniformly at random in $[0, 90]min$ in the simulation. Figure 4.11 shows the CDF of job completion times in the simulation. We see that the JobPacker significantly shortens the job completion times, compared with F-Hybrid and Corral. Specifically, JobPacker outperforms F-Hybrid with 43% improvement at the median job completion time. Compared with Corral, JobPacker reduces the median job completion time by 28%. We also see that the CDF of JobPacker is very similar to the CDF of F-EPS which however is very costly, indicating that JobPacker can be a cost-efficient solution for the network bottleneck problem in data-parallel frameworks. The results demonstrate the effectiveness of JobPacker on improving the job performance in Hybrid-DCN.

We also measured the traffic sent via OCS and via EPS in the online scenario, as shown in Figure 4.12. Similarly, compared with F-Hybrid and Corral, JobPacker has a much higher OCS utilization in the online scenario.

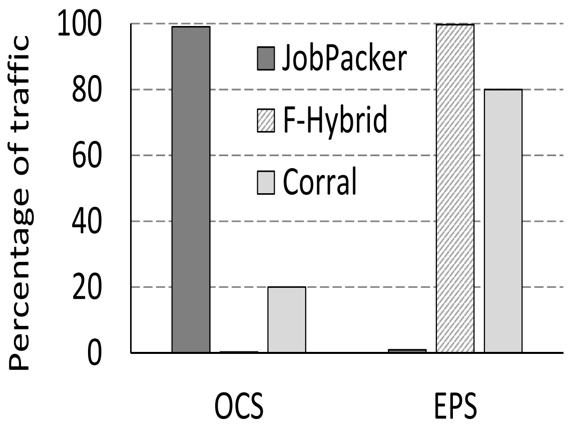


Figure 4.12: Percentage of traffic through OCS and EPS in the online scenario.

4.3.3 Emulation Results

In this section, we present the experimental results on GENI. We considered that all the 200 jobs are recurring and can be predicted with zero error. All the experiments were run for 20 times and the average results are reported. The OCS was reconfigured every 1 second [162] and the reconfiguration delay was 10ms. The slowstart threshold was 0.7. The over-provisioning ratio was set to 0, as all the jobs are recurring.

4.3.3.1 Batch Scenario

In the batch scenario, the 200 jobs were divided into 5 sub-batches. Figure 4.13 shows the makespan of different methods in GENI. All the results are normalized by the results of F-Hybrid. We see that JobPacker outperforms F-Hybrid by 49%. Compared with Corral, JobPacker has 33% improvement of makespan in GENI. We also see that in GENI, JobPacker achieves a comparable performance as F-EPS (less than 5% difference). The results in the emulation are consistent with the results in the simulation due to the same reasons, demonstrating the effectiveness of JobPacker on improving the performance of data-parallel frameworks in Hybrid-DCN.

We then measured the percentage of total traffic sent via OCS and EPS, respectively. Figure 4.14 shows the percentage of traffic sent by OCS and EPS in GENI.

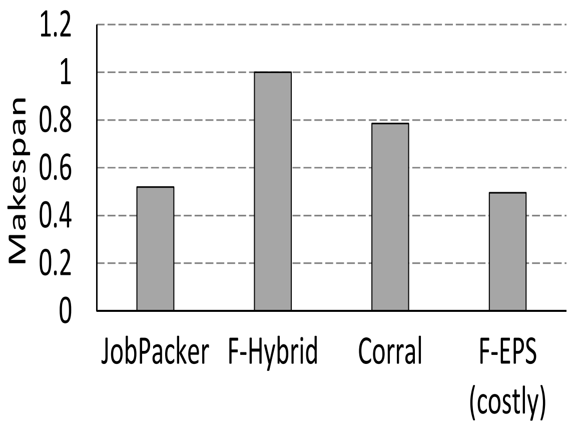


Figure 4.13: Makespan results in the batch scenario.

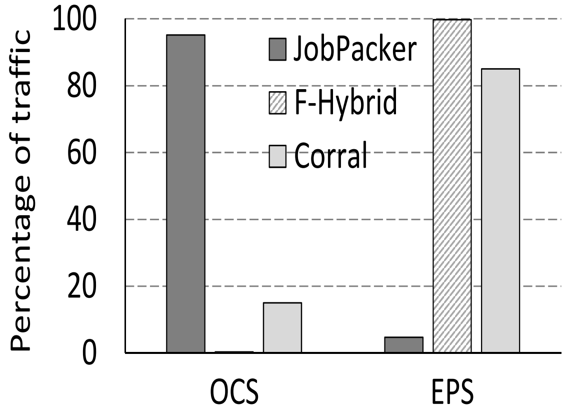


Figure 4.14: Percentage of traffic through OCS and EPS in the batch scenario.

Since F-EPS does not have OCS, we do not show its results. We see that the OCS has a much higher utilization in JobPacker in the batch scenario (96%), compared with F-Hybrid (0.8%) and Corral (16.0%). The results in the emulation are consistent with the results in the simulation due to the same reasons, demonstrating the outstanding performance of JobPacker on taking advantage of OCS.

4.3.3.2 Online Scenario

In this section, the jobs arrived uniformly at random in $[0, 20]min$ in GENI experiment. Figure 4.15 showa the CDF of job completion times in GENI. We see that the JobPacker significantly shortens the job completion times, compared with F-Hybrid

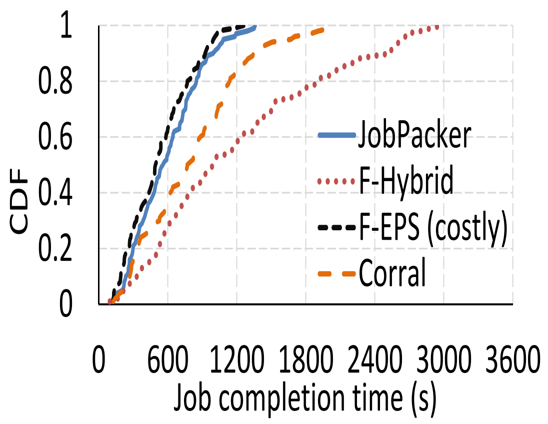


Figure 4.15: CDF of job completion times in the online scenario.

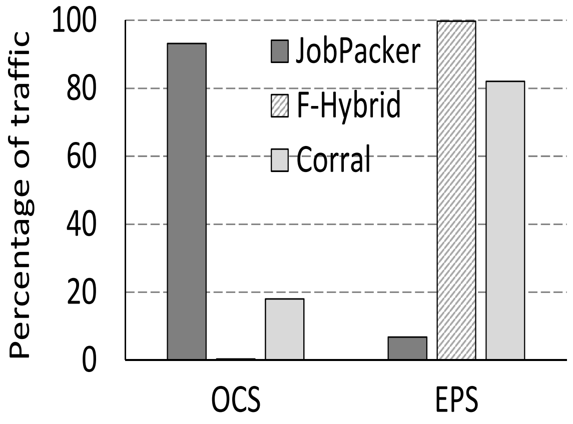


Figure 4.16: Percentage of traffic through OCS and EPS in the online scenario.

and Corral. Specifically, JobPacker outperforms F-Hybrid with 42% improvement at the median job completion time. Compared with Corral, JobPacker reduces the median job completion time by 27%. We also see that the CDF of JobPacker is very similar to the CDF of F-EPS which however is very costly, indicating that JobPacker can be a cost-efficient solution for the network bottleneck problem in data-parallel frameworks. The results in GENI are consistent with the results in the simulation due to the same reasons, demonstrating the effectiveness of JobPacker on improving the job performance in Hybrid-DCN.

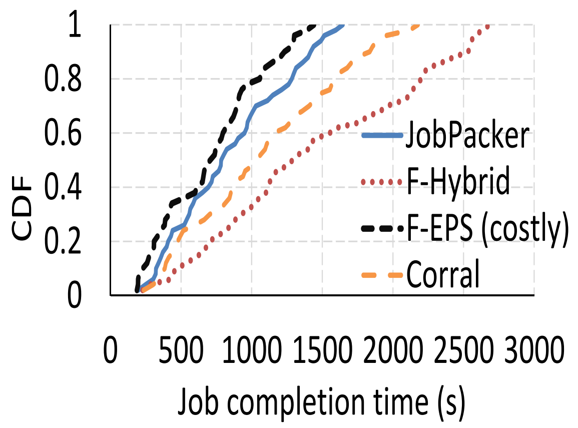
We also measured the traffic sent via OCS and via EPS in the online scenario, as shown in Figure 4.16. Similarly, compared with F-Hybrid and Corral, JobPacker has a much higher OCS utilization in the online scenario in GENI experiments.

4.3.4 Mix of Ad-hoc and Recurring Jobs

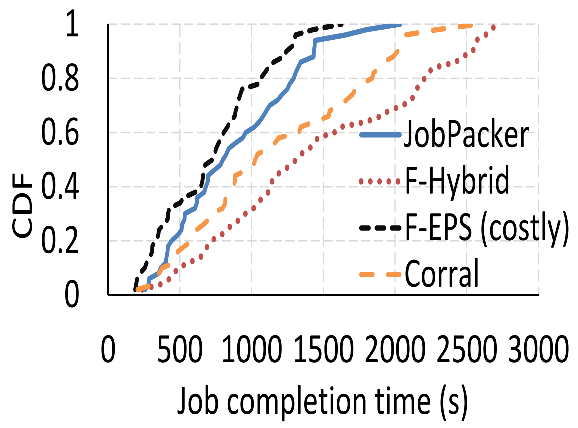
In this section, as in [94], we evaluated JobPacker in an online scenario, where there are a mix of ad-hoc and recurring jobs. Previous studies [2, 66, 94, 97] indicate that there are 40%-60% recurring jobs in the cluster. Therefore, we randomly selected half of the jobs as ad-hoc jobs and the rest are still recurring jobs. All the jobs arrived uniformly at random in $[0, 90]min$ and $[0, 20]min$ in simulation and GENI, respectively. In the offline scheduler, we set the over-provisioning ratio to 1.0.

4.3.4.1 Trace-driven Simulation Results

Figures 4.17(a) and 4.17(b) show the CDF of job completion times for recurring jobs and ad-hoc jobs in the simulation, respectively. Clearly, JobPacker generates shorter job completion times for both recurring and ad-hoc jobs, compared with F-Hybrid and Corral. In the simulation, we see that JobPacker reduces the median job completion time of recurring jobs and ad-hoc jobs by 40% and 38%, respectively, compared with F-Hybrid. We also see that JobPacker outperforms Corral by 24% and 25% for recurring jobs and ad-hoc jobs in the simulation. In addition, JobPacker achieves a comparable performance as F-EPS (which is very costly), demonstrating the superior performance of JobPacker. This is because in JobPacker, the recurring jobs can more effectively utilize the OCS to transfer their shuffle data, which significantly frees the network resource on EPS. Although we do not place the tasks of ad-hoc jobs on a few racks to aggregate the data transfers to use the OCS, they still benefit from the much lower utilization of network resource on EPS. Besides, as the recurring jobs finish faster, more computing resources are available for the ad-hoc jobs, which significantly increases the performance of ad-hoc jobs as well.

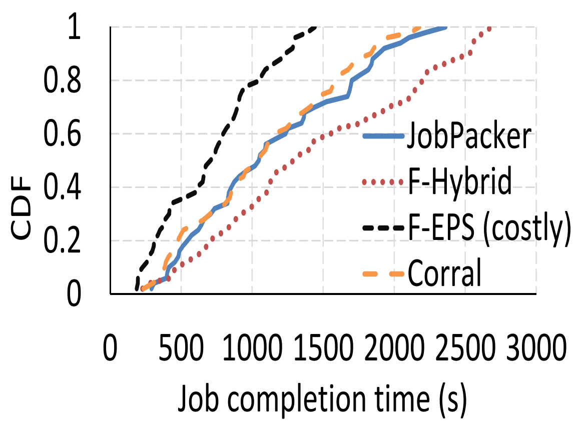


(a) Recurring jobs

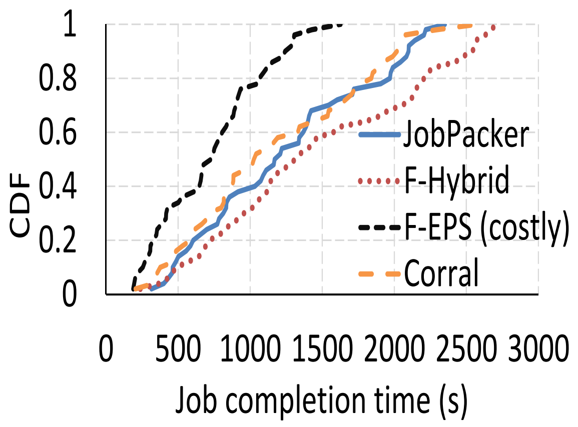


(b) Ad-hoc jobs

Figure 4.17: CDF of job completion times with a mix of jobs in the simulation.



(a) Recurring jobs



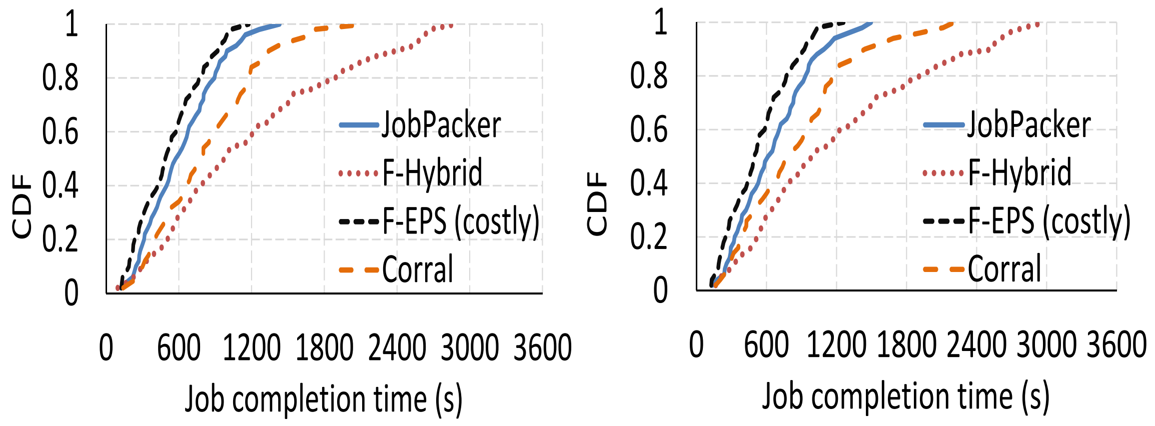
(b) Ad-hoc jobs

Figure 4.18: CDF of job completion times under the aggregation strategy for ad-hoc jobs.

We also tried to place the tasks of ad-hoc jobs in a few racks in Section 4.2.4 in the simulation environment. Figures 4.18(a) and 4.18(b) show the CDF of job completion times for recurring jobs and ad-hoc jobs, respectively, with the aggregation strategy mentioned in Section 4.2.4 for ad-hoc jobs. Clearly, we see that the performance of both recurring and ad-hoc jobs with JobPacker is degraded. This is because without knowing the shuffle data size priori, we cannot explore how many racks and which racks to run ad-hoc jobs. As a result, the ad-hoc jobs may sacrifice their parallelism during traffic aggregation and use the low bandwidth EPS to transfer the shuffle data.

4.3.4.2 Emulation Results

Figures 4.19(a) and 4.19(b) show the CDF of job completion times for recurring jobs and ad-hoc jobs in GENI, respectively. Clearly, JobPacker generates shorter job completion times for both recurring and ad-hoc jobs, compared with F-Hybrid and Corral. In GENI, JobPacker reduces the median job completion time of recurring jobs and ad-hoc jobs by 39% and 37%, respectively, We also see that JobPacker outperforms Corral by 26% and 29% for recurring jobs and ad-hoc jobs in GENI. In addition, JobPacker achieves a comparable performance as F-EPS (which is very costly) Thus, the results in the emulation are consistent with the results in the simulation, demonstrating the superior performance of JobPacker.



(a) Recurring jobs (b) Ad-hoc jobs

Figure 4.19: CDF of job completion times with a mix of jobs in GENI.

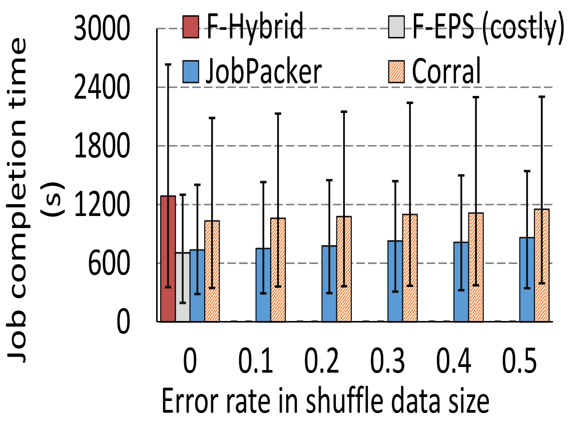
4.3.5 Sensitivity Analysis

In this section, we used our simulation environment to evaluate the robustness of JobPacker to several factors in online scenario, unless otherwise specified. We analyze the sensitivity when there are all recurring jobs, as in [94], unless otherwise specified. The experiment settings are the same as Section 4.3.2 unless otherwise specified.

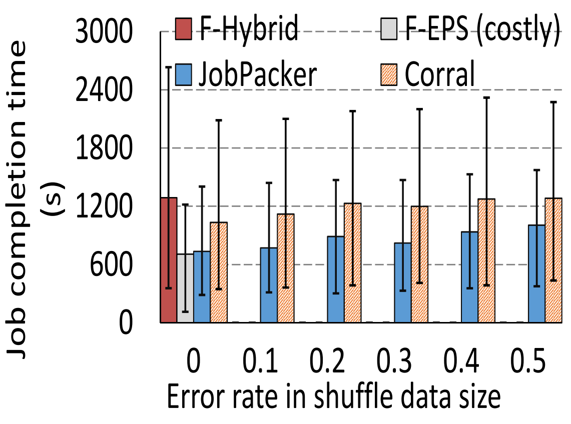
Error in prediction of shuffle data size We use the predicted shuffle data size to check if it is a shuffle-heavy job, determine all the feasible map-width and reduce-width, and compute the latency of shuffle stage. We define the prediction error rate of a job's shuffle data size as $\frac{|real-prediction|}{real}$. Though recent studies [2, 66, 94] show that the characteristics of recurring jobs can be predicted with a low error rate around 6.5%, we varied the error rate for the prediction of the shuffle data size of all the jobs by up to 50% to see how JobPacker performs. Figure 4.20(a) shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker versus different error rates. We do not measure the performance of F-Hybrid and F-EPS here since they do not predict the shuffle data size. As we see, the job completion times increase as the error rate increases. However, even with some prediction error, JobPacker still outperforms F-Hybrid by 33% and Corral by 25% at the median, and achieves similar performance as F-EPS, as JobPacker effectively utilizes the OCS.

