Stage Delay Scheduling: Speeding up DAG-style Data Analytics Jobs with Resource Interleaving

Wujie Shao*, Fei Xu*, Li Chen†, Haoyue Zheng*, Fangming Liu‡
*Department of Computer Science and Technology, East China Normal University.
†Department of Computer Science, University of Louisiana at Lafayette.
‡School of Computer Science and Technology, Huazhong University of Science and Technology.
*fxu@cs.ecnu.edu.cn, †li.chen@louisiana.edu, ‡fmliu@hust.edu.cn

ABSTRACT

To increase the resource utilization of datacenters, big data analytics jobs are commonly running stages in parallel which are organized into and scheduled according to the Directed Acyclic Graph (DAG). Through an in-depth analysis of the latest Alibaba cluster trace and our motivation experiments on Amazon EC2, however, we show that the CPU and network resources are still under-utilized due to the unwise stage scheduling, thereby prolonging the completion time of a DAG-style job (*e.g.,* Spark). While existing works on reducing the job completion time focus on either task scheduling or job scheduling, stage scheduling has received comparably little attention. In this paper, we design and implement *DelayStage*, a *simple yet effective* stage delay scheduling strategy to *interleave* the cluster resources across the parallel stages, so as to increase the cluster resource utilization and speed up the job performance. With the aim of minimizing the *makespan* of parallel stages, *DelayStage* judiciously arranges the execution of stages in a *pipelined* manner to maximize the performance benefits of resource interleaving. Extensive prototype experiments on 30 Amazon EC2 instances and complementary trace-driven simulations show that *DelayStage* can improve the cluster resource utilization by up to 81.8% and reduce the job completion time by up to 41.3%, in comparison to the stock Spark and the state-of-the-art stage scheduling strategies, yet with acceptable runtime overhead.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Theory of computation** → *Scheduling algorithms*; • **Computing methodologies** → *Parallel algorithms.*

KEYWORDS

stage delay scheduling, parallel stages, resource interleaving, job completion time, big data analytics

1 INTRODUCTION

Datacenters are hosting and processing an increasing amount of data, as reported by Forbes that 2.5 exabytes of data is daily generated and processed over the Internet [17]. Timely analysis of the massive amounts of data is mandated for decision making in modern companies like Facebook, Google, and Alibaba, where data analytics frameworks such as MapReduce [7], Spark [29], Flink [4], and Tensorflow [1] are employed to run big data analytics jobs. These jobs typically consist of a number of stages with inter-dependencies which are represented with Directed Acyclic Graphs (DAGs). Each stage is further divided into a number of tasks, reading input data over the network, processing data partitions using worker CPUs and writing output data to local disks [29].

It is typical for stages in a big data analytics job to be running in parallel. As evidenced by our motivational analysis (in Sec. 2.1) on the latest Alibaba cluster trace v2018 [2], *parallel stages* account for 79.1% of stages on average in production jobs. Unfortunately, the stages are naively scheduled to execute once they have completely acquired their input data. With such an unwise stage scheduling policy, parallel stages are easily executed in an almost "synchronized" manner — they fetch input data at the same time, competing for network bandwidth while leaving the CPU resource idle, and then processing data simultaneously, contending for CPU and leaving bandwidth idle instead. The significant fluctuation of resource utilization between the full and idle extremes inevitably results in an inefficient resource utilization. As demonstrated in our trace analysis, both the CPU and the network resource are under-utilized in production cluster, with around 20%-50% utilization on average. Undoubtedly, such heavy resource contention and inefficient resource usage would prolong the execution of DAG-style data analytics jobs.

To improve the cluster resource utilization and speed up the job performance, there have been a number of research works dedicated to task scheduling (*e.g.,* Quincy [15]), stage scheduling (*e.g.,* Tetris [8]), and job scheduling (*e.g.,* Flowtime [11]). These existing scheduling techniques above are mostly designed for optimizing the placement of (*i.e., where to place*) tasks or stages in both a single cluster (*e.g.,* Alibaba Fuxi [30], Graphene [10]) and across geo-distributed datacenters (*e.g.,* Geode [25], Tetrium [13]), as well as optimizing the job execution sequence (*e.g.,* LAS_MQ [12]). Nevertheless, comparatively little attention has been paid to optimizing the launch time of (*i.e., when to execute*) stages to improve job performance in the literature.

To fill this gap, we investigate from the perspective of manipulating the stage launch time, or more specifically, delaying the submission of parallel stages for a higher resource efficiency and a shorter job completion time. The intuition can be clearly illustrated with a motivation example of a Spark job, which consists of three parent stages and a child stage. With the default scheduling, the three parent stages will be launched at the same time, causing an intense resource contention. However, if we deliberately delay two of the parallel stages to some extents, the usage of bandwidth and CPU resources would be seamlessly interleaved, which eventually improves the job performance by accelerating its completion. Though it looks similar to Delay Scheduling [28], our idea is quite

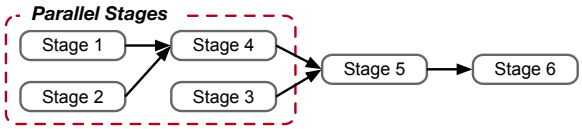


Figure 1: Illustration of parallel stages in the DAG of a representative big data analytics job (*i.e.,* the ALS job DAG with 6 stages). *Stage 1* runs in parallel with *Stage 2*, and *Stage 3* is executed in parallel with *Stage 1*, *Stage 2*, and *Stage 4*.

different for the following reasons. *First*, we delay parallel stages to accelerate job performance and improve cluster resource utilization, while Delay Scheduling delays the map task to achieve data locality and fairness. *Second*, we need to make decisions on which stage and how much time to delay, while Delay Scheduling does not. As a result, the deficiencies of existing works and the potential of great performance gains motivate us to develop a practical technique to optimize the launch time of parallel stages.

In this paper, we design *DelayStage*, a *simple yet effective* stage delay scheduling strategy to achieve acceleration of DAG-style data analytics jobs by *interleaving* the usage of resources (including CPU, network bandwidth and disk I/O) across multiple parallel stages in the cluster. In particular, *DelayStage* first organizes parallel stages into multiple execution paths in a descending order of path execution time according to the job's DAG. It then preferably schedules the stages in the *long-running* execution path, so as to reduce the resource fragments [10] and increase the cluster resource utilization. We implement a prototype of *DelayStage* based on Apache Spark [29] with two modules, including a *delay time calculator* and a *stage delayer*. With the aim of greedily minimizing the *makespan* of parallel stages, the calculator computes the best schedule for parallel stages, and then outputs the stage schedule time to the delayer, so that the execution of parallel stages is postponed accordingly in a lightweight manner.

We evaluate the effectiveness and runtime overhead of *DelayStage* with both extensive prototype experiments on 30 Amazon EC2 `m4.large` instances and large-scale simulations driven by Alibaba cluster trace v2018. Our experimental results show that *DelayStage* can improve the average utilization of CPU and network resources in the cluster by 7.2%-81.8%, and reduce the job completion time by 17.5%-41.3% compared to the stock Spark and the state-of-the-art stage scheduling strategies (*e.g.,* AggShuffle [16]). In addition, *DelayStage* incurs acceptable runtime overhead in practice.

The rest of the paper is organized as follows. Sec. 2 illustrates the inefficiency of cluster resource usage and our motivational example. Through modeling the makespan of parallel stages in Sec. 3, Sec. 4 designs and implements *DelayStage* based on Apache Spark to speed up the performance of a DAG-style job. Sec. 5 extensively evaluates the performance gains and runtime overhead of *DelayStage*. We discuss our contribution in the context of related work in Sec. 6. Finally, we conclude this paper in Sec. 7.

2 BACKGROUND AND MOTIVATION

In this section, we first characterize the execution of parallel stages in production jobs as well as the utilization of cluster resources from Alibaba cluster trace v2018. We then present an illustrative example to show how to speed up the job performance simply by delaying the scheduling time of parallel stages.

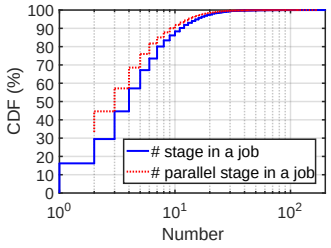


Figure 2: CDF of the number of stages and parallel stages executed in production jobs from Alibaba cluster trace.

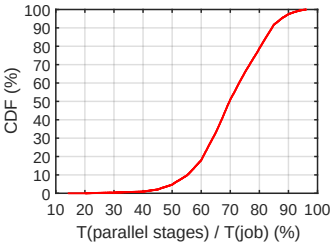


Figure 3: CDF of the proportion of the makespan of parallel stages to the job execution time in Alibaba trace.

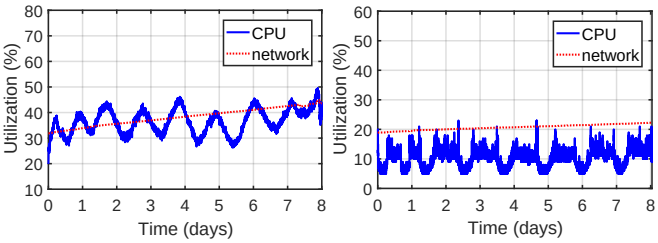


Figure 4: (a) Average CPU and network bandwidth utilization across 4,000 machines, and (b) CPU and network bandwidth utilization of a worker node (*i.e.,* `machine_id` = `m_2077`) over 8 days in Alibaba cluster trace.

2.1 Characterizing Parallel Stages and Resource Utilization

It is typical for a big data analytics (*e.g.,* Spark [29]) job to contain a number of pre-defined stages forming a DAG, with an example shown in Fig. 1 for an Alternating Least Squares (*i.e.,* ALS) job from Spark MLlib. These stages are executed according to the *dependency* in a DAG, as the *child* stage demands for the input data generated by its *parent* stages. In particular, a parent stage can be executed *in parallel* with another parent (or ancestor) stage, so as to improve the resource utilization of datacenters. Accordingly, we formally define *parallel stages* as the kind of stages which can be executed in parallel with *at least* one of the other stages in the job's DAG.