Error in job arrival time In the offline scheduler, we sort the jobs in the online scenario based on the predicted job arrival times. In practice, the job arrival time may vary from the predicted arrival time. In this experiment, we added a random time delay in the range of $[-200, 200]s$ to f portion of jobs, where f is varied in the range of $[10\%, 50\%]$. Figure 4.20(b) shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker with varying portion of delayed jobs. We see that although up to 50% of the jobs' arrival times are not accurate, JobPacker shortens the median job completion time by 23% and 22% compared with F-Hybrid and Corral, respectively, and achieves comparable performance to F-EPS.

Slowstart threshold We varied the slowstart threshold in the range of $[20\%, 80\%]$. Figure 4.21 shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker with different slowstart thresholds. We see that when the slowstart threshold is in the range of $[50\%, 80\%]$, the job completion times are almost the same, indicating that the slowstart threshold in JobPacker can be set to a sufficient large range to achieve the best performance. On the other hand, when the slowstart



(a) Error in shuffle data size



(b) Error in arrival time

Figure 4.20: Performance variation with error in prediction of job characteristics.

threshold is set to $\leq 40\%$, the job completion times slightly increase. In these cases, the reduce tasks of the jobs hog up the resources that can be allocated to the tasks of other jobs for too long because of low slowstart thresholds.

Over-provisioning ratio In the previous experiments when there are both recurring and ad-hoc jobs, we set the over-provisioning ratio to 1.0. In this experiment, we varied the over-provisioning ratio in the range of $[0, 2]$ and used the same other settings as in Section 4.3.4. Figure 4.22 shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker with different over-provisioning ratios. The figure indicates that if the over-provisioning ratio is too low, the job completion times are significantly affected, since the planned resources are not sufficient to complete the recurring and ad-hoc jobs. The performance is not affected much when the over-provisioning ratio becomes larger. This is because in real cluster running, the over-provisioned resources can be used to run the ad-hoc jobs and other recurring jobs planned on the same resources.

OCS reconfiguration interval We varied the OCS reconfiguration interval in the range of $[0.1s, 3s]$. Figure 4.23 shows the median, 5^{th}, and 95^{th} percentile job completion times of all the jobs in JobPacker with different OCS reconfiguration intervals.

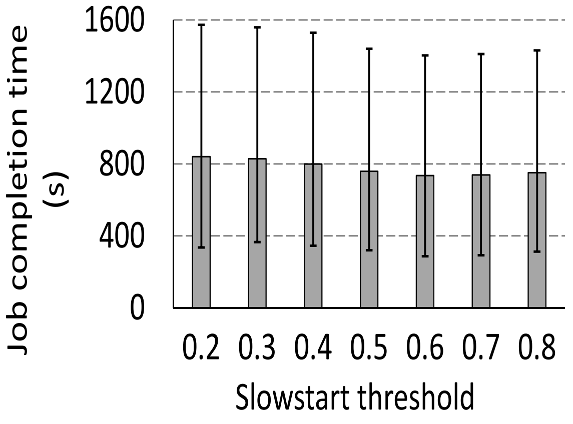


Figure 4.21: Varying slowstart threshold.

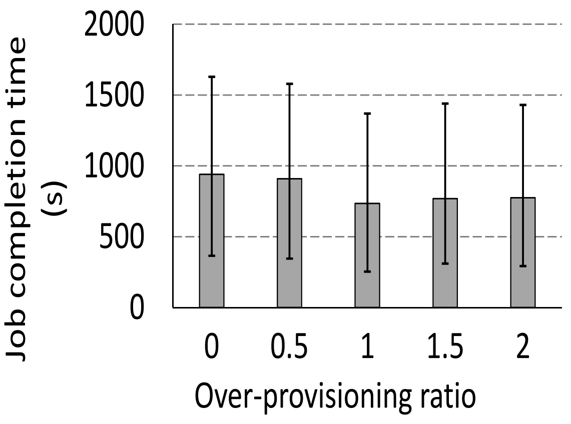


Figure 4.22: Varying over-provisioning ratios.

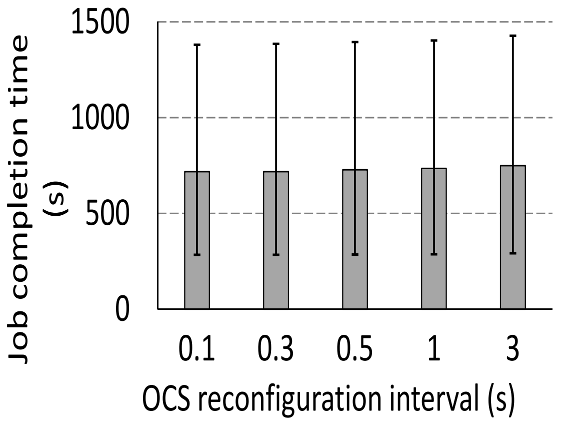


Figure 4.23: Varying OCS reconfiguration intervals.

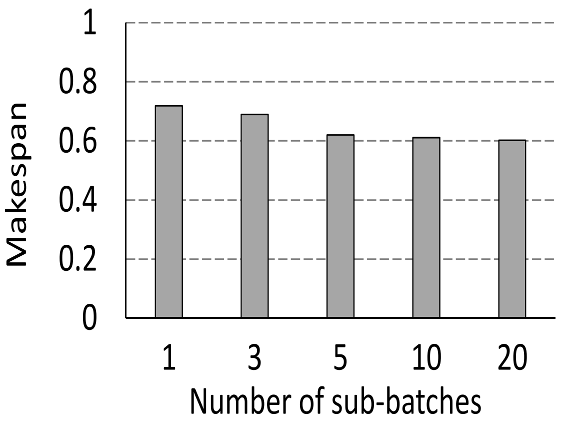


Figure 4.24: Varying the number of sub-batches.

We see that varying the reconfiguration interval only has minimal impacts on the performance of JobPacker. Even setting the reconfiguration interval to $3s$ can achieve a good performance (38% reduction on median job completion time compared with F-Hybrid). This is because JobPacker aggregates the shuffle traffic to only a few racks, which generates sparse and skew demand matrices for OCS. This allows OCS to be reconfigured less frequently.

The number of sub-batches In the batch scenario, we divide the entire batch into several sub-batches and sort each sub-batch individually. In this experiment, we varied the number of sub-batches B in the range of $[1, 20]$. Figure 4.24 shows the makespans in JobPacker with different number of sub-batches. All the results are normalized by the results of F-Hybrid. As B increases from 1 to 5, the makespan decreases, since placing the jobs into multiple sub-batches can prevent high competition of resource. However, when $B \geq 5$, increasing B only slightly impacts the performance. This is because dividing the workload into 5 sub-batches is sufficient to prevent the network contention from extremely shuffle-heavy jobs at the same time.

4.4 Summary

Previous studies propose hybrid electrical/optical datacenter networks, which leverage the high-bandwidth optical circuit switch to increase network capacity. In the hybrid networks, the optical circuit switch (OCS) is only used to transfer large flows between racks and the small flows are sent via the traditional electrical packet switch (EPS). Current job schedulers for data-parallel frameworks generate many small flows, which cannot efficiently leverage OCS to accelerate the data transfer. To efficiently utilize OCS, one approach is to aggregate the data traffic by placing the tasks of a data-parallel job on only a few racks. However, there is a tradeoff between task parallelism and traffic aggregation. While we can aggregate all the traffic if we place the tasks on only a few rack, it may sacrifice the parallelism which degrade job performance.

We propose JobPacker, a new job scheduler for data-parallel frameworks in Hybrid-DCN that more fully takes advantage of the OCS to improve job performance. JobPacker aggregates the data transfers of a job to use OCS effectively. Based on the predictable characteristics of recurring jobs, JobPacker has an offline scheduler to find out all feasible (map-width, reduce-width) pairs for every recurring job that can use OCS effectively while achieving sufficient parallelism, find out the best (map-width, reduce-width) pair with the shortest job completion time, and generate the global schedule including which racks and the sequence to run the recurring jobs that yields the best performance. The offline scheduler also has a new sorting method to prioritize the recurring jobs to prevent high resource contention while fully utilizing cluster resource. Based on the offline schedule, an online scheduler places input data and schedules the recurring jobs, and schedules non-recurring jobs to idle resources that are not assigned to recurring jobs. We evaluated JobPacker using large-scale simulation and small-scale emulation on GENI based on production workload, which demonstrates its higher performance in comparison with other schedulers. The results indicate that JobPacker reduces the makespan up to 49% and the median completion time up to 43%, compared to the state-of-the-art schedulers in Hybrid-DCN. In the future work, we would like to consider dependency among jobs in the job scheduling.

Chapter 5

Job Placement and Data Placement in Hybrid Scale-up/out Cluster

Many previous studies [13, 18, 60, 137] show that the small jobs (i.e., jobs with small data to process) may dominate the workloads in production, although the data-parallel frameworks were originally built for large jobs (i.e., jobs with large data to process). For example, the production workloads in Microsoft and Yahoo! clusters have median job input size under 14GB [18, 60, 137] and 90% of jobs on a Facebook cluster have input size under 100GB [13].

Conventionally, the data-parallel clusters consist of a large number of scale-out machines to process the data-intensive jobs. Recent studies [18, 105, 110] advocate to explore hybrid scale-up and scale-out clusters (in short *Hybrid* clusters) to handle current diverse workloads that consist of a large number of small jobs and a few large jobs. Here, scale-up is vertical scaling, which means adding more resources to the nodes of a system, and scale-out is horizontal scaling, which refers to adding more nodes with few processors and RAM to a system. Appuswamy *et al.* [18] evaluated the jobs with different characteristics on scale-up and scale-out machines and found that scale-up is significantly better in some cases, than scale-out. Hence, in this chapter, we aim to design a Hybrid cluster to handle the diverse workloads for high performance.

The remainder of this chapter is organized as follows. In Section 5.1, we present the motivations, benefits and challenges of designing a hybrid scale-up/out cluster. We describe the main design of Hybrid cluster with corresponding job placement and data placement strategies in Section 5.2. We present our experiment evaluation in Section 5.3. Section 5.4 concludes this chapter with remarks on our future work.

5.1 Background

5.1.1 Opportunities, Objectives and Benefits

The workloads in modern production clusters become diverse [32]. Since the performance of a job highly depends on where it runs, we may be able to find a better cost-performance tradeoff for each job with different machines, rather than pure cheap scale-out machines.

The large memory of scale-up machines provides benefits for the jobs with large shuffle data size [18]. In Hadoop, each map and reduce task runs in a JVM. The heap size is the memory allocated to each JVM for buffering data. The map outputs are written to a circular buffer in memory, which is determined by the heap size [18]. When the circular buffer is closed to full, the data is spilled to the local disk, which introduces overheads. Therefore, by increasing the heap size, it is less likely for the data to be spilled to local disk if the heap size is larger, leading to better performance in the shuffle phase. The heap size is 200MB for each JVM by default in Hadoop. However, with the large memory of scale-up machines, we can set a higher heap size with the large memory, which reduces the times to spill.

In addition, the excess memory on scale-up can be used as RAMdisk to store shuffle data to accelerate its read/write. Hence, the scale-up machines can provide more benefits to the jobs with large shuffle data size, as the shuffle is efficient in scale-up.

Previous studies show that scale-up machines can process small jobs faster than scale-out machines [18, 111, 112]. A big data analytic cluster traditionally consists of many cheap scale-out machines. Scale-up machines differs from scale-out machines in that scale-up machines have more powerful CPU and more RAM but less number of CPU cores in one machine. Therefore, scale-up machines may process the small jobs faster but large jobs slower [18, 111, 112] due to the fact that large jobs are generally data-intensive and can be processed faster with higher parallelism on scale-out machines. In this thesis, *we use scale-up (scale-out) job and small (large) job interchangeably.*

To show the benefit of scale-up machines in details, we have conducted a measurement study on Palmetto cluster [39] to compare the performance between scale-up machine and scale-out machines. The scale-up machine is equipped with 24 cores E5-2680V3 CPU, 128GB RAM size, while each scale-out machine is equipped with two 4 cores AMD Opteron 2356 CPU, 16GB RAM size. After some market investigations [45, 123], we find that the price of each selected scale-up machine (i.e., around $5000 each) is similar to 5 selected scale-out machines (around $1000 each)[1]. We deployed several configuration optimizations on the scale-up machine as in [18, 111, 112]. For example, we used half of the RAM (i.e., 64 GB) as RAMdisk to store the shuffle data, and also changed the heap size from default 200MB to 2.5GB.

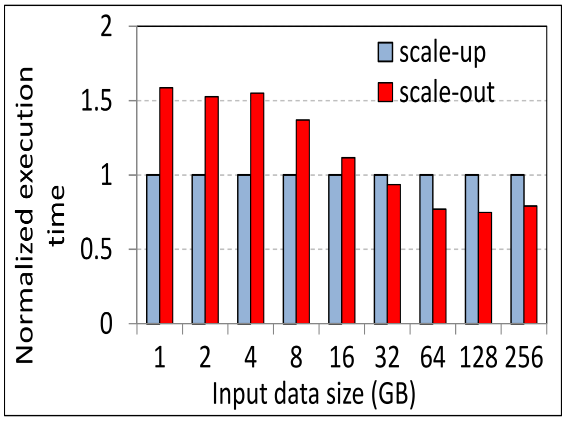
We ran TeraSort and WordCount [16] with different input data sizes on one scale-up machine and five scale-out machines, respectively. All results are the average of

[1]The prices were investigated in Feb, 2017

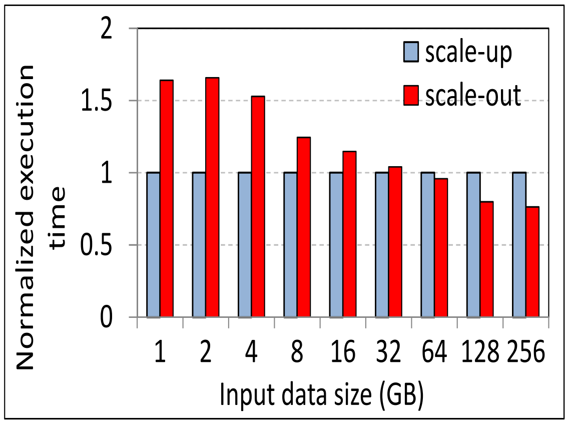
10 runs. Figure 5.1 shows the normalized execution time of TeraSort and WordCount (normalized by the results of scale-up machine) versus different input data sizes. For both jobs, when the input data size is smaller than certain threshold (called *crosspoint threshold*), the scale-up machine can outperform the scale-out machines by up to 70%; otherwise, the scale-out machines can outperform the scale-up machine by up to 35%. Similar conclusions are also observed in [18, 111, 112]. This is because (1) when the input data size is small, scale-up machine benefits the job with more powerful CPU and RAMdisk; (2) as the input data size of the job increases, the total number of CPU cores and memory limit the performance of the job on scale-up machine, while scale-out machines can benefit the job with more CPU cores and higher aggregate memory bandwidth. However, we observe that since the job characteristics of TeraSort and WordCount are different, they have different benefits from scale-up machines. Thus, the crosspoint thresholds for the two jobs are different (32GB for TeraSort and 64GB for WordCount).

In [111, 112], we have conducted thorough experiments for different types of applications (e.g., data-intensive and CPU-intensive) on the scale-up and scale-out machines, and provided an insightful analysis of the performance difference. Our results also suggest that using scale-up machines can provide more promising performance for the jobs with small input data size. However, clearly there is a job size beyond which scale-out becomes a better option. Such a job size is job-specific, depending on the job characteristics. For example, a job that generates more shuffle data may achieve more benefits from scale-up, resulting in a larger crosspoint threshold of job size, compared with a job that generates less shuffle data.

The job characteristics can be predicted with acceptable error [49, 66, 94]. Cluster workloads are known to contain a significant number of recurring jobs [2, 66]. A recurring job is a job that has the same script and is executed whenever new data become available. Therefore, for this kind for recurring jobs, they have similar



(a) TeraSort



(b) WordCount

Figure 5.1: Results of TeraSort and WordCount on scale-up and scale-out.

characteristics, such as input data size, shuffle data size, output data size, and also resource usage. Previous works show that we can predict the job characteristics of these recurring jobs accurately [49, 66, 94]. For example, using the techniques in [94] we can estimate the job input data size with only 6.5% of error on average. This accurate prediction of future job characteristics can help us accurately separate the future jobs to scale-up and scale-out jobs, and hence we can run the jobs accordingly on scale-up or scale-out machines.

The benefits of Hybrid cluster can be summarized as follows.

- For the small jobs, they are executed on the scale-up machines, which process the small jobs faster. Therefore, the performance of the small jobs is improved because they benefit from the scale-up machines on Hybrid cluster.

- For the large jobs, although Hybrid cluster does not have any direct improvement on them, they can also run faster. As the small jobs are executed on scale-up machines, the large jobs can be run on scale-out machines with less resource contention (e.g., CPU and network) from the large amount of small jobs in current big data analytic workloads, and hence, the large jobs can also finish earlier.

5.1.2 Design Challenges

In this section, we identify the key challenges in designing Hybrid clusters to improve the performance of big data analytic clusters. There are two main challenges – job placement challenges (J.1, J.2, and J.3) and data placement challenges (D.1 and D.2).

- **J.1** A proper job placement strategy is essential for the Hybrid cluster. The jobs with different job characteristics may benefit differently from scale-up and scale-out machines. Therefore, we need to adaptively place the jobs to scale-up or scale-out machines based on their job characteristics to achieve the most benefits for the jobs.