Parallel stages are common in production jobs. We analyze the latest Alibaba cluster trace v2018 [2], which includes 2,775,025 production jobs[1] running on 4,000 machines in a period of 8 days. Through a *topological sorting* of stages in the job's DAG, we find that over 68.6% of production jobs in Alibaba cluster trace have parallel stages. Similarly, we also observe that over 51% of jobs contain parallel stages in the job trace from Microsoft [10]. In more detail, we find that these jobs have 16,650,134 stages and 13,173,110 parallel stages in total, which indicates that the number of parallel stages accounts for over 79.1% of stages in a job on average. As evidenced in Fig. 2, the number of parallel stages is roughly close to the number of stages in a job.

Parallel stages' makespan dominates the job execution time. Through analyzing the start time and the end time of each stage in the Alibaba trace, we are able to obtain the execution time for all stages.

[1]We exclude the incomplete jobs before or after the 8-day trace time span

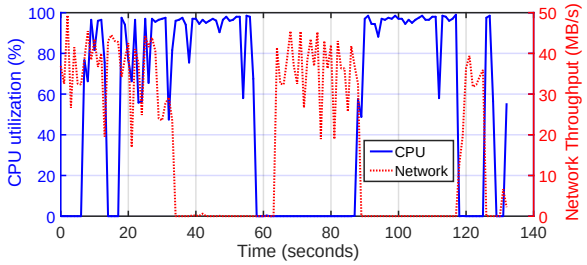


Figure 5: CPU utilization and network throughput of one worker node over time when running a ALS job with 3 GB input dataset in a three-node stock Spark cluster.

The stage runtime mostly spans from 10 seconds to 3,000 seconds in the trace. As shown in Fig. 3, the makespan (*i.e.,* execution time) of parallel stages accounts for more than 60% job completion time for over 80% of jobs. In particular, the average proportion of the makespan of parallel stages to the job execution time is 82.3%.

Unfortunately, *CPU and network resources are still under-utilized, despite the prevalence of parallel stages in production clusters.* As illustrated in Fig. 4(a), the average CPU utilization and network bandwidth utilization of 4,000 machines range from 20% to 50%, and 30% to 45%, respectively. Furthermore, we take a closer look at the resource utilization of machines during the execution of jobs. We find that the CPU utilization and the network bandwidth utilization fluctuate wildly from 0% to 98%, and from 0% to 62.2%, respectively. In particular, the CPU resource utilization of one worker node stays low-utilized (*i.e.,* below 10%) for around 39.1% of the job execution time as shown in Fig. 4(b). Accordingly, we infer that the *naive scheduling of stages leads to the inefficiency of resource usage during the job execution* in the cluster.

To confirm our hypothesis on the inefficient stage scheduling in the stock Spark, we examine the utilization of CPU and network resources of one worker node by launching a representative big data analytics job (*i.e.,* ALS) in a three-node cluster. As shown in Fig. 5, we observe that the CPU and network bandwidth resources of the worker node is either fully utilized or totally idle as expected. Specifically, the network bandwidth is idle during a period of 58 seconds (*i.e.,* from 32 to 62 seconds, and from 90 to 118 seconds). Similarly, the CPU resource is idle for around 38 seconds. The rationale is that, the *parallel stages are initially scheduled to execute at the same time* in the stock Spark, as long as the stage has acquired all of its input data and becomes ready for launch. As shown in Fig. 6 (a), the stock Spark executes *Stage 1*, *Stage 2*, and *Stage 3* simultaneously. Such a naive stage scheduling can cause the contention on the CPU and network bandwidth among the parallel stages, thereby making the resource utilization fluctuate significantly. As an example, *Stage 1* and *Stage 3* contend for the network resource during the period from 0 to 20 seconds, and they also contend for the CPU resource in the period from 32 to 60 seconds. As a result, the unwise scheduling of parallel stages is likely to cause the inefficiency of resource usage, thereby prolonging the job completion time.

2.2 An Illustrative Example

To improve the resource utilization of the cluster and speed up the job performance, we propose a *simple yet effective* approach

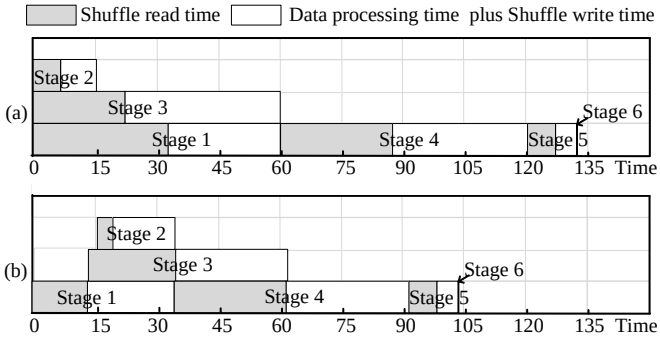


Figure 6: An illustrative motivation example with the ALS job: (a) the stock Spark naively launches the parallel stages at the same time, making them compete for the cluster resources; (b) If we simply postpone the scheduling of *Stage 2* and *Stage 3*, the utilization of the network bandwidth and computing resources can be improved by 31.3% and 40.1%, respectively, thereby reducing the job completion time from 133 seconds to 104 seconds. The gray and white parts of a stage denote the shuffle read time and the data processing time plus shuffle write time, respectively.

to *interleave* the cluster (*e.g.,* CPU, network bandwidth, disk I/O) resources across the stages by delaying the execution of parallel stages. Still with the ALS job as our motivation example illustrated in Fig. 6, we next present how our proposed approach manages to achieve efficient usage of cluster resources and thus the accelerated job completion time.