- **J.2** The job placement strategy should consider the load balancing. After we schedule the jobs to scale-up or scale-out machines based on their job characteristics, severe load imbalance may occur on different types of machines. For example, suppose a large amount of small jobs are submitted to Hybrid cluster simultaneously, while there are not many large jobs. If we still run the jobs on different machines based on their job characteristics, it leads to overload on the scale-up machines, while under-utilizing on the scale-out machines. Hence, the job placement strategy needs to adaptively schedule small jobs to the other type of machines to avoid overload.

- **J.3** It seems that it is straightforward to address J.2 challenge by moving tasks from scale-up (scale-out) to scale-out (scale-up) when one type of machines are under-utilized. However, this mechanism is not sufficient since the scale-up and scale-out machines have different capability of computing for the tasks, which is a typical problem in heterogeneous clusters. The different computing speed results in significant imbalance progress of tasks within a job, that is, fast machines complete the tasks faster and need to wait for the slow machines to complete the tasks of the same job. This leads to a non-negligible delay and significantly degrades the performance of the job [4, 173].

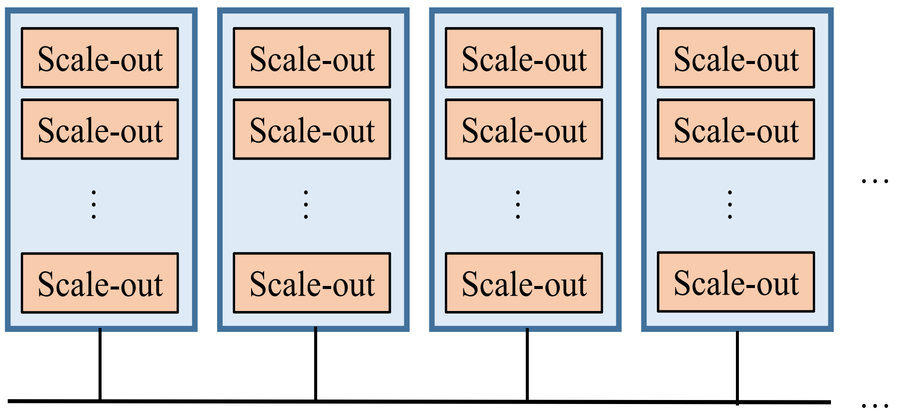
- **D.1** Data locality is an essential factor for high performance [172]. Since we adaptively place a job to scale-up or scale-out machines based on its job characteristics, in order to maintain data locality, we need to accordingly place the data of every job to the machines that the job is supposed to run on.
- **D.2** We cannot simply place the data of scale-up jobs on scale-up machines and the data of scale-out jobs on scale-out machines. This is because the jobs may be adjusted between scale-up and scale-out machines for load balancing according to J.2 challenge. If the adjustment of some jobs occurs, the data locality cannot maintain, which degrades the performance of Hybrid.

5.2 Design of Our Hybrid Cluster

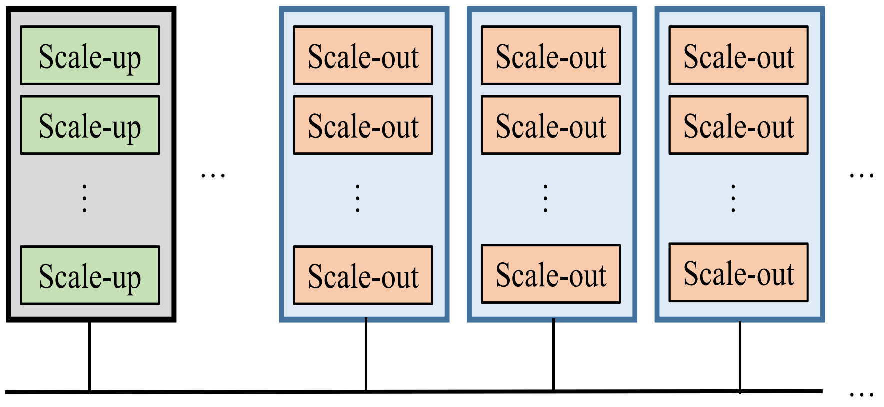
In this section, we present the design of *Hybrid* cluster, including the architecture of the cluster, job placement strategy and data placement strategy for this cluster.

5.2.1 Hybrid Cluster Architecture

In this section, we introduce the architecture for Hybrid cluster. In a traditional Hadoop cluster, it generally consists of a large amount of scale-out machines, as shown in Figure 5.2(a). The architecture of traditional scale-out Hadoop cluster is organized into multiple racks [154]. In Hybrid cluster, we replace part of the scale-out machines by scale-up machines that have the same cost as the scale-out machines. However, there are two questions in the design of Hybrid cluster architecture – where to place the scale-up machines in Hybrid and how many scale-out machines should be replaced by the scale-up machines.



(a) Traditional scale-out cluster.



(b) Hybrid scale-up/out cluster.

Figure 5.2: Traditional scale-out cluster versus hybrid scale-up/out heterogeneous cluster. The traditional cluster consists of multiple racks of scale-out machines, while the Hybrid cluster consists multiple racks of scale-up and also multiple racks of scale-out machines. The Hybrid cluster has a similar cost as the traditional cluster.

Where to place the scale-up machines in Hybrid? We propose the architecture of Hybrid cluster as shown in Figure 5.2(b) – placing the scale-up machines and the scale-out machines on separate racks so that no scale-up and scale-out machines are on the same rack. The reasons why we use this architecture are summarized as follows.

• We can reduce cross-rack network traffic by using this architecture. Jalaparti *et al.* [94] demonstrated that most small jobs in current production workloads can be placed in only one rack without sacrificing the parallelism and compromising the performance. Using this property, we can place the input data of the small jobs in a single rack. As a result, the map tasks of a job can run on this rack for map input data locality and the map output data is also generated in the rack. Subsequently, we can schedule the reduce tasks of the job on the same rack, so that the shuffle data transfers of this job are all within one rack, which reduces the cross-rack network resource. In our designed Hybrid architecture, scale-up machines are in one rack, so that the small jobs placed on scale-up machines can run within one rack, resulting in less cross-rack network traffic from small jobs. Furthermore, this architecture can benefit the large jobs that cannot be run in a single rack, as there is less contention of cross-rack network resource from the small jobs.

- This architecture plays an important role in solving the data placement challenges. For more details, we refer to the data placement strategy in Section 5.2.4.
- This architecture is easy to implement on Apache YARN [16]. We will explain this in details in Section 5.3.1.

How many scale-out machines should be replaced by scale-up machines? Empirical studies in [32, 49, 172] show that the relative proportion of small and large jobs in a cluster is expected to remain *stable* over time. Hence, the cluster operators can determine the number of scale-out machines to be replaced by scale-up machines based on the portion of small jobs in the workload. If there are more small jobs in the workload, the cluster operators can replace more scale-out machines with scale-up machines; otherwise, they replace fewer scale-out machines with scale-up machines. For instance, the scale-out jobs are considerably fewer in typical workloads, but dominate the cluster resource usage (e.g., 80% to 99%) [32, 49, 172]. As a result, in general, the cluster operators only need to replace a few scale-out machines (e.g., 1% to 20%) with scale-up machines to accelerate the small jobs.

5.2.2 Differentiating Small and Large Jobs

Previous studies [49, 66, 94] show that a large number of jobs in production clusters are *recurring* and the job characteristics (e.g., shuffle data size) of the recurring jobs can be predicted with a small error (e.g., 6.5% [94]). Hence, for recurring jobs, we can leverage these predicted job characteristics to determine the types of them. However, for non-recurring jobs, we can only know the number of map/reduce tasks of them. In this section, we introduce how to differentiate small (scale-up) and large (scale-out) jobs for both recurring and non-recurring jobs.

As we mentioned in Section 5.1.1, jobs with different characteristics may benefit differently from scale-up and scale-out machines. To differentiate the jobs into two types based on their job characteristics, the natural thinking is to use machine learning technique, which takes the job characteristics as inputs and predicts each job's type. One question is what job characteristics we should use as inputs. We decide to use the number of map/reduce tasks and shuffle data size as the inputs, due to the reasons below. (1) As shown in Figure 5.1(a), the input data size of a job is clearly one characteristic that affects the performance of the job on different machines. Since input data size of a job is linear to the number of map tasks of the job, we use the number of map tasks. (2) As mentioned in Section 5.1.1, since scale-up machines can benefit the jobs with large shuffle data size because of the large memory size on scale-up machines, shuffle data size is a non-negligible characteristic. (3) The works [18, 110] show that the number of reduce tasks of a job is a factor that affects the performance of the job on different machines.

For the machine learning technique, we use the Support Vector Machine (SVM) [43] model. SVM is a classifier model that maps the feature data (i.e., job characteristics) as points in high-dimensional space, so that the different categories are clearly separated by a gap. More formally, SVM constructs a *hyperplane* to separate the data into two categories, so that the distance from the *hyperplane* to the nearest data point on each side is maximized.

We use SVM rather than other classifiers because of the reasons below.

- The job characteristics (the number of map/reduce tasks, shuffle data size) are all continuous features, which can be handled by SVM.
- SVM is widely used for binary classification, which matches our case that divides jobs into two types.
- The points mapped by the characteristics of jobs may not be linearly separable in space. SVM provides kernel function to create a nonlinear classifier.
- SVM constructs a clear hyperplane to separate the jobs, so that we can calculate the distance between a job to the hyperplane. This property is useful for the job stealing strategy in Section 5.2.3.

While we can consider more job characteristics as inputs and use other machine learning models to classify the jobs, our results in Section 5.3 show that using these three factors (the number of map/reduce tasks, shuffle data size) can already determine the types of the jobs with 96.2% accuracy.

For each Hybrid cluster, it requires us to train a corresponding SVM classifier using sufficient training data first. Then, how we classify recurring and non-recurring jobs is summarized as follows.

Recurring jobs When a recurring job is submitted to the cluster, we can predict its job type using the trained SVM classifier based on the job characteristics (i.e., the number of map/reduce tasks and shuffle data size).

Non-recurring jobs As to the non-recurring job, *only* two job characteristics (the number of map/reduce tasks) are known priori. So, the question is what shuffle data size we should use for non-recurring job.

When determining the type of a non-recurring job, we argue that the occurrence of classifying a large job as a small job would be much worse than the occurrence of opposite. This is because classifying a large job as a small job allows the large jobs to run on the scale-up machines, which causes two issues. First, in Hybrid cluster, we leverage the scale-up machines to accelerate the small jobs, but not large jobs. The performance of large jobs may be greatly degraded, compared with the performance on scale-out machines (see Figure 5.1(a)). Second, the large jobs consume a large amount of resources [49] and run for a long time. If we place the large jobs on scale-up machines when the scale-up machines are under-utilized, the large jobs may occupy all the scale-up machines for a long time. In this case, the small jobs have to wait for the large job and hence the performance of the small jobs are severely degraded. Consider an extreme case that multiple large jobs are submitted to the cluster while there are temporally no small jobs. If we run the large jobs on the scale-up machines, the whole cluster ends up running the large jobs. When some small jobs are submitted to the cluster afterwards, they have to wait for the large jobs for a long time.

On the contrary, classifying a small job as a large job allows the small jobs to run on the scale-out machines. In this case, the only impact is that this small job cannot leverage the scale-up machines to accelerate its execution, which causes minimal impact and is much less severe than the case above.

Hence, when determining the type of a non-recurring job, we aim to avoid the occurrence of classifying a large job as a small job. To achieve this, we treat the shuffle data size of the non-recurring job as 0. This is because scale-up machines can benefit the job with large shuffle data size as mentioned in Section 5.1.1. By treating the shuffle data size of a job as 0, if the job is predicted as “small”, its actual job type will definitely be “small”, since with the job’s actual shuffle data size, the job will obtain more benefits on scale-up machines.

Summary We leverage a SVM classifier to differentiate the small and large jobs based on the job characteristics. The input characteristics for a non-recurring job are the number of map/reduce tasks and a constant shuffle data size (i.e., zero), while the input characteristics for a recurring job are the number of map/reduce tasks and its shuffle data size.

5.2.3 Job Placement Strategy

In this section, we present the job placement strategy accompanied with Hybrid cluster to address the job placement challenges in Section 5.1.2.

Placing the jobs accordingly to scale-up or scale-out job queue (for J.1 challenge). First, when the jobs are submitted to the cluster, the job placement strategy divides the jobs into scale-up job queue and scale-out job queue, using the machine learning technique introduced in Section 5.2.2. The scale-up job queue of jobs are scheduled on scale-up machines, while the scale-out job queue of jobs are scheduled on scale-out machines. Next, Hybrid further sorts the jobs in each queue based on the pre-defined cluster scheduler, such as Fairness [62] and Capacity [26] and put each queue into a queue. When new jobs are submitted, we repeat the previous steps to put the new jobs into corresponding queues.

When a container on a node is available, if it is on a scale-up (scale-out) node, the resource manager (RM) assigns a task from the first job in the scale-up (scale-out) job queue to the container. In this case, if there is not any adjustment (i.e., the job stealing strategy that will be introduced below) on the two queues, the scale-up and scale-out jobs are *restricted* to run on scale-up and scale-out machines, respectively.

Job stealing to balance the job loads for scale-up and scale-out machines (for J.2 and J.3 challenges). In order to solve the J.2 challenge, we propose a job stealing policy that actively steals jobs from the scale-up job queue to the scale-out job queue to achieve load balance.

The job stealing steals the entire jobs instead of individual tasks because of challenge J.3. If the job stealing steals tasks between scale-up machines and scale-out machines, it may occur that the tasks of the same job run on both scale-up and scale-out machines. As aforementioned, running the tasks of a job on different kinds of machines may lead to extremely poor performance for the job [4, 68, 105, 173]. Therefore, we utilize a job-level stealing to handle challenge J.3. Specifically, when the job stealing steals a job between two queues (i.e., scale-up queue and scale-out queue), it finds a job that is not yet started and change its assigned queue to the other queue. Notice that the job-level stealing policy does not incur any overhead since the stolen jobs are still in the queue and not started yet, and no data movement is needed using the data placement strategy in Section 5.2.4.

During job stealing, we propose to *restrict* the large jobs to run only on the scale-out machines and do not allow large jobs to run on scale-up machines, due to the reasons mentioned in Section 5.2.2 that large jobs suffer poor performance on scale-up machines and may occupy the resources of scale-up machines for a long time, which also degrades the performance of small jobs. Besides, in a production workload, most of the jobs are small and the average arrival time between two small jobs is short [18, 32, 49]. In this case, as the small jobs are submitted very frequent, the scale-up machines are expected to be under-utilized for only a short time and will soon become fully-utilized again.

Hence, large jobs are *restricted* to run on scale-out machines, while small jobs are *not restricted* to run on scale-up machines and are allowed to run on both machines, as shown in Figure 5.3. Specifically, we allow the scale-out machines to steal jobs from the scale-up machines, when the scale-out machines are under-utilized. The job stealing can be described in details as follows:

(i) If a scale-up machine requests for a task but there are not any scale-up jobs,

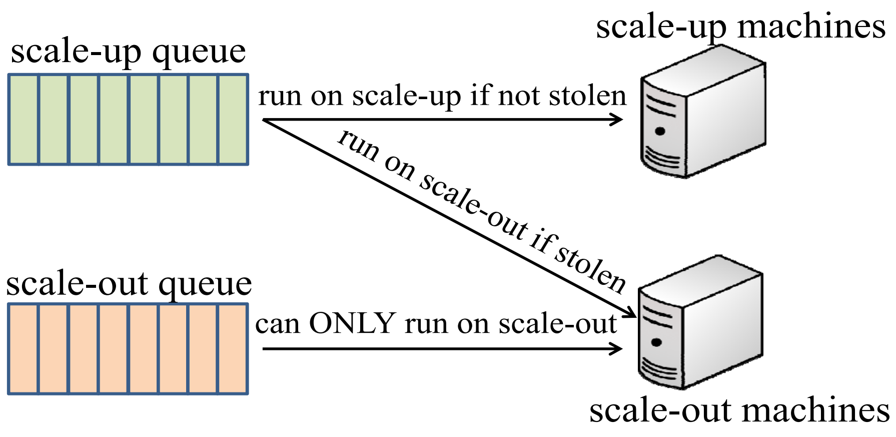

scale-up queue
scale-up machines
run on scale-up if not stolen
run on scale-out if stolen
scale-out queue
can ONLY run on scale-out
scale-out machines

Figure 5.3: Job stealing policy.

the RM delays to schedule any jobs to the scale-up machines until the next scale-up job is submitted.

(ii) If a scale-out machine requests for a task but there are not any scale-out jobs awaiting to schedule, the RM actively "steals" a job from the scale-up job queue. Once a job is stolen to scale-out machines from scale-up machines, all the tasks of this job are restricted to run on scale-out machines.

An important issue is which job to steal. In order not to degrade the performance of the stolen job, it is better to find a scale-up job that is as close to the *hyperplane* as possible, which means that the stolen job is the most similar to scale-out jobs in the queue of scale-up jobs and hence the stealing generates minimal impact to the stolen job.

Thus, to determine which job to steal, the RM computes the distances between the *hyperplane* and the point that represents each scale-up job, and select the one with the smallest distance. After stealing, this job becomes a scale-out job and is restricted to run all its tasks on scale-out machines.

5.2.4 Data Placement Strategy

In this section, we introduce the data placement strategy accompanied with the Hybrid cluster to address data placement challenges in Section 5.1.2.

In our prior study [110], we proposed to configure a remote file system with the hybrid scale-up/out cluster to store all the input datasets, so that both scale-up and scale-out jobs can process the same piece of data. In addition, the remote file system provides easier centralized management for the administrator.

Table 5.1: Four architectures in our measurement.

	Scale-up	Scale-out
OFS	up-OFS	out-OFS
HDFS	up-HDFS	out-HDFS

In Clemson Palmetto HPC cluster [39], it provides the remote file system OrangeFS (OFS) that can be configured with Hadoop by replacing the traditional HDFS. Thus, in [110], we built four architectures as shown in Table 5.1: scale-up machines with OFS (denoted by up-OFS), scale-up machines with HDFS (denoted by up-HDFS), scale-out machines with OFS (denoted by out-OFS), and scale-out machines with HDFS (denoted by out-HDFS). We then measured the performance of representative Hadoop applications (i.e., shuffle-intensive and map-intensive) on these architectures. We aim to see if the use of a remote file system can provide efficient data storage as we expected and whether it brings about any side-effect to scale-up cluster or scale-out cluster.

Through our measurement study, we confirmed the benefits of the remote file system. Thus, we proposed to configure a remote file system with the hybrid scale-up/out cluster to solve the data placement challenges. Real cluster experiments show that in the HPC environment our hybrid scale-up/out architecture with remote file

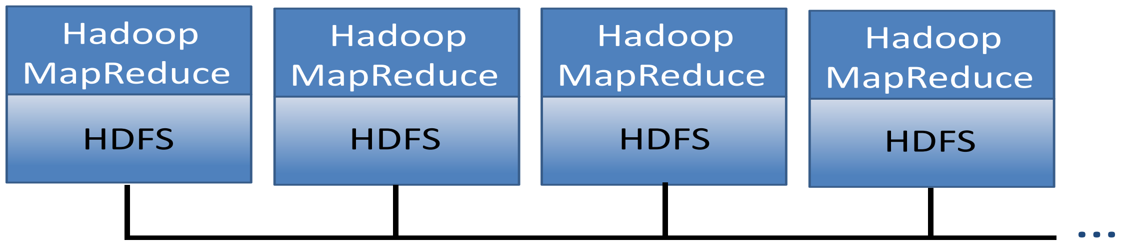


Figure 5.4: Typical Hadoop with HDFS local storage.

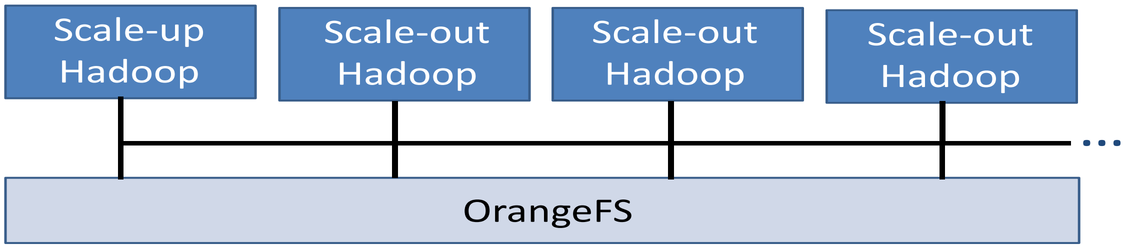


Figure 5.5: Hybrid scale-up/out Hadoop with a remote storage.

system outperforms both the traditional Hadoop architecture with HDFS and with OFS in terms of throughput and job completion time.

Our prior study [110] demonstrates the potential of using remote file system for hybrid scale-up/out architecture. Nevertheless, this solution requires a large amount of remote data read/write. Thus, we present a replication-based data placement strategy with local HDFS in this dissertation to solve the data placement challenges.

In general, there are three replicas for each data block in the cluster [16]. For the common case, HDFS's replication placement policy is to put one replica in one node in one rack, another replica in a node in a different (remote) rack, and the third replica in a different node in the same remote rack. In other words, the three replicas are placed in two racks; one replica in a rack and two replicas in another rack. In this paper, we assume that the default replication factor (i.e., 3) is used. To solve the challenges of data placement, our data placement strategy takes advantage of the replication placement strategy.

Placing the data on both scale-up and scale-out machines (for D.1 and D.2

challenges). We propose to place the replicas of each data block on both scale-up and scale-out machines. Specifically, for a data block, the *first and second replicas* are placed on the scale-out machines, while the *third replica* of the data block is placed on the scale-up machines. Such a placement strategy can solve the D.1 and D.2 challenges.

- First, D.1 challenge is solved since both scale-up and scale-out jobs can locate their data on scale-up and scale-out machines, respectively.
- Second, even when a job is stolen to run different type of machines by the job stealing strategy, the job can still locate its data on the other type of machines, which solves the D.2 challenge.

Notice that with the help of Hybrid architecture in Section 5.2.1, the proposed data placement strategy also satisfies the default replication placement rule in HDFS that places the replicas for each data block in two different racks.

5.3 Performance Evaluation

In this section, we evaluate how the Hybrid cluster performs by real cluster run and large-scale trace-driven simulation. We consider two job submission scenarios, batch and online scenarios [94].

- The batch scenario means that the jobs are submitted to the cluster at the same time. The metric to evaluate the performance in this scenario is the makespan (i.e., the time to complete all the jobs in the batch).
- The online scenario means that each job is submitted to the cluster at a specific job arrival time. The performance metric in this scenario is the average job completion time (i.e., the end time of a job minus the job arrival time).

5.3.1 Experiment Setup and Workload

Real cluster experiments We configured a Hybrid cluster on Palmetto cluster [39]. We used 40 scale-out machines from 4 different racks, each of which contains 10 scale-out machines. We used 2 scale-up machines from the same rack. Therefore, the Hybrid cluster consists of 4 scale-out racks and 1 scale-up rack. The scale-up machine is equipped with 24 cores E5-2680V3 CPU, 128GB RAM size, while each scale-out machine is equipped with two 4 cores AMD Opteron 2356 CPU, 16GB RAM size. After some market investigations [45, 123], we find that the price of each selected scale-up machine (i.e., around \$5000 each) is similar to 5 selected scale-out machines (around \$1000 each)[2]. We deployed several configuration optimizations on the scale-up machine as in [18, 110]. For example, we used half of the RAM (i.e., 64 GB) as RAMdisk to store the shuffle data, and also changed the heap size from default 200MB to 2.5GB. As to the traditional clusters, we used 50 scale-out machines from 5 different racks, each of which contains 10 scale-out machines. Thus, the Hybrid cluster has a similar cost as the traditional cluster.

We implemented the job placement strategy on top of the Apache YARN [16] framework. We ran on Hadoop 2.7.1. We emulated the data placement strategy on our Hybrid cluster. After we configured the Hybrid cluster, we placed the data blocks of each job to the racks based on our data placement strategy before the jobs were submitted to the cluster. For the traditional cluster, we use the default data placement strategy.

Large-scale simulation In order to show the performance of Hybrid cluster in a large scale, we built an event-based simulator (details in Chapter 6) as [49, 94, 128] to simulate the real cluster. We simulated a Hybrid scale-up/out cluster and a traditional scale-out cluster. In the simulation, the traditional cluster consists of 600

[2]The prices were investigated in July, 2017

scale-out machines, which is organized to 20 racks with 30 scale-out machines each. The Hybrid cluster consists of 19 racks of scale-out machines and 1 rack of scale-up machines. In each rack of scale-out machine, there are 30 scale-out machines. In the rack of scale-up machine, it contains 6 scale-up machines. In the simulation, each scale-out machine can run 8 tasks simultaneously, while each scale-up machine can run 24 tasks simultaneously.

Workload To train the SVM classifier for the Hybrid cluster, we replayed a 25428-job Facebook workload trace [32] using the tools provided by [32]. We first ran the 25428 jobs one by one in the workload on scale-up machines and scale-out machines, respectively, and then parsed the logs. We compared the execution time of each job on different machines to determine the type of each job. We used the results as the training dataset for SVM classifier. We applied the radial basis function (RBF) kernel [43] to train the SVM classifier. In order to avoid over-fitting and get the best parameters for SVM with RBF kernel, we used the most common method – cross validation [100].

We used another 24442-job Facebook workload (FB-2010) [32] to validate the SVM classifier and evaluate Hybrid. For the real cluster run, we randomly selected 1000 jobs from this workload (the work in [94] selected 200 jobs in experiments). In the online scenario, the jobs arrived uniformly in a range of $[0, 60] minutes$. For the simulation, we ran the whole workload to evaluate the performance. In the online scenario in simulation, the jobs were submitted to the cluster based on the job arrival time in the trace, which lasts for 24 hours.

Baselines. We compared our Hybrid cluster (Hybrid in short) against the baselines below.

(1) Fair scheduling [62]. The Fair scheduling assigns resources to different jobs in a fair manner, so that each job receives the same resources over time. In order to evaluate the performance of our Hybrid cluster and proposed job placement and da-

ta placement strategies, we compare Hybrid with both Fair scheduling and proposed strategies (**H-FS-with**) against Hybrid cluster with Fair Scheduling but without proposed strategies (**H-FS-w/o**), and traditional scale-out cluster with Fair scheduling (**FS**).

(2) Delay scheduling [172]. The priority of jobs in delay scheduling are the same as Fair scheduling. When the next job does not have a data-local task, Delay scheduler delays assigning resource to the job and looks for a data-local task from the following jobs. In this case, we compare Hybrid with Delay scheduling and proposed strategies (**H-DS-with**) against Hybrid cluster with Delay Scheduling but without proposed strategies (**H-DS-w/o**), and traditional scale-out cluster with Delay scheduling (**DS**).

Note that Hybrid cluster is not only compatible with Fair and Delay, but also compatible with other schedulers such as Corral [94] and Morpheus [97] by sticking to the proposed job placement and data placement strategies in the paper.

5.3.2 Real Cluster Results

Accuracy of SVM

To evaluate the accuracy of SVM, similarly, we replayed the FB-2010 workload on scale-up and scale-out machines, parsed the logs to get the execution times of all the jobs on the scale-up and scale-out machines, and determined the type for every job. We then applied the trained SVM classifier to predict each job in FB-2010. First, we assume all the jobs in FB-2010 are recurring and have predictable job characteristics. In this case, the SVM determines the types of all the jobs in FB-2010 with 96.2% accuracy. Then, we assume 50% of jobs are recurring, while the remaining jobs are non-recurring and only the number of map/reduce tasks is known priori. In this case, the SVM achieves 85.7% accuracy, with no large jobs being classified as small jobs.

Next, we present the experimental results in the real cluster experiments. In the experiments below, 50% of the jobs are recurring and the remaining jobs are non-recurring, unless otherwise specified.

Batch scenario Figure 5.6 shows the makespan of the entire workload in the real cluster run for Hybrid and traditional clusters. We see that H-FS-with and H-DS-with achieve 40% and 32% reduction of the makespan over FS and DS, respectively, which demonstrates the effectiveness of Hybrid cluster. This is because (i) the scale-up jobs in Hybrid cluster are run on the scale-up machines, which process scale-up jobs faster; and (ii) the scale-out jobs in Hybrid cluster benefit from the less contention of resources (e.g., CPU and network) from scale-up jobs. The figures also show that H-FS-with and H-DS-with significantly outperform H-FS-w/o and H-DS-w/o, respectively. Without our proposed strategies, the scale-out jobs can be placed on the scale-up machines, which severely degrades the performance of both scale-up and scale-out jobs. Therefore, H-FS-w/o and H-DS-w/o are even worse than FS and DS, respectively. It indicates the effectiveness of our proposed job placement and data placement strategies.

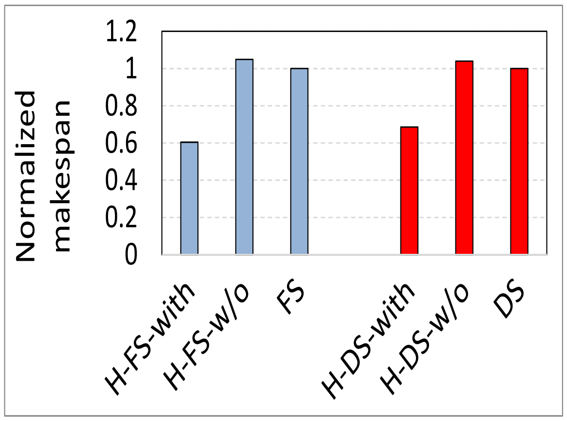


Figure 5.6: Makespan results for all the jobs in the batch scenario.

We further measure the makespan reduction of the scale-up jobs. Figure 5.7 shows the makespan of the scale-up jobs in the real cluster run. We see that H-FS-with and H-DS-with achieve a large reduction of makespan of the scale-up jobs – 53% and 46%

reduction of the makespan over FS and DS, respectively. Hybrid cluster significantly reduces the makespan of scale-up jobs because of two reasons: (i) the scale-up jobs in Hybrid cluster are run on the scale-up machines, which process scale-up jobs much faster; and (ii) the scale-up jobs in Hybrid cluster benefit from less contention of resources with the scale-out jobs. The figures also show that H-FS-with and H-DS-with significantly outperform H-FS-w/o and H-DS-w/o, respectively, which indicates the effectiveness of our proposed job placement and data placement strategies.

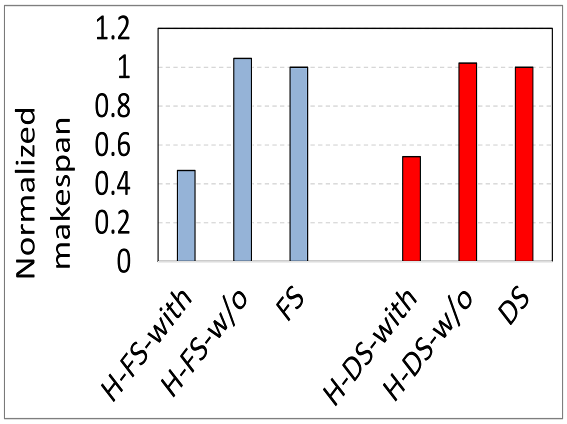


Figure 5.7: Makespan results for scale-up jobs in the batch scenario.

Online scenario In this scenario, the results of H-FS-with and H-DS-with are normalized to FS and DS, respectively. Figure 5.8 shows the average job completion time for the entire workload in the real cluster run for Hybrid and traditional clusters. We see that H-FS-with and H-DS-with achieve 51% and 52% reduction of the average job completion time over FS and DS, respectively, which demonstrates the effectiveness of Hybrid cluster. The figures also show that H-FS-with and H-DS-with significantly outperform H-FS-w/o and H-DS-w/o, respectively, which indicates the effectiveness of our proposed job placement and data placement strategies.

Figures 5.9(a) and 5.9(b) show the average job completion time for scale-up jobs and scale-out jobs in the real cluster run, respectively. Comparing with FS and DS, H-FS-with and H-DS-with reduce the average job completion time of scale-up jobs

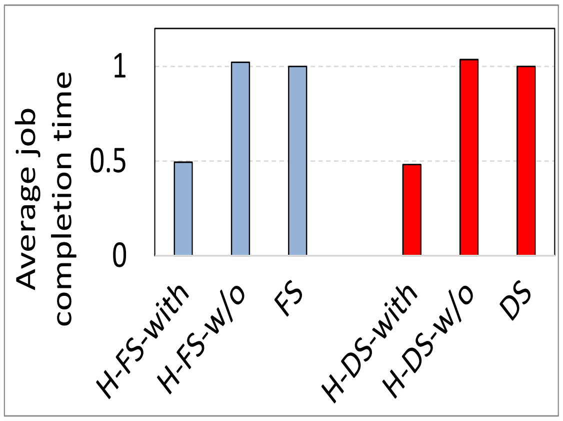
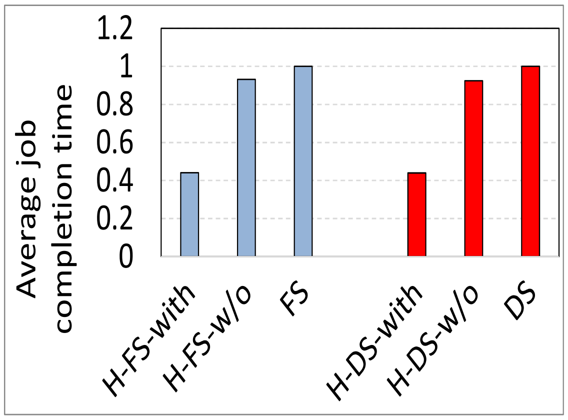
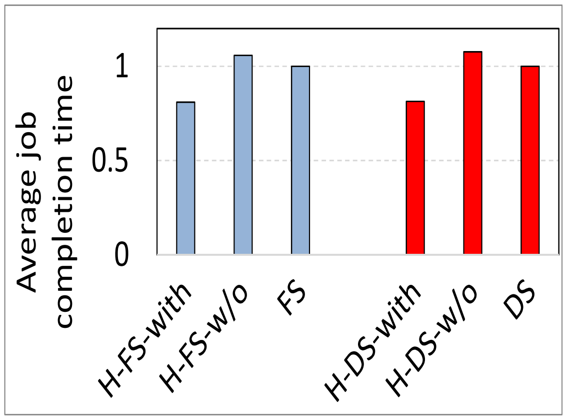


Figure 5.8: Job completion time results for entire workloads in the online scenario.



(a) Average job completion time for scale-up jobs.



(b) Average job completion time for scale-out job-s.

Figure 5.9: Job completion time results in the online scenario.

significantly (more than 55%), while the average job completion time of scale-out jobs is only reduced mildly (up to 20%). This is because in Hybrid cluster scale-up jobs benefit from running on scale-up machines, which process the scale-up jobs much faster. On the other hand, Hybrid cluster does not improve scale-out jobs directly. Instead, the scale-out jobs only benefit from the less contention of resources from scale-up jobs. The figures also show that H-FS-with and H-DS-with significantly outperform H-FS-w/o and H-DS-w/o, respectively, which indicates the effectiveness of our proposed job placement and data placement strategies.

Figure 5.10 shows the cumulative fraction of the completion times of the jobs in

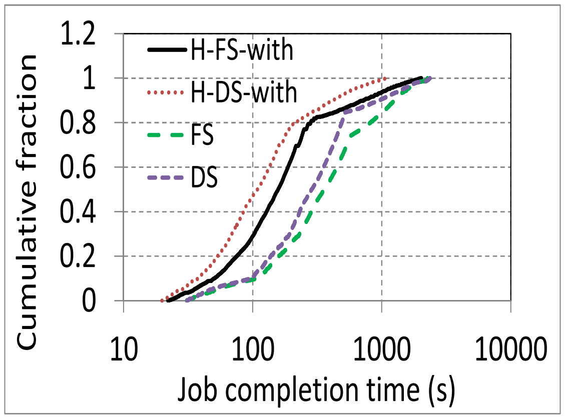


Figure 5.10: CDF of job completion time in the online scenario.

the entire workload in the real cluster run. We see that H-FS-with and H-DS-with outperform FS and DS, respectively, with around 60% improvement at the median for the job completion time. Especially, Hybrid has more significant effect on the jobs with job completion time less than 100s. This is because the scale-up jobs are run on scale-up machines, which process small jobs much faster. On the other hand, we see that Hybrid has less impact on the scale-out jobs, since the scale-out jobs only benefit from the less contention of resources from scale-up jobs. We also observe that for some jobs with extremely large data sizes, their performance on Hybrid cluster is even worse than that on the traditional cluster. This is because in Hybrid, we replace some scale-out machines in the traditional cluster by scale-up machines, resulting in fewer machines allowed to process the scale-out jobs. Therefore, the jobs with extremely large data sizes may have worse performance in Hybrid than in the traditional cluster. Figure 5.10 does not have the results of H-FS-w/o and H-DS-w/o, since the results of H-FS-w/o and H-DS-w/o are similar to FS and DS, which makes the figures difficult to distinguish.