In general, each stage execution can be divided into three phases: shuffle reading the input data over the network, executing the stage by the CPU resource, and shuffle writing the output data to the disk [29]. As observed in Fig. 6 (a), there is intense network contention among *Stage 1*, *Stage 2*, and *Stage 3* during the first 10 seconds in stock Spark, while the CPU resource remains idle. Later, *Stage 1* and *Stage 3* contend for the CPU resource during the period of 32 to 60 seconds along the timeline, while the network resource is totally idle. Finally, the CPU resource becomes idle again for almost 30 seconds, when *Stage 4* starts to fetch (*i.e.,* shuffle read) the input data from *Stage 1* and *Stage 2*. Accordingly, the naive stage scheduling inevitably causes the resource wastage of the CPU and network bandwidth. In contrast, if we simply delay the execution of *Stage 2* and *Stage 3* as illustrated in Fig. 6 (b), the CPU and network resources are *interleaved* across *Stage 1*, *Stage 2*, and *Stage 3* during the period of 13 to 32 seconds, and across *Stage 3* and *Stage 4* during the period of 32 to 60 seconds. As a result, our stage delay scheduling can improve the average CPU utilization from 52.34% to 68.73%, and the average network throughput from 17.91 MB/s to 25.2 MB/s, thereby shortening the job completion time by 27.8%.

3 MODELING MAKESPAN OF PARALLEL STAGES FOR A DAG-STYLE JOB

In this section, we build an analytical model to formulate the makespan of parallel stages in a DAG-style job particularly with different stage scheduling plans (*i.e.,* deciding when to submit the

Table 1: Notations in our stage scheduling model

Notation	Definition
$\mathcal{K}$	Set of parallel stages in a DAG-style job
$\mathcal{P}_m$	Set of stages in an execution path m of a job's DAG
T_m	Execution time of an execution path m
$\mathcal{I}$	Set of nodes that store input data of a stage k
$\mathcal{W}$	Set of worker nodes that execute tasks of a stage k
t_k	Execution time of a stage k
t_k^w	Task execution time of a stage k on a worker node w
$s_k^{i,w}$	Size of shuffle input data that a stage k reads from a node i to a worker node w
$B_k^{i,w}$	Available bandwidth for a stage k over the network link from a node i to a worker node w ($i \neq w$)
d_k^w	Size of shuffle data to a worker node w of a stage k
D_k^w	Available disk bandwidth for a stage k on a node w
ϵ_k^w	Number of available executors for running a stage k on a worker node w
R_k	Data processing rate of a stage k on an executor
$B^{i,w}$	Available bandwidth over the network link from a node i to a worker node w ($i \neq w$)
D^w	Available disk bandwidth on a node w
ϵ^w	Number of available executors on a worker node w
$\mathcal{X}$	Set of delayed scheduling time of parallel stages

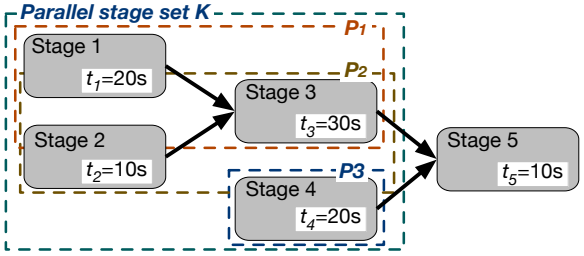


Figure 7: Illustration of execution paths in a DAG-style job. The parallel stage set $\mathcal{K}$ includes Stage 1, 2, 3, and 4, which can be organized into three execution paths. We put Stage 1 and 3 as an execution path $\mathcal{P}_1$, and Stage 2 and 3 as another execution path $\mathcal{P}_2$. The remaining Stage 4 is put as the last execution path $\mathcal{P}_3$. In particular, Stage 5 is not put into parallel stage set $\mathcal{K}$, as it is executed *sequentially* with other stages.

parallel stages in a job). The key notations used in our stage scheduling model are summarized in Table 1.

3.1 Parallel Stage Scheduling Problem for A DAG-Style Job

We consider a DAG-style (*e.g.*, Spark) job with a set of parallel stages which is denoted by $\mathcal{K} = \{k_1, k_2, ..., k_n\}$. The parallel stages are running simultaneously in the worker nodes and belong to different execution paths. As illustrated in Fig. 7, the set of parallel stages $\mathcal{K}$ includes a set of execution paths denoted by $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_m\}$, and each execution path $\mathcal{P}_m$ consists of a set of stages $\{k_i \in \mathcal{K}\}$. For each stage k, we denote a set of nodes that store the input data as

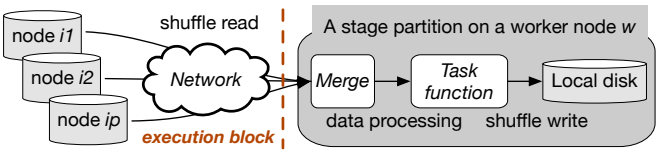


Figure 8: Detailed procedure of a task (*i.e.,* a stage partition) execution.

$\mathcal{I} = \{i_1, i_2, ..., i_p\}$, and a set of worker nodes $\mathcal{W} = \{w_1, w_2, ..., w_q\}$ that process the stage input data and shuffle write the intermediate data to their local disks.