Summary:

- In the batch scenario, Hybrid cluster can reduce the makespan of the entire workload and the scale-up jobs significantly.

- In the online scenario, Hybrid cluster can reduce the average job completion times of all the jobs significantly.

5.3.3 Trace-driven Simulation Results

Next, we present the experimental results in the trace-driven simulation. In the experiments below, 50% of the jobs are recurring and the remaining jobs are non-recurring, unless otherwise specified.

Figure 5.11 shows the makespan of the entire workload in the large-scale simulation for Hybrid and traditional clusters. Interestingly, in the simulation, the makespan of the entire workload is reduced by only 27% and 20% in H-FS-w/o and H-DS-w/o, compared with FS and DS. We will explain the reason below.

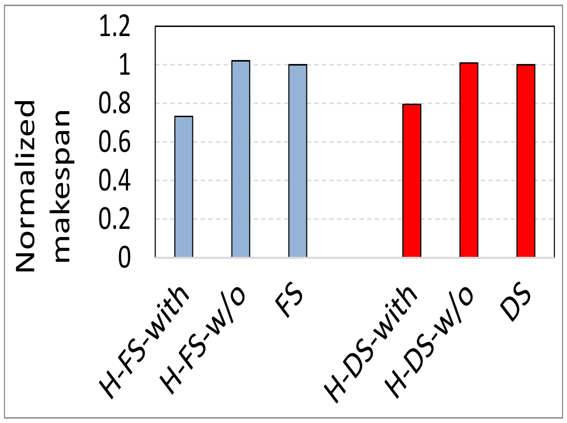


Figure 5.11: Makespan results for all the jobs in the batch scenario.

Figure 5.12 shows the makespan of the scale-up jobs in the large-scale simulation, which are consistent with the results in the real cluster run due to the same reasons.

In the simulation, the makespan of the entire workload is reduced by only 27% and 20% in Hybrid. This is because the entire FB-2010 workload is highly skewed. Most of the jobs (more than 85%) in the workload have input data sizes less than 100MB, while some jobs in the workload have input data sizes extremely large (more than 5TB). These large jobs are all characterized as scale-out jobs and dominate

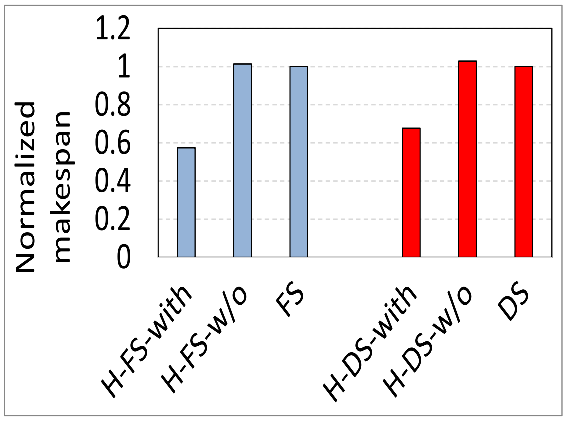


Figure 5.12: Makespan results for scale-up jobs in the batch scenario.

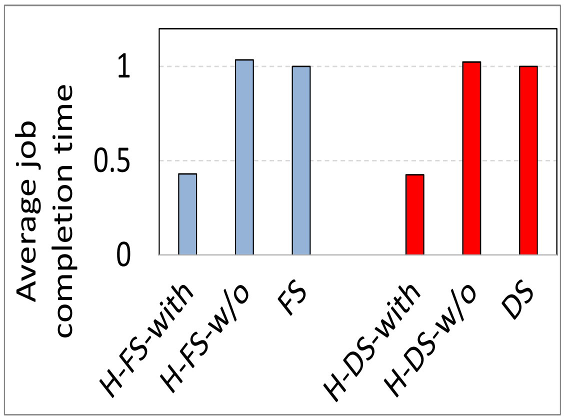
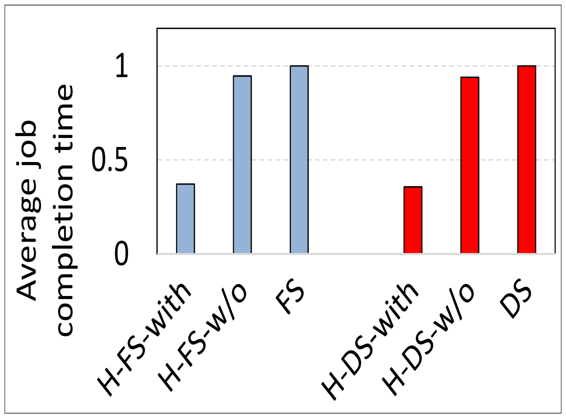


Figure 5.13: Job completion time results for entire workloads in the online scenario.

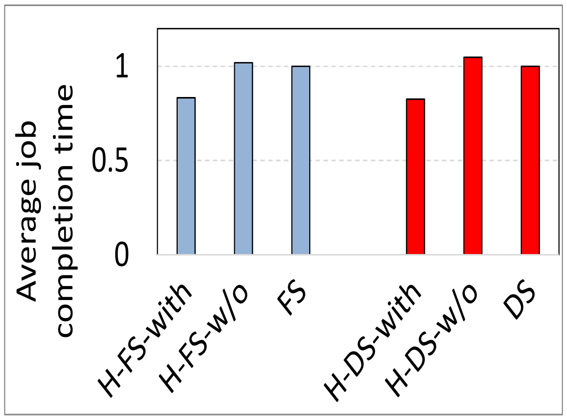
the makespan of the entire workload. This means that in the simulation, after all the scale-up jobs are completed, there are still scale-out jobs running in the cluster, which dominates the makespan of the workload. Therefore, although Hybrid cluster reduces the makespan of scale-up jobs significantly, it reduces the makespan of the entire workload by only 27% and 20% in the simulation.

Figure 5.13 shows the average job completion time for the entire workload in the simulation. We see that the Hybrid cluster with a large scale simulation is consistent with the results in the small-scale real cluster run due to the same reasons.

Figures 5.14(a) and 5.14(b) shows the average job completion time for scale-up jobs and scale-out jobs in the simulation, respectively. We see that the Hybrid cluster



(a) Average job completion time for scale-up jobs.



(b) Average job completion time for scale-out jobs.

Figure 5.14: Job completion time results in the online scenario.

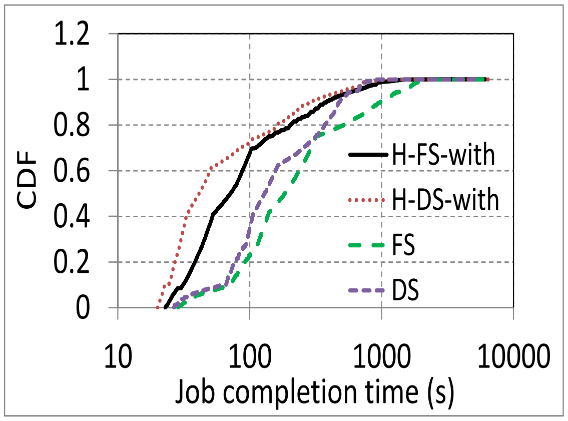


Figure 5.15: CDF of job completion time in the online scenario.

with a large scale simulation is consistent with the results in the small-scale real cluster run due to the same reasons, which indicates the effectiveness of our proposed job placement and data placement strategies.

Figure 5.15 shows the cumulative fraction of completion time in the simulation. It confirms our observations in the real cluster run due to the same reasons.

Summary:

- In the batch scenario, Hybrid cluster can reduce the makespan of the entire workload and the scale-up jobs significantly.

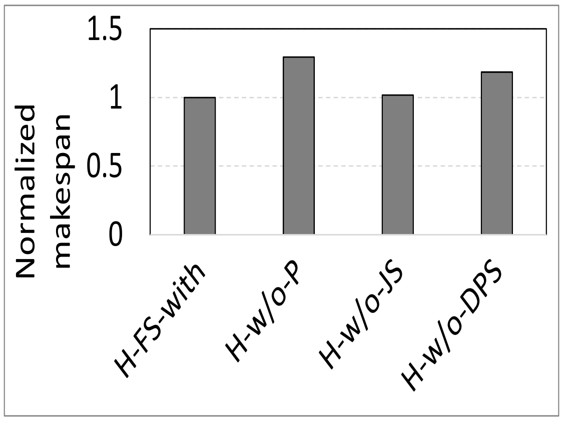
- In the online scenario, Hybrid cluster can reduce the average job completion times of all the jobs significantly.

5.3.4 Effectiveness of Each Strategy

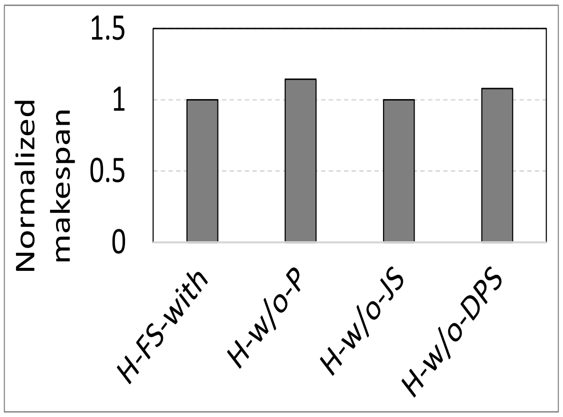
In this section, we aim to investigate the effectiveness of the job placement and data placement strategies in Hybrid. We measure the performance of Hybrid without our job placement strategy that places jobs accordingly to scale-up or scale-out machines (*H-w/o-P*), Hybrid without job stealing policy (*H-w/o-JS*), and Hybrid without our data placement strategy (*H-w/o-DPS*). In this section, we only measure them on H-FS-with and normalize the results to H-FS-with. Specifically,

- H-w/o-P uses the Fair scheduling to schedule the jobs, which does not take into account the job characteristics.
- H-w/o-JS does not adopt the job stealing policy even when the scale-out machines are under-utilized.
- H-w/o-DPS does not use our replication placement technology, and only uses the default random replication placement in HDFS.

Batch scenario Figure 5.16(a) shows the makespan of H-FS-with, H-w/o-P, H-w/o-JS, and H-w/o-DPS in the real cluster run. We see that the makespan of H-w/o-P is increased by 29%, when comparing with H-FS-with. It indicates that the performance of H-w/o-P is even worse than the traditional cluster with Fair scheduling (shown in Figure 5.6) because of the following reasons. Without our job placement s-trategy to place the jobs accordingly to scale-up or scale-out machines, (i) the scale-up jobs may be assigned to the scale-out machines and hence they cannot take advantage of the scale-up machines; and (ii) some tasks of scale-out jobs run in the scale-up machines, which is very slow and hence results in poor performance. We also observe that H-w/o-JS actually provides the same performance as H-FS-with in the batch



(a) Makespan without strategies in the real cluster run.

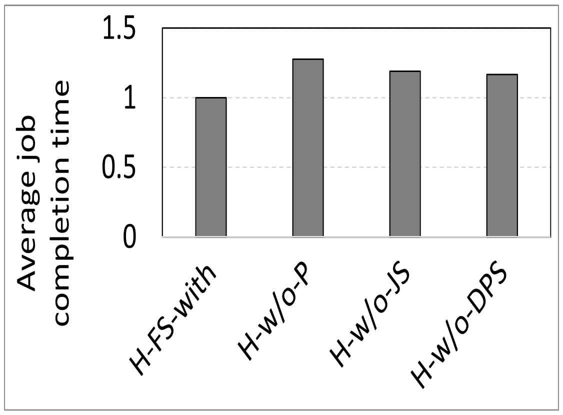


(b) Makespan without strategies in the simulation.

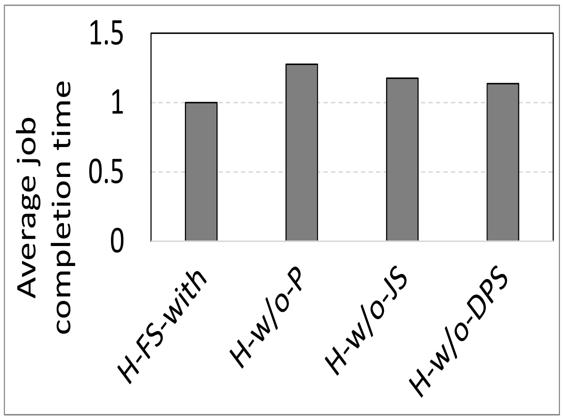
Figure 5.16: Measurement results of each strategy in Hybrid.

scenario. This is because in the batch scenario, the large jobs are all submitted to the cluster, which makes the scale-out machines fully utilized all the time. Thus, the job stealing actually does not have any effect on this FB-2010 workload as the scale-out machines are never under-utilized. On the other hand, the makespan of H-w/o-DPS is increased by 18%, when comparing with H-FS-with. This is because without our data placement strategy, some tasks may fail to maintain data locality, which degrades the performance. Figure 5.16(b) shows the makespan of H-FS-with, H-w/o-P, H-w/o-JS, and H-w/o-DPS in the large-scale simulation, which are consistent with the results in the real cluster run due to the same reasons.

Online scenario Figures 5.17(a) and 5.17(b) show the average job completion time of H-FS-with, H-w/o-P, H-w/o-JS, and H-w/o-DPS in the real cluster run and in the simulation, respectively. We see that when comparing with H-FS-with, the average job completion time of H-w/o-P is increased by 27%, while the average job completion time of H-w/o-DPS is increased by 16%. This is because of the same reasons in Figures 5.16(a) and 5.16(b). For H-w/o-JS, we see from the figures that it increases the average job completion time on H-FS-with in the online scenario by around 19%. This is because the job stealing policy steals scale-up jobs to run on



(a) Average job completion time without strategies in the real cluster run.



(b) Average job completion time without strategies in the simulation.

Figure 5.17: Measurement results of each strategy in Hybrid.

scale-out machines when the scale-out machines are under-utilized. Thus, the job stealing policy helps reduce the response time of the jobs (i.e., the start time of the job minus the job arrival time).

Summary: Through the results, we demonstrate the effectiveness of job placement and data placement strategies for Hybrid cluster and conclude that when accompanying with our strategies, Hybrid cluster can achieve better performance.

5.3.5 Sensitivity Analysis

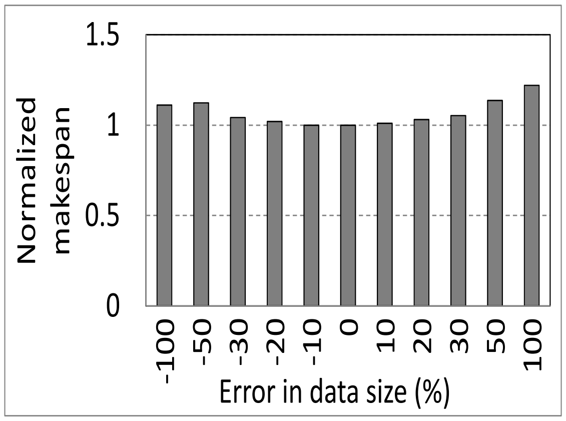
The benefits of Hybrid cluster depend on the prediction of job characteristics. In this section, we evaluate the robustness of Hybrid cluster to the prediction error. We only measure the results for H-FS-with and the results are normalized to the results when there is no error. In this experiment, the error rate is defined as

$$\frac{prediction\_value - real\_value}{real\_value}. \tag{5.1}$$

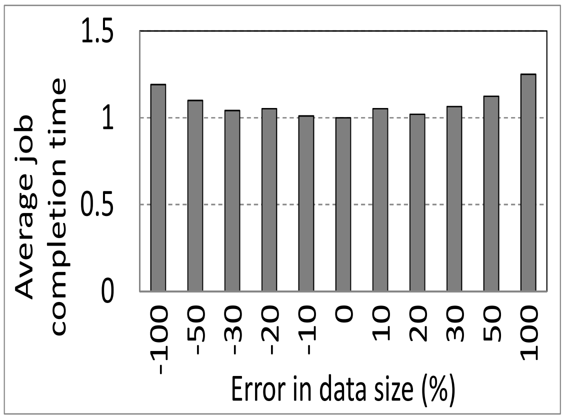
A negative error rate means $prediction\_value < real\_value$. For the error rate of "-100", it means $prediction\_value << real\_value$. For example, suppose the *prediction_value* is 0.5 GB and *real_value* is 16GB. Thus, the error rate is -96.875%≈-100%.

Error in predicted shuffle data size Although previous studies show that the prediction error of job characteristics can be as low as 6.5% on average [94], we varied the error rate of predicted job shuffle data size by up to 100%. The results below are normalized to the result of H-FS-with without error.

Figures 5.18(a) shows the makespan versus error in data size in the batch scenario. The figure indicates that Hybrid cluster can maintain very similar makespan when the error is less than 30%. However, as the error increases, the makespan is also increased. This is because as the error increases, the job type of more jobs may be wrongly decided. This could cause some scale-out jobs to be run on scale-up machines, which reduces the benefits of Hybrid cluster. Figures 5.18(b) shows the average job completion time versus error in data size in the online scenario. Similar results are observed: the Hybrid cluster maintains similar performance when the error is low; however, as the error increases, the performance gets worse. The results demonstrate that the performance of Hybrid cluster is not quite sensitive to small error in predicted job data size.



(a) Makespan versus data size prediction error rate.



(b) Average job completion time versus data size prediction error rate.

Figure 5.18: Sensitivity analysis.

5.4 Summary

We observe that the current workloads for big data analytic have diverse characteristics. It is advocated in previous studies that we can process the jobs with different characteristics using different machines. In this paper, we aim to design a Hybrid scale-up/out cluster to improve the performance of big data analytics. However, there are two main challenges – job placement and data placement challenges. In the job placement challenge, it requires us to adaptively place a job to either a scale-up or a scale-out machine to achieve the most benefits for the jobs. For the data placement, it is critical to guarantee that the scale-up (scale-out) jobs have their data on scale-up (scale-out) machines to maintain high data locality.

We propose a Hybrid architecture with corresponding job placement strategy and data placement strategy to address the challenges. In Hybrid architecture, we separate scale-up and scale-out machines to different racks. In the job placement strategy, we actively place the jobs to different machines based on their characteristics. In data placement strategy, we use replication placement technique to maintain high data locality. We implement Hybrid on top of YARN. We evaluate Hybrid by running a production workload (FB-2010) with both real cluster run and large-scale trace-driven simulation. The results show that accompanying with our strategies, Hybrid cluster can reduce the makespan by up to 40% and the median job completion time by up to 60%, compared to tradition scale-out cluster with state-of-the-art schedulers. In the future, we plan to further explore a more complex Hybrid cluster with more kinds of machines to fit more diverse workloads.

Chapter 6

Simulator Framework

In the evaluations in previous chapters, we built a simulator to evaluate our proposed schedulers in large scale. In this chapter, we will introduce the implementation details of the simulator.

The prototype of our simulator is based on Hadoop [84]. Our simulator is *trace-driven* and aims to mimic the work flows in a typical Hadoop cluster – how the tasks of jobs are scheduled and processed, and how the data is transmitted through network. *It is worth mentioning that the focus of this dissertation is not to build fancy simulator for Hadoop.* Thus, the goal of our built simulator is NOT to provide accurate per-job performance prediction, but to serve as an initial and intermediate step to evaluate the impact of scheduling decisions on the system-level performance in a large scale, so that we can have an *accurate enough* understanding of the performance of the schedulers.

The remainder of this chapter is organized as follows. We describe the main design of our simulator in Section 6.1. Section 6.2 presents the validation of our simulator. Section 6.3 concludes this chapter.

6.1 Design

6.1.1 Simulator Framework

Figure 6.1 shows the simulator framework. The simulator consists of a resource manager, simulated Hadoop nodes and a flow-level simulator. The input of the simulators is a job trace, which is a record of various parameters and characteristics of jobs from a real Hadoop cluster, such as job arrival time, the number of map/reduce tasks, task duration, and input/shuffle/output data size. The resource manager then generates jobs at their arrival time. Besides, the resource manager periodically receives heartbeat from the simulated nodes about their resource status and the status of running tasks on the nodes. Based on the pre-defined scheduling algorithm, the resource manager determines which nodes to schedule the tasks of jobs in the queue.

Each node has a certain number of containers, based on node specification. On arrival of a task assignment on a node, the simulator determines whether data transfer is involved. If yes, the data transmission information is sent to the flow-level simulator. When the corresponding transmissions are finished, the node *runs* the task by letting the container on the node wait a certain time (as specified in the job trace). The above work flows are exactly the same as the typical Hadoop cluster.

6.1.2 Job Trace

The workload trace is derived from the SWIM Facebook workloads [32]. The original trace contains the jobID, job submission time, and input/shuffle/output data size of the jobs. However, it does not provide any information about the duration of each map and reduce task, which is important in our simulation. In order to obtain the duration of each map and reduce task, we can use the synthesized execution framework in [32] to generate the corresponding jobs based on the trace and run the

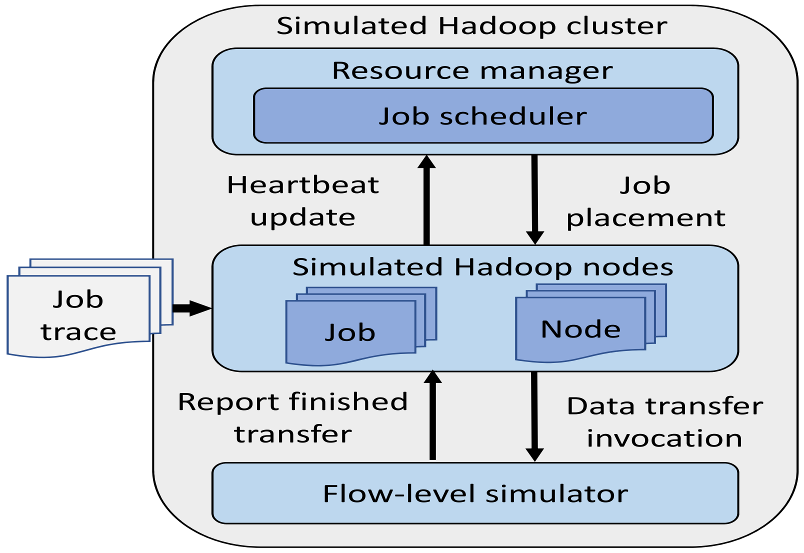


Figure 6.1: Simulator framework.

jobs one by one on a single-node cluster. We choose to carry out experiments on a single-node cluster because it allows us to collect the duration of each map and reduce task without considering the network factor. We then collect the Hadoop logs as the job trace.

6.1.3 Resource Manager

The resource manager is responsible for scheduling the tasks of jobs. The resource manager first checks whether there are new jobs submitted to the cluster and initializes the corresponding jobs if needed. Then, after receiving every heartbeat from the nodes, it checks whether there are any empty containers in the cluster and assigns the tasks to the containers based on the scheduling algorithm. Here, the scheduling algorithm is based on the scheduler we specify.

6.1.4 Modeling the Nodes

The simulator creates a number of simulated nodes based on the cluster specification. Each node consists of certain resources. As in many previous works [3, 48, 74, 94, 128], the simulator models each node's resources as a certain number of containers, each of which can run a task at a time. Notice that the nodes in the simulator have the same number of containers as the single node used to run the trace.

Each node keeps track of the status of the tasks running on it. It periodically sends the heartbeat message to the resource manager about its current resource status and completions of tasks.

In addition to the computing resources (i.e., containers), each node also has a disk that stores the input data blocks. For a map task, if its input data block is located on the same node as where it runs, then data transmission is not needed. Our simulator supports the input data replication of Hadoop, namely there are three copies placed on three nodes for each input data block. For each map task, its input data blocks are placed based on either the default data placement strategy of Hadoop (as introduce in Chapter 2), or the specified data placement strategy.

6.1.5 Modeling the Tasks

When a node receives the message of running a task, the simulator models the task in the sequence of events as follows: (i) The simulator checks if its input data is needed to fetch from remote hosts. If yes, a data request is sent to the flow-level simulator, i.e., generating new flows. The task waits until a message from the flow-level simulator indicating that the data is received. (ii) The task is then *run* on the node by letting the container on the node wait a certain time (as specified in the job trace). (iii) Finally, the heartbeat message indicating the completion of the task is sent to the resource manager.

6.1.6 Flow-level Simulator

Depending on granularity of simulation models, network simulators can be classified into two categories: packet-level simulator and flow-level simulator.

The packet-level simulator mimics behavior of every packet in a network, such as the packet arrivals and departures. While packet-level simulator can more accurately simulate the real network, previous study [6] show that existing packet-level simulators, such as ns-2 [151] and ns-3 [125], are not suitable for simulating large scale network environments due to the computational complexity. For instance, a simulation with 8,192 hosts each sending at 1Gbps would have to process $2.5 * 10^{11}$ packets for a 60 second run. In a packet-level simulator, it would take 71 hours to simulate the transmission of 1 million packets per second using, which is too long for a simulation [6].

Thus, we built a flow-level simulator rather than the packet-level simulator in our case to coarsely model the behavior of the network. The flow-level simulator mimics behavior of every flow in a network and aims to capture the flow transfer duration from flow bandwidth allocation procedures. The flows arrive at the network according to the placement of a tasks. When a task is assigned to a node, if it needs to fetch data from remote node(s), then a new flow arrives. According to different datacenter network architectures, we have two different flow-level simulators.

6.1.6.1 Traditional EPS Datacenter Network Architecture

In traditional EPS datacenter network architectures (used in Chapters 3 and 5), the flow-level simulator proceeds in discrete time ticks. At each tick, the simulator updates the rates of all flows in the network and generates new flows if needed. To determine the rate of a flow at a tick, an underlying mechanism that effects "statistical bandwidth sharing" is used [6], i.e., the rate of a flow is adjusted in a max-min fairness manner based on the bandwidth capacity and the number of flows that it shares the bandwidth with at that tick. At each time tick, a flow transmits data through the network according to the rate allotted to it. When a flow completes sending all its bytes, the flow departs and the simulator notifies the corresponding node so that it can continue to process tasks.

6.1.6.2 Hybrid-DCN

In Hybrid-DCN, in addition to the conventional EPS datacenter network architecture, there is an OCS network. Thus, the flow-level simulator consists of two networks. For the EPS network, the flow-level simulator is the same as the one in Section 6.1.6.1. For the OCS network, given a demand matrix M_o at a time, the simulator needs to compute the optimal configuration for OCS and then adaptively reconfigure OCS to determine which racks are connected, so that the size of the total traffic sent via OCS is the maximum. As in many previous studies [63, 162], to compute the optimal configuration for OCS, we used Edmonds' algorithm [58], which can compute the optimal configuration in polynomial time.

When a task is assigned to a node, if it needs to fetch data from remote node(s), then a new flow arrives. If the new flow exceeds the elephant flow threshold, it will use the OCS network and vice versa.

6.2 Simulator Validation

We would like to emphasize that instead of building a complex simulator that can accurately reflect the job completion times of all the jobs, we are more interested in implementing a simulator that provides an accurate enough estimation of the performance of the cluster with the proposed schedulers with respect to the performance of the cluster with the baseline schedulers.

In this section, we validate our simulator by comparing the performance results under the simulator and a 40-node cluster in CloudLab [41] with traditional datacenter network architecture. The 40 nodes were organized in 8 racks with interconnection of 1Gbps Ethernet. Each rack contains 5 nodes and each node has 1Gbps Ethernet interconnect, resulting in a 5:1 oversubscription ratio. The number of containers on each node was set to 16 and the input block size was set to 128MB. The parameters in the simulator were set to the same accordingly.

The metric we used to validate our simulator with the real cluster is defined as

$$ratio = \frac{\text{Result}_{Scheduler1}}{\text{Result}_{Scheduler2}}, \tag{6.1}$$

where *Result* is the normalized throughput or the normalized average job completion time (e.g., normalized to the *Result* of Fair scheduler in our case). Scheduler1 and Scheduler2 are the proposed scheduler and baseline scheduler, respectively. The intuition of this metric is that, if the difference between the *ratio* in the simulation and the *ratio* in the real cluster is within an acceptable range, then we can claim that our simulator can present an *accurate enough* estimation of the performance of the proposed scheduler.

To validate the simulator, we used 200 jobs selected from the Facebook trace [32] as in Section 3.3. We exploited the method in Section 6.1.2 to first collect the history logs as the job trace. We ran the 200 jobs for 20 times with different schedulers (Fair, Delay, SW-delay, NAS in Section 3.3) in the simulator and in the real cluster, respectively. We computed the average *Result* (i.e., throughput and average job completion time) of these 20 runs for different schedulers.

Table 6.1: Throughput comparison.

Throughput	NAS/Fair	NAS/Delay	NAS/SW-delay
ratio (simulation)	1.58	1.39	1.27
ratio (real cluster)	1.63	1.47	1.30

Table 6.2: Average job completion time comparison.

Average job completion time	NAS/Fair	NAS/Delay	NAS/SW-delay
ratio (simulation)	0.62	0.69	0.71
ratio (real cluster)	0.57	0.64	0.68

Tables 6.1 and 6.2 show the *ratio* results with respect to throughput and average job completion time, respectively. We see that the difference of *ratio* between the simulation and real cluster is in the range of 0.03-0.08 for throughput and 0.03-0.05 for average job completion time. The results demonstrate that our simulator can present an *accurate enough* estimation of the performance of the proposed scheduler.

6.3 Summary

In this chapter, we presented the implementation details of our simulator that aims to provide an *accurate enough* estimation of the performance of the cluster with the proposed schedulers with respect to the performance of the cluster with the baseline schedulers. Based on this goal, we define a new evaluation metric called *ratio* to validate our simulator, where the *ratio* reflects the performance of the cluster with the proposed schedulers with respect to the performance of the cluster with the baseline schedulers. We validated our simulator by comparing the *ratio* under the simulator and a 40-node cluster in CloudLab [41]. Results show that our simulator can present an *accurate enough* estimation of the performance of the proposed scheduler.

Chapter 7

Related Work

Over the past decades, more and more applications are scaling out to large clusters to deal with the rapidly increasing data volumes. There are concentrated efforts in supporting efficient execution of data-parallel applications. In this chapter, we discuss existing efforts in improving the performance of data-parallel applications.

The remainder of this chapter is organized as follows. Section 7.1 discusses the efforts in building high-capacity, low-latency datacenter networks. Section 7.2 describes studies of workload characterization in data-parallel clusters. Section 7.3 presents the efforts in improving the performance of data-parallel clusters. Section 7.4 introduces the works about job schedulers in data-parallel clusters. Section 7.5 presents the research of job schedulers in HPC, Grid, and multiprocessor systems.

7.1 Datacenter Networks

With the tremendously increasing bandwidth demand of distributed data-parallel applications, much research has been focusing on building high-capacity, low latency datacenter networks to support modern data-parallel clusters [40, 63, 77, 80, 81, 124, 144, 162]. To take advantage of the datacenter networks, many efforts have gone into designing solutions to minimizing the flow completion times or to ensuring fairness [6, 8, 9, 19, 20, 51, 57, 79, 82, 87, 93, 95, 102, 106, 132, 133, 140–143, 158, 169, 170].

However, despite the rapid innovations in the infrastructure and network scheduling of datacenter networks, they are still application-agnostic. Data-parallel applications care about *all* the flows, while the general networking solutions schedule each flow independently. To address such mismatch in application-level and network-level, several flow-level techniques [35–38, 53, 89, 114, 121, 165] have been proposed to decrease the communication time of data-parallel frameworks. They abstract the flows in data-parallel framework as *Coflow*, a collection of flows that follow the same objectives, such as shuffling stage within a MapReduce job. Based on the Coflow abstraction, these studies design flow-level techniques to schedule Coflows to shorten the Coflow completion time (i.e., the completion time of the last flow in a Coflow). The efforts in such network-level schedulers are orthogonal to our dissertation and can be combined with our work for better performance.

7.2 Workload Characterization of Data-parallel Clusters

Many efforts have been devoted to characterizing the workloads on MapReduce and cloud platforms. Chen *et al.* [32] analyzed and compared two production MapReduce traces from Yahoo and Facebook in order to develop a vocabulary for describing MapReduce workloads. Their another work [31] characterized new MapReduce workloads, which are driven in part by interactive analysis and with heavy use of query-like programming frameworks such as Hive on top of MapReduce. Ren *et al.* [136] characterized a workload from Taobao at the granularity of job and task, respectively, which provides an understanding of the performance and the job characteristics of Hadoop in the production environment. Kavulya *et al.* [99] analyzed the characteristics of the workload based on MapReduce logs from the $M45$ supercomputing cluster. Mishra *et al.* [119] described the workload characterization and its application to the Google Cloud Backend. It includes behavior characteristics such as CPU and bandwidth. Williams *et al.* [168] proposed TideWatch, a system that enables a cloud provider to monitor and predict the cyclicality of cloud workloads. The output of TideWatch, in particular the period duration and grouping of virtual machines, can be exploited to optimize VM management in a number of dimensions. CAppuswamy *et al.* [18] conducted an evaluation of representative Hadoop jobs on scale-up and scale-out machines, respectively. They found that scale-up machines achieve better performance for jobs with data size at the range of MB and GB.

These works demonstrate that the current workloads in data-parallel clusters are highly diverse, which forms the motivations of our dissertation.

7.3 Performance of Data-parallel Clusters

Many works focus on improving the performance of the data-parallel clusters from different aspects such as job scheduling [26, 62, 75, 86, 176], data placement [13, 15, 61, 109, 111, 112], intermediate data shuffling [42, 44, 155, 166] and improving small job performance [60]. The work in [155] replaces HDFS with the Lustre file system and places shuffle data in Lustre. MapReduce online [42] sends shuffle data directly from map tasks to reduce tasks without spilling the shuffle data to the disks in order to reduce the shuffle phase duration. Camdoop [44] performs in-network aggregation of shuffle data during data forwarding in order to decrease the network traffic. Wang *et al.* [166] proposed JVM-Bypassing shuffling for Hadoop to avoid the overhead and limitations of the JVM. Unlike these previous works that focus on improving the performance of traditional Hadoop using different methods, we focus on designing appropriate job schedulers to improve the performance of diverse workloads in data-parallel clusters with different architectures.

7.4 Job Schedulers in Data-parallel Clusters

Several efforts [26, 62, 71] aim to achieve fairness among jobs or users for the map tasks. Fair scheduler [62] is most widely used in real clusters to achieve fairness among jobs, i.e., each job occupies approximately the same amount of resources. Capacity scheduler [26], developed by Yahoo, shares the available resources fairly among multiple organizations according to their computing requests to meet the SLA (service level agreement). Dominant Resource Fairness scheduler [71] achieves a max-min fairness for multiple resources (e.g., CPU, memory and I/O). In a multi-resource cluster, the scheduling is determined by the user's dominant share, which is calculated by the maximum share of the user's allocated resources. Delay scheduler [172] reduces network traffic by solving the tradeoff between fairness and map input data locality. When the job selected based on fairness cannot launch a local task, Delay scheduler delays the job a small amount of time and launches a local task instead to maintain high data locality. Quincy [91] calculates the cost of each assignment of map tasks and nodes based on locality and fairness, and uses a min-cost flow algorithm to find the optimal scheduling assignment. However, the above schedulers focus on the scheduling of map tasks, which cannot address the network problems caused by shuffle-heavy jobs. In the dissertation, we focus on designing job schedulers in different architectures of data-parallel clusters to improve the cluster performance.