To formulate the task execution time on a worker node w, we look into the detailed procedure of each stage partition as illustrated in Fig. 8. Specifically, it consists of three parts: (shuffle) reading the input data, executing the task, and shuffle writing the intermediate data to the local disk. In more detail, each stage partition (*i.e.,* a single task) first shuffle reads the input data from a set of nodes $\mathcal{I}$. The shuffle input data is then merged and processed on a worker node w, and generates the *intermediate data* which is finally shuffle written on the local disk of the worker node w. In particular, the process of merging data to the task function and shuffle write is *blocked* by the shuffle read process. This reason is that only after each task (*i.e.,* the stage partition) has acquired the whole of its input data, can such a stage partition start the data processing in the task [29]. Accordingly, the task execution time t_k^w on a worker node w can be formulated as

$$t_k^w = \max_{i \in \mathcal{I}} \left(\frac{s_k^{i,w}}{B_k^{i,w}} \right) + \frac{\sum_{i \in \mathcal{I}} s_k^{i,w}}{\epsilon_k^w \cdot R_k} + \frac{d_k^w}{D_k^w}, \tag{1}$$

where for each stage k, $s_k^{i,w}$ denotes the shuffle input data transferred from a node i to a worker w, and d_k^w denotes the shuffle write data to the disk of a worker w. $B_k^{i,w}$ and D_k^w denote the available network bandwidth from a node i to a worker w and the available disk bandwidth on a worker w. R_k is the stage data processing rate on an executor, and ϵ_k^w is the available number of executors on a worker w.

In more detail, the first item of Eq. (1) denotes the network transfer time of shuffle input data of stage k on worker node w, which is determined by the longest transmission time among all the input data partitions in upstream workers. The second item and the third item are the task (*i.e.,* a stage partition) processing time and the shuffle write time on worker node w, respectively. For simplicity, we assume the computing executor resource (*i.e.,* ϵ^w) and the bandwidth (*i.e.,* network, disk) resource ($B^{i,w}$) are *equally* shared among the parallel stages. Accordingly, the stage execution time is determined by the slowest worker $w \in \mathcal{W}$ [29]. With the execution time of stage partition defined in Eq. (1), we are able to formulate the stage execution time t_k as

$$t_k = \max_{w \in \mathcal{W}} \left(t_k^w \right). \tag{2}$$

Furthermore, we formulate the execution time T_m as the sum of the execution time of stages along an execution path m, which is given by

$$T_m = \sum_{k \in \mathcal{P}_m} (x_k + t_k), \tag{3}$$

where x_k denotes the *delayed scheduling* (*i.e.,* submission) time for a stage k, as opposed to the immediate submission with the stock stage scheduling in Spark. We proceed to define our stage scheduling problem to minimize the *makespan* of parallel stages for a DAG-style job as below,

$$\min_{\mathcal{X}} \quad \max_{\mathcal{P}_m \in \mathcal{P}} \left(T_m\right) \tag{4}$$

$$\text{s.t.} \quad x_k \geq 0, \quad \forall x_k \in \mathcal{X} \tag{5}$$

$$x_k \geq x_j + t_j, \quad \forall (x_j, x_k) \in \mathcal{Q} \tag{6}$$

$$\mathcal{Q} = \{(x_1, x_2) | H_m(x_1) < H_m(x_2), \mathcal{P}_m \in \mathcal{P}\} \tag{7}$$

The model variable is denoted by $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, which represents the set of delayed scheduling time for each parallel stage in the set $\mathcal{K}$ for a DAG-style job. In Constraint (7), $H_m(x_k)$ denotes the sequence index of stage x_k in the execution path $\mathcal{P}_m$, of which the value can be $1, 2, \cdots, |\mathcal{P}_m|$(*i.e.,* the total number of sequential stages in $\mathcal{P}_m$). The set $\mathcal{Q}$ represents the whole set of pairs of stages which have sequential dependency constraints. As intuitively indicated by the definition of $\mathcal{Q}$, for a stage pair $(x_j, x_k) \in \mathcal{Q}$, stage x_j has a smaller sequence index than x_k in an execution path. As such, stage x_k should be executed after the completion of x_j, as represented in Constraint (6).

3.2 Analysis of the Hidden Complexity

Although the form of the optimization seems neat and simple, the complexity is hidden in the representation of t_k, which is involved in both the objective and Constraint (6). In particular, t_k is determined by t_k^w, which further relies on $B_k^{i,w}$, ϵ_k^w, and D_k^w, as shown in Eq. (1). These variables are time-dependent, and are determined by the number of parallel stages (*i.e.,* how many stages are executed simultaneously), denoted by $f_\tau^w(\mathcal{X})$, at a particular worker w at a given time τ. Intuitively, the numbers depend on both the submission time and the execution process of tasks that are sharing the resources, which are non-trivial to calculate even in a numerical way when the submission time of all the stages in $\mathcal{X}$ are given. Thus, it is even harder to represent $f_\tau^w(\mathcal{X})$ in a closed-form expression of $\mathcal{X}$. If $f_\tau^w(\mathcal{X})$ is non-convex, problem defined in Eq. (4) becomes a non-convex optimization problem [5], which is at least NP-hard. Even if $f_\tau^w(\mathcal{X})$ is convex, the closed-form expression is difficult to obtain due to the inter-dependency between $f_\tau^w(\mathcal{X})$ and t_k^w. Rather than spending enormous efforts on unveiling the complex relationship which may turn out to be in vain, we propose our algorithmic solution to be presented in the next section.