Some previous studies consider the scheduling of reduce tasks to improve the cluster performance. Chen *et al.* [29] proposed a theoretical linear programming framework to model the map-shuffle-reduce performance of MapReduce and developed a constant factor approximation algorithm to solve the model, which determines the order of map and reduce tasks on each processor to minimize the task response time. Guo *et al.* [83] presented *ishuffle* that actively pushes map output data to nodes and flexibly schedules reduce tasks considering workload balance. In order to address the monopolizing behavior of long reduce tasks, Wang *et al.* [167] presented a Preemptive ReduceTask mechanism, which enables the reduce tasks to be preempted in a work-conserving manner, i.e., without dumping data. Coupling scheduler [147] gradually launches reduce tasks based on the progress of map tasks rather than using a greedy algorithm to launch reduce tasks like Fair scheduler [62]. Tan *et al.* [148] formulated the reduce task scheduling that minimizes the shuffle data transfer cost to a classic stochastic assignment problem to find out the optimal reduce task placement. Jiang *et al.* [96] designed Symbiosis, which identifies and corrects unbalanced utilization of multiple resources during runtime to improve the resource utilization such as computing and network resources. Purlieus [129] aims to improve locality of MapReduce in a cloud by carefully placing virtual machine and data. ShuffleWatcher [3] reduces the cross-rack congestion by delaying all the reduce tasks and tries to place the map tasks into one or a fewer racks. However, the above works cannot fully address the network bottlenecks caused by a large amount of shuffle-heavy jobs, since they either cannot reduce/avoid network congestion, or incur additional drawbacks such as increasing the cross-rack traffic by reading map input data. In addition, these works cannot efficiently use OCS in Hybrid-DCN to accelerate the data transfer to improve the job performance.

Recently, there have been plenty of studies [28, 62, 73, 74, 94, 96, 97, 172] focusing on designing cluster schedulers to improve the performance of the clusters (e.g., throughput and SLOs). On one hand, these works are not network-aware and hence they can neither handle the network bottlenecks caused by shuffle-heavy jobs nor efficiently exploit OCS in Hybrid-DCN. On the other hand, these studies are complementary to our design of hybrid scale-up/out cluster and can be combined with the Hybrid cluster for better performance. Specifically, we first utilize our work to place the jobs and data into different machines, and then the cluster schedulers in those previous studies can be further applied to allocate the resource among scale-up machines and scale-out machines efficiently.

The job scheduling problem in heterogeneous cluster has attracted much attention [4, 68, 105, 173]. They identify the causes of poor performance on heterogenous cluster and address the poor performance using techniques such as scheduling backup copies and estimating the job progress and prioritizing different jobs based on their progress. However, none of these proposals consider the diversity of jobs in their solutions. In this dissertation, we leverage the observation that different machines may result in different performance for different jobs and design a Hybrid cluster to accelerate big data analytics.

7.5 Job Schedulers in HPC, Grids and Multiprocessor Systems

In the world of high performance computing (HPC), Grid computing and multiprocessor systems, much research has been conducted on the job schedulers [1, 12, 21, 24, 25, 46, 52, 54–56, 59, 67, 70, 72, 88, 98, 103, 104, 107, 108, 115, 116, 131, 135, 138, 139, 146, 152, 156, 157, 161, 164, 171, 175]. Similar to the schedulers in data-parallel clusters, the schedulers also assign tasks to different machines/processors, considering different requirements such as resource requirements, placement constraints, and dependencies. The scheduling objectives are also very similar to the schedulers in data-parallel clusters, including minimizing makespan, minimizing mean completion time, maximizing throughput, fairness, or a combination of these. The papers [23, 65, 130] present good reviews on the topic.

The works in [21, 72, 101, 108] consider malleable task scheduling in multiprocessor systems, where each task can run on multiple servers. However, none of them address the issues of scheduling in the context of efficiently exploiting the network to achieve better performance.

In data-parallel clusters, tasks are independent to each other so that killing one of them will not impact others. On the contrast, in the programming model like MPI [126] and multiprocessor approaches such as coscheduling [127], tasks run concurrently and communicate during their execution. Such a key difference leads to new schedulers in data-parallel clusters, compared with the task scheduling in multiprocessor.

HPC schedulers [85, 92, 145, 177] schedule HPC jobs to run on a fixed number of machines which communicate through a mechanism like MPI. The allocations to the HPC jobs only change infrequently (e.g., when a node fails). This is because killing or moving a single process of MPI jobs typically requires the restart of all of the other processes, as these jobs consist of sets of stateful processes communicating across the network.

On the contrast, the jobs in data-parallel clusters are elastic to achieve different optimization goals. In data-parallel clusters, multiple jobs exploit statistical multiplexing to share the cluster and the nodes that can be assigned to a job are continuously changing with time. Thus, a node is in general assigned to different jobs once its current tasks complete, rather than giving any job a long-term private allocation. This is the primary reason that the previous scheduling techniques such as gang scheduling [64] are no longer applicable to the data-parallel clusters. Instead, the scheduler in data-parallel clusters needs to react dynamically to the conditions in the cluster (e.g., nodes, network and job characteristics).

In addition, HPC jobs are usually CPU bound or communication bound and the HPC schedulers often target environment that has specialized hardware, such as Infiniband and SANs. The data locality at the node level is often not considered in these schedulers, which, however, is a very crucial factor in the schedulers for data-parallel clusters.

The Grid schedulers like [149, 150] consider the locality constraints, but at the level of different geographic sites. The papers [22, 33] also considered to replicate data across different sites, which is similar to the replication in data-parallel cluster. The difference is that the schedulers in data-parallel clusters focus on the task placement in a local-area cluster and mainly consider the data locality at the level of node or rack locality.

Chapter 8

Conclusions

8.1 Summary of Dissertation

Although data-parallel frameworks were originally designed to process *data-intensive jobs* with large input datasets, many previous studies show that current production clusters process increasingly *diverse* jobs with different job characteristics (e.g., input data size, shuffle data size, and output data size). It is crucial to improve the performance the data-parallel clusters with diverse workloads.

In this dissertation, we mainly investigated the thesis statement:

- We can improve the performance of current state-of-the-art schedulers (e.g., Fair and Delay schedulers in Hadoop) by balancing the network traffic temporally and enforcing the data locality for shuffle data, aggregating the data transfers to efficiently exploit optical circuit switch in hybrid electrical/optical datacenter network while still guaranteeing parallelism of the jobs, and adaptively scheduling a job to either scale-up machines or scale-out machines that benefit the job the most in hybrid scale-up/out cluster.

Based on the results presented in the preceding chapters, we believe the work in this dissertation supports this thesis statement. We have presented job schedulers in different architectures to improve the performance of data-parallel clusters with diverse workloads. We claim the following contributions in this dissertation:

- In Chapter 3, we presented a new network-aware MapReduce scheduler (NAS). NAS consists of three mechanisms: Map Task Scheduling (MTS), Congestion-reduction Reduce Task Scheduling (CR-RTS) and Congestion-avoidance Reduce Task Scheduling (CA-RTS). These three mechanisms jointly work to balance the cross-rack network traffic temporally and reduce cross-rack network traffic. We implemented NAS in Hadoop on a supercomputing cluster. Through large-scale trace-driven simulation based on the Facebook workload and real Hadoop cluster experiment, we showed that NAS greatly improves cluster throughput and reduces average job completion time compared with the Fair, Delay and ShuffleWatcher schedulers.

- In Chapter 4, we presented a job scheduler JobPacker for data-parallel frameworks to meet the needs of modern advanced hybrid electrical/optical datacenter networks. JobPacker aggregates the data transfers of a job to use OCS effectively. Based on the predictable characteristics of recurring jobs, JobPacker has an offline scheduler to find out all feasible (map-width, reduce-width) pairs for every recurring job that can use OCS effectively while achieving sufficient parallelism, find out the best (map-width, reduce-width) pair with the shortest job completion time, and generate the global schedule including which racks and the sequence to run the recurring jobs that yields the best performance. The offline scheduler also has a new sorting method to prioritize the recurring jobs to prevent high resource contention while fully utilizing cluster resource. Based on the offline schedule, an online scheduler places input data and schedules the recurring jobs, and schedules non-recurring jobs to idle resources that are not assigned to recurring jobs. We evaluated JobPacker using large-scale simulation and small-scale emulation on GENI based on production workload, which demonstrates its higher performance in comparison with other schedulers.

- In Chapter 5, we illustrated the challenges in designing a hybrid scale-up/out cluster and presented corresponding job placement and data placement strategies to handle these challenges. In Hybrid architecture, we separate scale-up and scale-out machines to different racks. In the job placement strategy, we actively place the jobs to different machines based on their characteristics. In data placement strategy, we use replication placement technique to maintain high data locality. We evaluated Hybrid by running a production workload (FB-2010) with both real cluster run and trace-driven simulation. The results show that accompanying with our strategies, Hybrid cluster can reduce the makespan and the job completion time, compared to tradition scale-out cluster with state-of-the-art schedulers.

8.2 Future Work

This dissertation represents some of the first steps to support efficient data-parallel frameworks in modern advanced datacenter networks and cluster architecture. The work can be improved, enhanced or extended in many ways. We anticipate that, in the near future, more and more advanced datacenter architectures would appear and be deployed in practice. Future work might focus more on bridging the gaps between data-parallel frameworks and modern advanced datacenter architectures. Specifically, we list a number of potential future work here.

- With the more and more workloads in production, the jobs may have dependency among them, i.e., a job's output is the input of another job. We will extend our schedulers in this dissertation to consider the placement of dependent jobs to reduce the network traffic generated from input data reading among the dependent jobs.

- With the rapid development of optical technologies, we anticipate that optical switching will dominate the future advanced datacenter networks in the future. We will further identify the challenges for data-parallel frameworks to use the emerging optical switching technologies, and explore appropriate job schedulers to keep pace to meet the needs of the advanced datacenter networks.

- With the current trends that there are more and more jobs in production, we anticipate that the jobs would be more and more diverse. We plan to further explore a more complex Hybrid cluster with more kinds of machines to handle more diverse workloads.

Bibliography

[1] T. L. Adam, K. M. Chandy, and J. Dickson. “A comparison of list schedules for parallel processing systems”. In: *Communications of the ACM* 17.12 (1974), pp. 685–690.

[2] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. “Re-optimizing data-parallel computing”. In: *Proc. of USENIX ATC*. 2012.

[3] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. “ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters”. In: *Proc. of ATC*. 2014.

[4] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. “Tarazu: Optimizing MapReduce on Heterogeneous Clusters”. In: *Proc. of ASPLOS*. 2012, pp. 61–74.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. “A Scalable, Commodity Data Center Network Architecture”. In: *Proc. of SIGCOMM*. 2008, pp. 63–74.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. “Hedera: Dynamic Flow Scheduling for Data Center Networks”. In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI’10. 2010, pp. 19–19.

[7] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. “CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics”. In: *Proc. of NSDI.* 2017.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. “Data center tcp (dctcp)”. In: *ACM SIGCOMM computer communication review.* Vol. 40. 4. ACM. 2010, pp. 63–74.

[9] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. “pfabric: Minimal near-optimal datacenter transport”. In: *ACM SIGCOMM Computer Communication Review.* Vol. 43. 4. ACM. 2013, pp. 435–446.

[10] *Amazon S3.* https://aws.amazon.com/s3/. 2018.

[11] *Amazon web service.* http://aws.amazon.com/.

[12] P. R. Amestoy, A. Guermouche, J.-Y. L’Excellent, and S. Pralet. “Hybrid scheduling for the parallel solution of linear systems”. In: *Parallel computing* 32.2 (2006), pp. 136–156.

[13] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. “PACMan: Coordinated memory caching for parallel jobs”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation.* USENIX Association. 2012, pp. 20–20.

[14] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. “Reining in the Outliers in Map-reduce Clusters Using Mantri”. In: *Proc. of OSDI.* 2010, pp. 1–16.

[15] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. “Scarlett: coping with skewed content popularity in

mapreduce clusters". In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 287–300.

[16] *Apache Hadoop*. http://hadoop.apache.org/.

[17] *Apache Spark*. `https://spark.apache.org/`.

[18] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. "Scale-up vs Scale-out for Hadoop: Time to Rethink?" In: *Proc. of SOCC*. 2013.

[19] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. "Information-Agnostic Flow Scheduling for Commodity Data Centers." In: *NSDI*. 2015, pp. 455–468.

[20] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. "Towards predictable datacenter networks". In: *Proc. of the SIGCOMM*. Toronto, Ontario, Canada, 2011. ISBN: 978-1-4503-0797-0. DOI: `10.1145/2018436.2018465`. URL: `http://doi.acm.org/10.1145/2018436.2018465`.

[21] K. P. B. P. Banerjee. "An approximate algorithm for the partitionable independent task scheduling problem". In: *Urbana* 51 (1990), p. 61801.

[22] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. "Explicit Control in the Batch-Aware Distributed File System." In: *NSDI*. Vol. 4. 2004, pp. 365–378.

[23] P. Brucker and P Brucker. *Scheduling algorithms*. Vol. 3. Springer, 2007.

[24] R. Buyya and M. Murshed. "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing". In: *Concurrency and computation: practice and experience* 14.13-15 (2002), pp. 1175–1220.

[25] U. S. C. Ernemann V. Hamscher. “On Advantages of Grid Computing for Parallel Job Scheduling”. In: *Cluster Computing On the Grid* (2005).

[26] *Capacity Scheduler*. `http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html`.

[27] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. “SCOPE: easy and efficient parallel processing of massive data sets”. In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1265–1276.

[28] C. Chen, W. Wang, S. Zhang, and B. Li. “Cluster Fair Queueing: Speeding up Data-Parallel Jobs with Delay Guarantees”. In: *Proc. of INFOCOM*. 2017.

[29] F. Chen, M. Kodialam, and T. Lakshman. “Joint scheduling of processing and shuffle phases in mapreduce systems”. In: *Proc. of INFOCOM*. 2012.

[30] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen. “OSA: An optical switching architecture for data center networks with unprecedented flexibility”. In: *Proc. of NSDI*. 2012.

[31] Y. Chen, S. Alspaugh, and R. Katz. “Interactive Analytical Processing in Big Data Systems: A CrossIndustry Study of MapReduce Workloads.” In: *Proc. of VLDB*. 2012.

[32] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. “The Case for Evaluating MapReduce Performance Using Workload Suites.” In: *Proc. of MASCOTS*. 2011.

[33] A. Chervenak, E. Deelman, M. Livny, M.-H. Su, R. Schuler, S. Bharathi, G. Mehta, and K. Vahi. “Data placement for scientific applications in distributed environments”. In: *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*. IEEE Computer Society. 2007, pp. 267–274.

[34] M. Chowdhury, S. Kandula, and I. Stoica. “Leveraging endpoint flexibility in data-intensive clusters”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. ACM. 2013, pp. 231–242.

[35] M. Chowdhury and I. Stoica. “Coflow: A networking abstraction for cluster applications”. In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM. 2012, pp. 31–36.

[36] M. Chowdhury and I. Stoica. “Efficient coflow scheduling without prior knowledge”. In: *Proc. of SIGCOMM*. 2015.

[37] M. Chowdhury, Y. Zhong, and I. Stoica. “Efficient coflow scheduling with varys”. In: *Proc. of SIGCOMM*. 2014.

[38] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. “Managing data transfers in computer clusters with orchestra”. In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 98–109.

[39] *Clemson Palmetto HPC cluster*. http://citi.clemson.edu/palmetto/. 2015].

[40] C. Clos. “A Study of Non-Blocking Switching Networks”. In: *Bell Labs Technical Journal* 32.2 (1953), pp. 406–424.

[41] *CloudLab*. https://www.cloudlab.us/. 2018.

[42] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. “MapReduce Online”. In: *Proc. of NSDI*. 2010.

[43] C. Cortes and V. Vapnik. “Support-vector networks”. In: *Machine learning* 20.3 (1995), pp. 273–297.

[44] P. Costa, A. Donnelly, A. Rowstron, and G. O’Shea. “Camdoop: Exploiting In-network Aggregation for Big Data Applications”. In: *Proc. of NSDI*. 2012.

[45] *CPU-world*. `http://www.cpu-world.com/`.

[46] S. Darbha and D. P. Agrawal. “Optimal scheduling algorithm for distributed-memory machines”. In: *IEEE transactions on parallel and distributed systems* 9.1 (1998), pp. 87–95.

[47] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Proc. of OSDI*. 2004.

[48] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. “Hawk: Hybrid Datacenter Scheduling”. In: *Proc. of USENIX ATC*. 2015.

[49] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. “Hawk: hybrid datacenter scheduling”. In: *Proc. of ATC*. 2015, pp. 499–510.

[50] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. “Job-aware Scheduling in Eagle: Divide and Stick to Your Probes”. In: *Proc. of SOCC*. 2016.

[51] A. Demers, S. Keshav, and S. Shenker. “Analysis and simulation of a fair queueing algorithm”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 19. 4. ACM. 1989, pp. 1–12.

[52] V. Di Martino and M. Mililotti. “Sub optimal scheduling in a grid using genetic algorithms”. In: *Parallel computing* 30.5-6 (2004), pp. 553–565.

[53] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. “Decentralized task-aware scheduling for data center networks”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 431–442.

[54] L. R. Dror G. Feitelson. “Metrics and benchmarking for parallel job scheduling”. In: *Lecture Notes in Computer Science* (2006).

[55] L. R. Dror G. Feitelson. “Parallel job scheduling: Issues and approaches”. In: *Lecture Notes in Computer Science* (2005).

[56] U. S. K. C. S. P. W. Dror G. Feitelson Larry Rudolph. “Theory and practice in parallel job scheduling”. In: *Lecture Notes in Computer Science* (2005).

[57] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Citeseer, 2008.

[58] J. Edmonds. “Paths, trees, and flowers”. In: *Canadian Journal of mathematics* 17.3 (1965), pp. 449–467.

[59] H. El-Rewini and T. G. Lewis. “Scheduling parallel program tasks onto arbitrary target machines”. In: *Journal of parallel and Distributed Computing* 9.2 (1990), pp. 138–153.

[60] K. Elmeleegy. “Piranha: Optimizing Short Jobs in Hadoop”. In: *Proc. of VLDB Endow.* 2013.

[61] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. “CoHadoop: flexible data placement and its exploitation in Hadoop”. In: *Proceedings of the VLDB Endowment* 4.9 (2011), pp. 575–585.

[62] *Fair Scheduler*. `http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html`.

[63] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. “Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers”. In: *Proc. of SIGCOMM*. 2010.

[64] D. G. Feitelson and L. Rudolph. “Gang scheduling performance benefits for fine-grain synchronization”. In: *Journal of Parallel and distributed Computing* 16.4 (1992), pp. 306–318.

[65] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. “Theory and practice in parallel job scheduling”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1997, pp. 1–34.

[66] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. “Jockey: guaranteed job latency in data parallel clusters”. In: *Proc. of EuroSys*. 2012.