4 DESIGN AND IMPLEMENTATION OF STAGE DELAY SCHEDULING

Based on our analytical performance model and problem analysis in the previous section, we proceed to design *DelayStage*, a *simple yet effective* stage scheduling strategy to solve our optimization problem. With the adjusted scheduling time (*i.e.,* $\mathcal{X}$) of parallel stages, *DelayStage* is able to increase the cluster resource utilization and reduce the job completion time. In addition, we unveil the implementation details of our *DelayStage* scheduler on Apache Spark [29].

4.1 Algorithm Design

To particularly answer "*which stage* and *how much time* should we delay during the job execution," our *DelayStage* strategy in Alg. 1 is quite *simple and intuitive*: we preferably schedule the stages in the *long-running* execution path with high priority, in order to minimize the resource fragments [10] and maximize the benefits of resource interleaving. Within an execution path $\mathcal{P}_m$, we further calculate the best scheduling time for each stage according to the stage dependency sequence, with the aim of greedily minimizing the makespan of parallel stages.

Algorithm 1: *DelayStage*: Stage delay scheduling strategy for minimizing job completion time and improving cluster resource utilization.

Input: The job's DAG; available network bandwidth $B^{i,w}$, disk bandwidth D^w and executor resources ϵ^w ($\forall w \in \mathcal{W}$); data processing rate R_k, the size of shuffle input data $s_k^{i,w}$ and shuffle write data d_k^w of stage k ($\forall k \in \mathcal{K}$).

Output: Delayed scheduling (*i.e.,* submission) time set $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ for parallel stages in a job.

1: **Initialize:** $\mathcal{X} \leftarrow \{-1, \ldots, -1\}$ and the set of execution paths $\mathcal{P}$ according to the job's DAG;
2: Calculate the initial stage execution time $\hat{t_k} \leftarrow$ Eq. (2) ($\forall k \in \mathcal{K}$), as if stage k is *running alone* in the cluster;
3: Calculate the execution time $T_m \leftarrow \sum_{k \in \mathcal{P}_m} \hat{t_k}$ on each path, and **initialize** the *makespan* of parallel stages $T_{max} \leftarrow \max_{\mathcal{P}_m \in \mathcal{P}} (T_m)$;
4: Sort $\mathcal{P}_m$ in $\mathcal{P}$ according to the *descending* order of T_m;
5: **for all** $\mathcal{P}_m$ in $\mathcal{P}$ **do**
6: **for all** k in $\mathcal{P}_m$ **do**
7: **if** $x_k \neq -1$ **then**
8: **continue**;
9: **end if**
10: Obtain the lower limit l_k and the upper limit u_k of x_k: $l_k \leftarrow$ (stage k has a parent stage j) ? $(x_j + t_j) : 0$; $u_k \leftarrow T_{max}$; $T_{max} \leftarrow \infty$;
11: **for all** $\hat{x_k} \in [l_k, u_k]$ **do**
12: Obtain the parameters D_k^w, $B_k^{i,w}$, and ϵ_k^w according to the parallelism of stage k;
13: Calculate the stage execution time $t_k \leftarrow$ Eq. (2);
14: Update the completion time of the subsequent stages in $\mathcal{P}_m$ and that of the scheduled stages interfering with the stage k;
15: Calculate the path execution time $T_m \leftarrow$ Eq. (3);
16: **if** $T_{max} > \max_{k \in \mathcal{P}_m} (T_m)$ **then**
17: $T_{max} \leftarrow \max_{k \in \mathcal{P}_m} (T_m)$, $x_k \leftarrow \hat{x_k}$;
18: **end if**
19: **end for**
20: **end for**
21: **end for**

Specifically, we first obtain the execution path set $\mathcal{P}$ by a *topological sorting* of the job's DAG, and initialize the stage execution time $\hat{t_k}, \forall k \in \mathcal{K}$, the path execution time $T_m, \forall \mathcal{P}_m \in \mathcal{P}$ and the makespan T_{max} of parallel stages (lines 1-3). With the sorted $\mathcal{P}_m$ according to T_m in a descending order, we then iterate over stages one by one in all the execution paths $\mathcal{P}_m$ (lines 4-6). In particular, we skip the scheduled stages in $\mathcal{P}_m$ as it has already been processed in a former execution path (lines 7-9). For each stage k, *DelayStage*

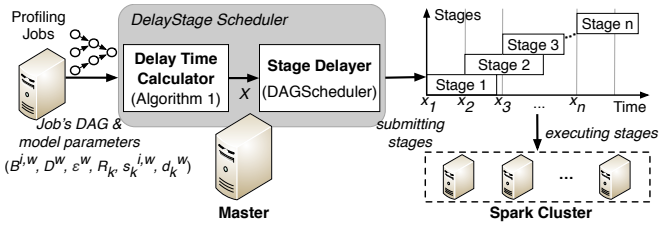


Figure 9: *DelayStage* implementation on Apache Spark.

further obtains the lower and upper bounds of the delayed scheduling time x_k (line 10) according to the stage dependency in $\mathcal{P}_m$. We set T_{max} as an infinite value as we need to store a minimum value of T_{max} in the following loop defined in lines 11-18. Assuming the time is slotted (*e.g.,* one second per slot), *DelayStage* iterates over x_k in the range of its lower and upper limits (line 11). With a candidate stage scheduling time x_k in each iteration, we first obtain the model parameters D_k^w, $B_k^{i,w}$, and ϵ_k^w, based on which we calculate the stage execution time t_k, the completion time of subsequent stages in $\mathcal{P}_m$ and update the completion time of the previously scheduled stages (lines 12-14). After calculating the path execution time T_m, *DelayStage* finally identifies the best schedule of delayed stage submission time x_k to greedily minimize the makespan T_{max} of parallel stages (lines 15-18).

Remark: The complexity of Alg. 1 is in the order of $O(|\mathcal{K}| \cdot m)$, where $m = \max_{k \in \mathcal{K}}\{u_k - l_k\}$ denotes the cardinality of the scheduling time space for x_k. As evidenced by Fig. 2, around 90% of production jobs have less than 15 parallel stages. Accordingly, the complexity of Alg. 1 can be reduced to $O(m)$ as $|\mathcal{K}|$ is limited. The computation time of our *DelayStage* strategy is well contained, which will be validated in Sec. 5.3. In addition, we intuitively sort the sequence of execution paths $\mathcal{P}_m$ in the *descending* order of T_m (line 4) in Alg. 1, so as to greedily derive the performance benefit from the resource interleaving across the stages in the long-running execution path. *DelayStage* can also work with the *random* or *ascending* sequence of $\mathcal{P}_m$ according to the path execution time T_m. We will examine the performance gains of *DelayStage* with the three kinds of $\mathcal{P}_m$ sequence, and validate the effectiveness of our stage delay scheduling strategy in Sec. 5.3.

4.2 Implementation of *DelayStage* on Apache Spark

We implement a prototype of our *DelayStage* scheduler based on Apache Spark [29], as Spark is a representative DAG-style data processing framework. Specifically, our prototype of *DelayStage* is built upon Apache Spark v2.3.1 with over 600 lines of C++ and Scala codes which are publicly available on GitHub[2]. As shown in Fig. 9, our *DelayStage* scheduler is deployed on the master of Spark, which includes two modules elaborated as follows.

Delay Time Calculator: The delay time calculator is actually the implementation of our stage delay scheduling strategy in Alg. 1. The input of our calculator can be obtained by profiling jobs on a worker node. Specifically, we sample the input data (*e.g.,* 10%) and profile the job on a single executor according to our prior work *iSpot* [27]. Through analyzing the log information (*e.g.,* `eventlog` in Spark) generated by the job profiling, we extract the job's DAG information including parallel stage set $\mathcal{K}$ and the execution path set $\mathcal{P}$, as well as the data processing rate R_k, the size of shuffle input data $s_k^{i,w}$ and shuffle write data d_k^w of each stage k. In addition, the available network bandwidth and disk I/O bandwidth of worker nodes can be periodically measured using the `netperf` and `iotop` command line tools. With the job's DAG information and the model parameters including $B^{i,w}$, D^w, ϵ^w, R_k, $s_k^{i,w}$, and d_k^w, our delay time calculator is able to output the delayed scheduling time of parallel stages $\mathcal{X} = \{x_1, x_2, ..., x_n\}$ by Alg. 1. In particular, we store the result $\mathcal{X}$ in the default configuration file `metrics.properties` in Apache Spark.

Stage Delayer: Based on the output result $\mathcal{X}$, the stage delayer first extracts the delayed scheduling time for parallel stages in `metrics.properties` and then postpones the submission time of these stages accordingly. In more detail, we add a new API function `stageDelayScheduling()` to simply `sleep` the submission of a stage k for a period of x_k. We invoke `stageDelayScheduling()` in the function `submitStage()` defined in `DAGScheduler.scala`, so as to delay the submission and execution of parallel stages in Spark. Accordingly, our stage delayer is simple and lightweight, without affecting the main scheduling functions like task placement or data placement in the cluster.

[2]https://github.com/icloud-ecnu/delaystage.

Table 2: Specifications of four representative benchmark workloads in our experiment.

Workload	Specification
ConnectedComponents	10 GB synthetic input data
CosineSimilarity	30 GB synthetic input data
LDA	140 million documents from Wikipedia articles [26] trained for 10 iterations
TriangleCount	10 million users and 100 million connections of synthetic input data

5 PERFORMANCE EVALUATION

In this section, we evaluate the effectiveness and runtime overhead of our *DelayStage* strategy by carrying out a set of real-world experiments with four representative DAG-style data analytics workloads on Amazon Elastic Compute Cloud (EC2) [3]. Furthermore, we conducted extensive simulations driven by the latest Alibaba cluster trace v2018 [2] to obtain complementary insights.

5.1 Experimental Setup

Cluster configurations: We set up a Spark cluster on Amazon EC2 [3] using 30 `m4.large` instances, each equipped with 2 Intel Xeon CPU E5-2670 v2 vCPUs, 8 GB memory, 32 GB SSD (Solid-State Drive) and the network bandwidth ranging from 100 Mbps to 480 Mbps. On each instance, we launch two executors, each configured with 1 vCPU and 2 GB memory accordingly. For simplicity, we configure the parameters in Spark as the default values and set up the HDFS cluster storage with 3 dedicated instances.