[67] I. Foster. *Designing and building parallel programs.* Vol. 78. Addison Wesley Publishing Company Boston, 1995.

[68] R. Gandhi, D. Xie, and Y. C. Hu. “PIKACHU: How to Rebalance Load in Optimizing Mapreduce on Heterogeneous Clusters”. In: *Proc. of ATC.* 2013, pp. 61–66.

[69] *GENI.* https://www.geni.net.

[70] A. R. P. S. Gerald Sabin Rajkumar Kettimuthu. “Scheduling of Parallel Jobs in a Heterogeneous Multi-site Environment”. In: *Lecture Notes in Computer Science* (2003).

[71] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. “Dominant Resource Fairness: Fair Allocation of Multiple Resource Types”. In: *Proc. of NSDI.* 2011.

[72] R. L. Graham. “Bounds on multiprocessing timing anomalies”. In: *SIAM journal on Applied Mathematics* 17.2 (1969), pp. 416–429.

[73] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. “Altruistic Scheduling in Multi-Resource Clusters”. In: *Proc. of OSDI.* 2016.

[74] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. “GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters”. In: *Proc. of OSDI.* 2016.

[75] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. “Multi-resource Packing for Cluster Schedulers”. In: *Proc. of SIGCOMM.* 2014.

[76] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. “Multi-resource packing for cluster schedulers”. In: *Proc. of SIGCOMM.* 2014.

[77] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. “VL2: a scalable and flexible data center network”. In: *Proc. of SIGCOMM*. 2009.

[78] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. “VL2: A Scalable and Flexible Data Center Network”. In: *Proc. of SIGCOMM*. 2009, pp. 51–62.

[79] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. “SecondNet: a data center network virtualization architecture with bandwidth guarantees.” In: *Proc. of CoNEXT*. 2010. ISBN: 978-1-4503-0448-1. URL: http://dblp.uni-trier.de/db/conf/conext/conext2010.html#GuoLWYKSWZ10.

[80] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. “BCube: a high performance, server-centric network architecture for modular data centers”. In: *ACM SIGCOMM Computer Communication Review* 39.4 (2009), pp. 63–74.

[81] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. “Dcell: a scalable and fault-tolerant network structure for data centers”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 38. 4. ACM. 2008, pp. 75–86.

[82] J. Guo, F. Liu, D. Zeng, J. Lui, and H. Jin. “A Cooperative Game Based Allocation for Sharing Data Center Networks”. In: *Proc. of INFOCOM*. 2013.

[83] Y. Guo, J. Rao, and X. Zhou. “iShuffle: Improving Hadoop Performance with Shuffle-on-Write”. In: *Proc. of ICAC*. 2013.

[84] *Hadoop*. http://hadoop.apache.org/. URL: http://hadoop.apache.org/.

[85] R. L. Henderson. “Job scheduling under the portable batch system”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1995, p-p. 279–294.

[86] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. “Mesos: A Platform for Fine-grained Resource Sharing in the Data Center”. In: *Proc. of NSDI*. 2011.

[87] C.-Y. Hong, M. Caesar, and P Godfrey. “Finishing flows quickly with preemptive scheduling”. In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM. 2012, pp. 127–138.

[88] E. S. Hou, N. Ansari, and H. Ren. “A genetic algorithm for multiprocessor scheduling”. In: *IEEE Transactions on Parallel and Distributed systems* 5.2 (1994), pp. 113–120.

[89] X. S. Huang, X. S. Sun, and T. Ng. “Sunflow: Efficient Optical Circuit Scheduling for Coflows”. In: *Proc. of CoNEXT*. 2016, pp. 297–311.

[90] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. “Dryad: distributed data-parallel programs from sequential building blocks”. In: *ACM SIGOPS operating systems review*. Vol. 41. 3. ACM. 2007, pp. 59–72.

[91] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. “Quincy: Fair Scheduling for Distributed Computing Clusters”. In: *Proc. of SOSP*. 2009, pp. 261–276.

[92] D. Jackson, Q. Snell, and M. Clement. “Core algorithms of the Maui scheduler”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2001, pp. 87–102.

[93] J. Jaffe. “Bottleneck flow control”. In: *IEEE Transactions on Communications* 29.7 (1981), pp. 954–962.

[94] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. “Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can”. In: *Proc. of SIGCOMM.* 2015, pp. 407–420.

[95] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. “EyeQ: Practical Network Performance Isolation at the Edge”. In: *Proc. of NSDI.* 2013.

[96] J. Jiang, S. Ma, B. Li, and B. Li. “Symbiosis: Network-Aware Task Scheduling in Data-Parallel Frameworks”. In: *Proc. of INFOCOM.* 2016.

[97] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. “Morpheus: Towards Automated SLOs for Enterprise Clusters”. In: *Proc. of OSDI.* 2016.

[98] H. Kasahara and S. Narita. “Practical multiprocessor scheduling algorithms for efficient parallel processing”. In: *IEEE Trans. Comput.;(United States)* 100 (1984).

[99] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. “An analysis of traces from a production MapReduce cluster”. In: *Proc. of CCGRID.* 2010.

[100] R. Kohavi et al. “A study of cross-validation and bootstrap for accuracy estimation and model selection”. In: *Proc. of IJCAI.* 1995.

[101] Y.-K. Kwok and I. Ahmad. “Static scheduling algorithms for allocating directed task graphs to multiprocessors”. In: *ACM Computing Surveys (CSUR)* 31.4 (1999), pp. 406–471.

[102] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. “Cicada: Introducing Predictive Guarantees for Cloud Networks”. In: *Proc. of HotCloud*. 2014.

[103] E. L. Lawler, J. K. Lenstra, A. H. R. Kan, and D. B. Shmoys. “Sequencing and scheduling: Algorithms and complexity”. In: *Handbooks in operations research and management science* 4 (1993), pp. 445–522.

[104] E. A. Lee and D. G. Messerschmitt. “Static scheduling of synchronous data flow programs for digital signal processing”. In: *IEEE Transactions on computers* 100.1 (1987), pp. 24–35.

[105] G. Lee, B.-G. Chun, and H. Katz. “Heterogeneity-aware Resource Allocation and Scheduling in the Cloud”. In: *Proc. of HotCloud*. 2011.

[106] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J. Kang, and P. Sharma. “Application-driven bandwidth guarantees in datacenters”. In: *Proc. of SIGCOMM*. 2014, pp. 467–478.

[107] J. K. Lenstra, D. B. Shmoys, and E. Tardos. “Approximation algorithms for scheduling unrelated parallel machines”. In: *Mathematical programming* 46.1-3 (1990), pp. 259–271.

[108] R. Lepere, D. Trystram, and G. J. Woeginger. “Approximation algorithms for scheduling malleable tasks under precedence constraints”. In: *International Journal of Foundations of Computer Science* 13.04 (2002), pp. 613–627.

[109] Z. Li, H. Shen, J. Denton, and W. Ligon. “Comparing Application Performance on HPC-based Hadoop Platforms with Local Storage and Dedicated Storage”. In: *Proc. of BigData*. 2016.

[110] Z. Li and H. Shen. “Designing A Hybrid Scale-Up/Out Hadoop Architecture Based on Performance Measurements for High Application Performance”. In: *Proc. of ICPP*. 2015.

[111] Z. Li and H. Shen. "Measuring Scale-Up and Scale-Out Hadoop with Remote and Local File Systems and Selecting the Best Platform". In: *IEEE Transactions on Parallel and Distributed Systems* 28.11 (2017), pp. 3201–3214.

[112] Z. Li and H. Shen. "Performance Measurement on Scale-up and Scale-out Hadoop with Remote and Local File Systems". In: *Proc. of Cloud*. 2016.

[113] H. Liu, M. K. Mukerjee, C. Li, N. Feltman, G. Papen, S. Savage, S. Seshan, G. M. Voelker, D. G. Andersen, M. Kaminsky, G. Porter, and A. C. Snoeren. "Scheduling techniques for hybrid circuit/packet networks". In: *Proc. of CoNext*. 2015.

[114] Y. Lu, G. Chen, L. Luo, K. Tan, Y. Xiong, X. Wang, and E. Chen. "One More Queue is Enough: Minimizing Flow Completion Time with Explicit Priority Notification". In: *Proc. of INFOCOM*. 2017.

[115] M. Maheswaran and H. J. Siegel. "A dynamic matching and scheduling algorithm for heterogeneous computing systems". In: *Heterogeneous Computing Workshop, 1998.(HCW 98) Proceedings. 1998 Seventh*. IEEE. 1998, pp. 57–69.

[116] G Manimaran and C. S. R. Murthy. "An efficient dynamic scheduling algorithm for multiprocessor real-time systems". In: *IEEE Transactions on Parallel and Distributed Systems* 9.3 (1998), pp. 312–319.

[117] *Microsoft Azure*. https://azure.microsoft.com/en-us/.

[118] *Microsoft Azure Storage*. https://azure.microsoft.com/en-us/services/storage/.

[119] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. "Towards characterizing cloud backend workloads: insights from Google compute clusters." In: *SIGMETRICS Performance Evaluation Review* 37.4 (2010), pp. 34–41.

[120] J. Mudigonda, P. Yalagandula, and J. C. Mogul. "Taming the Flying Cable Monster: A Topology Design and Optimization Framework for Data-Center Networks." In: *USENIX Annual Technical Conference*. 2011.

[121] A. Munir, T. He, R. Raghavendra, F. Le, and A. X. Liu. "Network Scheduling Aware Task Placement in Datacenters". In: *Proc. of CoNEXT*. 2016.

[122] V. Nagarajan, J. Wolf, A. Balmin, and K. Hildrum. "Flowflex: Malleable scheduling for flows of mapreduce jobs". In: *Proc. of Middleware*. 2013.

[123] *Newegg*. `http://www.newegg.com/`.

[124] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. "Portland: a scalable fault-tolerant layer 2 data center network fabric". In: *ACM SIGCOMM Computer Communication Review*. Vol. 39. 4. ACM. 2009, pp. 39–50.

[125] *ns-3 network simulator*. https://www.nsnam.org/1. 2018.

[126] *Open MPI*. http://www.open-mpi.org/. 2018.

[127] J. Ousterhout. "Scheduling techniques for concurrent systems". In: *Proceedings of the 3rd International Conference on Distributed Computing Systems*. 1982, pp. 22–30.

[128] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. "Sparrow: distributed, low latency scheduling". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 69–84.

[129] B. Palanisamy, A. Singh, L. Liu, and B. Jain. "Purlieus: locality-aware resource allocation for MapReduce in a cloud". In: *Proc. of SC*. 2011.

[130] M. L. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2016.

[131] C. D. Polychronopoulos and D. J. Kuck. “Guided self-scheduling: A practical scheduling scheme for parallel supercomputers”. In: *Ieee transactions on computers* 100.12 (1987), pp. 1425–1439.

[132] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. Santos. “Elasticswitch: practical work-conserving bandwidth guarantees for cloud computing”. In: *Proc. of SIGCOMM*. 2013.

[133] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. “FairCloud: sharing the network in cloud computing.” In: *Proc. of SIGCOMM*. 2012.

[134] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. “Integrating Microsecond Circuit Switching into the Data Center”. In: *Proc. of SIGCOMM*. 2013.

[135] S. Ranaweera and D. P. Agrawal. “A task duplication based scheduling algorithm for heterogeneous systems”. In: *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE. 2000, pp. 445–450.

[136] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou. “Workload characterization on a production Hadoop cluster: A case study on Taobao.” In: *Proc. of IISWC*. 2012.

[137] A. Rowstron, D. Narayanan, A. Donnelly, G. O’Shea, and A. Douglas. “Nobody ever got fired for using Hadoop on a cluster”. In: *Proc. of HotCDP*. 2012.

[138] V. S. S. Srinivasan R. Kettimuthu. “Characterization of backfilling strategies for parallel job scheduling”. In: *Parallel Processing Workshops* (2002).

[139] H. A. Schmidt, K. Strimmer, M. Vingron, and A. von Haeseler. "TREE-PUZZLE: maximum likelihood phylogenetic analysis using quartets and parallel computing". In: *Bioinformatics* 18.3 (2002), pp. 502–504.

[140] H. Shen and Z. Li. "New bandwidth sharing and pricing policies to achieve a win-win situation for cloud provider and tenants". In: *Proc. of INFOCOM*. 2014.

[141] H. Shen and Z. Li. "New bandwidth sharing and pricing policies to achieve a win-win situation for cloud provider and tenants". In: *IEEE Transactions on Parallel and Distributed Systems* 27.9 (2016), pp. 2682–2697.

[142] H. Shen, L. Yu, L. Chen, and Z. Li. "Goodbye to fixed bandwidth reservation: Job scheduling with elastic bandwidth reservation in clouds". In: *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–8.

[143] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. "Sharing the data center network". In: *Proc. of NSDI*. Boston, MA, 2011. URL: `http://dl.acm.org/citation.cfm?id=1972457.1972489`.

[144] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, et al. "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network". In: *ACM SIGCOMM computer communication review*. Vol. 45. 4. ACM. 2015, pp. 183–197.

[145] G. Staples. "Torque resource manager". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 8.

[146] E. D. Taillard. "Parallel taboo search techniques for the job shop scheduling problem". In: *ORSA journal on Computing* 6.2 (1994), pp. 108–117.

[147] J. Tan, X. Meng, and L. Zhang. “Coupling task progress for mapreduce resource-aware scheduling”. In: *Proc. of INFOCOM*. 2013.

[148] J. Tan, S. Meng, X. Meng, and L. Zhang. “Improving ReduceTask data locality for sequential MapReduce jobs”. In: *Proc. of INFOCOM*. 2013.

[149] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. “Condor: a distributed job scheduler”. In: *Beowulf cluster computing with Linux*. MIT press. 2001, pp. 307–350.

[150] D. Thain, T. Tannenbaum, and M. Livny. “Distributed computing in practice: the Condor experience”. In: *Concurrency and computation: practice and experience* 17.2-4 (2005), pp. 323–356.

[151] *The Network Simulator-ns-2*. http://www.isi.edu/nsnam/ns/. 2003.

[152] H. Topcuoglu, S. Hariri, and M.-y. Wu. “Performance-effective and low-complexity task scheduling for heterogeneous computing”. In: *IEEE transactions on parallel and distributed systems* 13.3 (2002), pp. 260–274.

[153] J. Turek, J. L. Wolf, and P. S. Yu. “Approximate algorithms scheduling parallelizable tasks”. In: *Proc. of SPAA*. 1992.

[154] *Typical Hadoop Cluster*. `https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.4.0/bk_cluster-planning-guide/content/typical-hadoop-cluster-hardware.1.html`.

[155] *Using Lustre with Apache Hadoop*. http://wiki.lustre.org/images/1/1b/Hadoop_wp_v0.4.2.pdf.

[156] R. Y. Uwe Schwiegelshohn. “Analysis of first-come-first-serve parallel job scheduling”. In: *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms* (1998).

[157] R. Y. Uwe Schwiegelshohn. “Fairness in parallel job scheduling”. In: *Journal of Scheduling* (2000).

[158] B. Vamanan, J. Hasan, and T. Vijaykumar. “Deadline-aware datacenter tcp (d2tcp)”. In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 115–126.

[159] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler. “Apache hadoop yarn: Yet another resource negotiator”. In: *Proc. of SOCC.* 2013.

[160] A. Verma, L. Cherkasova, and R. H. Campbell. “Resource provisioning framework for mapreduce jobs with performance goals”. In: *Proc. of Middleware.* 2011.

[161] A. S. R. Y. Volker Hamscher Uwe Schwiegelshohn. “Evaluation of Job-Scheduling Strategies for Grid Computing”. In: *Lecture Notes in Computer Science* (2002).

[162] G. Wang, D. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan. “c-Through: Part-time optics in data centers”. In: *Proc. of SIGCOMM.* 2010.

[163] G. Wang, T. Ng, and A. Shaikh. “Programming your network at run-time for big data applications”. In: *Proc. of HotSDN.* 2012.

[164] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. “Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach”. In: *Journal of parallel and distributed computing* 47.1 (1997), pp. 8–22.

[165] W. Wang, S. Ma, B. Li, and B. Li. “Coflex: Navigating the Fairness-Efficiency Tradeoff for Coflow Scheduling”. In: *Proc. of INFOCOM.* 2017.

[166] Y. Wang, C. Xu, X. Li, and W. Yu. “JVM-Bypass for Efficient Hadoop Shuffling”. In: *Proc. of IPDPS.* 2013.

[167] Y. Wang, J. Tan, W. Yu, L. Zhang, X. Meng, and X. Li. "Preemptive ReduceTask Scheduling for Fair and Fast Job Completion". In: *Proc. of ICAC*. 2013, pp. 279–289.

[168] D. Williams, S. Zheng, X. Zhang, and H. Jamjoom. "TideWatch: Fingerprinting the Cyclicality of Big Data Workloads". In: *Proc. of INFOCOM*. 2014.

[169] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. "Better never than late: Meeting deadlines in datacenter networks". In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 50–61.

[170] H. Wu, Z. Feng, C. Guo, and Y. Zhang. "ICTCP: Incast congestion control for TCP in data-center networks". In: *IEEE/ACM transactions on networking* 21.2 (2013), pp. 345–358.

[171] T. Yang and A. Gerasoulis. "DSC: Scheduling parallel tasks on an unbounded number of processors". In: *IEEE Transactions on Parallel and Distributed Systems* 5.9 (1994), pp. 951–967.

[172] M. Zaharia, D. Borthakur, S. Sen, K. Elmeleegy, S. Shenker, and I. Stoica. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling". In: *Proc. of EuroSys*. 2010.

[173] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. "Improving MapReduce Performance in Heterogeneous Environments". In: *Proc. of OSDI*. 2008, pp. 29–42.

[174] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. *Job scheduling for multi-user mapreduce clusters*. Tech. rep. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, 2009.

[175] L. Zhang, Y. Chen, R. Sun, S. Jing, and B. Yang. “A task scheduling algorithm based on PSO for grid computing”. In: *International Journal of Computational Intelligence Research* 4.1 (2008), pp. 37–43.

[176] Y. Zhao and J. Wu. “Dache: A data aware caching for big-data applications using the MapReduce framework”. In: *Proc. of INFOCOM.* 2013.

[177] S. Zhou. “Lsf: Load sharing in large heterogeneous distributed systems”. In: *I Workshop on Cluster Computing.* Vol. 136. 1992.