Workloads and datasets: We select four representative benchmark workloads of Spark, which include ConnectedComponents

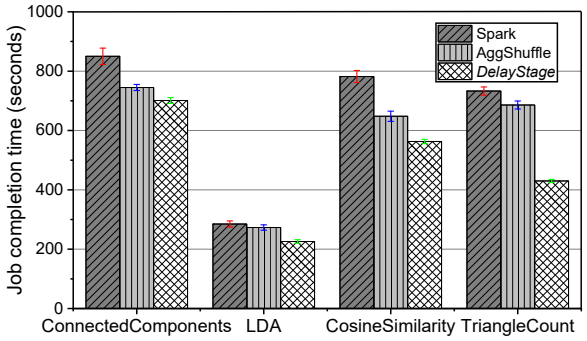


Figure 10: Job completion time of four representative workloads with stage scheduling strategies of the stock Spark, AggShuffle, and *DelayStage*.

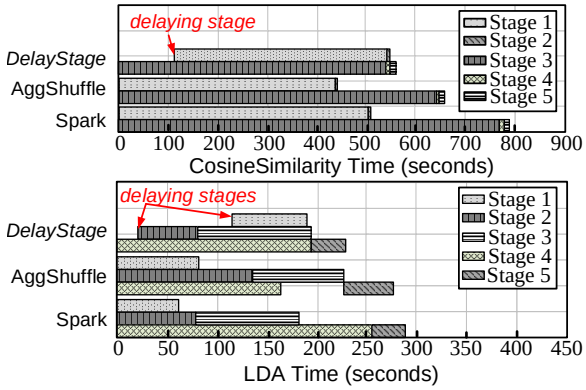


Figure 11: Stage execution time breakdown for CosineSimilarity with 5 stages and LDA with 5 stages. Each rectangular block represents the execution time of one stage.

with 5 stages and TriangleCount with 11 stages from Spark GraphX, CosineSimilarity with 5 stages and LDA (*i.e.,* Latent Dirichlet Allocation) with 5 stages from Spark MLlib. Specifically, we run ConnectedComponents and CosineSimilarity applications with 10 GB and 30 GB synthetic data, respectively, LDA with 140 million Wikipedia documents and TriangleCount applications with 100 million connections of synthetic data. We summarize the workload specifications above in Table 2.

Baselines: We compare our *DelayStage* strategy with two comparable baselines: *1)* the naive stage scheduling in the stock Spark, which immediately submits stages once they have acquired all the shuffle input data; *2)* AggShuffle [16], a state-of-the-art scheduling strategy which proactively transfers data to the child stage once the computation is completed, so as to pipeline the data transfer over the network and reduce the job completion time. We also compare *DelayStage* (the descending order of $\mathcal{P}_m$ by default) with its variations based on the random order and the ascending order of $\mathcal{P}_m$, as discussed in Sec. 4.1), in our trace-driven simulation.

Metrics: We illustrate the effectiveness of *DelayStage* mainly using three important metrics: job completion time, stage execution time and resource utilization of worker nodes. To make our performance evaluation accurate, we run our experiment five times for each workload, and illustrate the job performance with error bars of standard deviations.

5.2 Effectiveness of *DelayStage*

Job completion time: Fig. 10 compares the completion time of four representative big data analytics jobs under the stock Spark, AggShuffle, and *DelayStage*. Specifically, we observe that *DelayStage* shortens the job completion time by 17.5%-41.3% as compared to the stock *Spark*. The rationale is that our *DelayStage* strategy adjusts the scheduling time of parallel stages to greedily achieve resource interleaving across stages, thereby reducing the makespan of parallel stages and the job completion time. Meanwhile, *DelayStage* outperforms AggShuffle by a moderate performance improvement (*i.e.,* 4.2%-17.4%) because of two reasons: *First,* our *DelayStage* strategy seeks to improve the utilization of multiple resources including CPU, network, and disk, while AggShuffle only optimizes the usage of network bandwidth. Through tracing the CPU utilization of a worker node, we find that there exists a moderate amount of CPU idle periods with AggShuffle. *Second,* AggShuffle relies on the variance of task execution time within a stage, and thus the job performance improvement of AggShuffle becomes trivial when the stage tasks (*e.g.,* LDA) has nearly *homogeneous* stage partitions.

Furthermore, we observe that *DelayStage* achieves the most *stable* job performance (with the least execution time variance), as compared to the other two baseline strategies. This is because *DelayStage* tries to mitigate the resource contention across multiple parallel stages, by delaying the scheduling time of stages with more performance gains, so as to minimize the job completion time. In particular, we observe that the ConnectedComponents workload achieves the least performance improvement (*i.e.,* 17.5%) with *DelayStage*. This is because the execution time of sequential stages accounts for over half (*i.e.,* around 54.8%) of the job completion time for ConnectedComponents, leaving a limited optimization space for minimizing the job completion time.

Stage execution time: We breakdown the execution of two workloads (*i.e.,* CosineSimilarity, LDA) by stages, and the stage execution of ConnectedComponents and TriangleCount are shown in the Appendix A.1 of our technical report [23] due to the page limit. Specifically, we show the execution of parallel stages in the form of execution paths (defined in Fig. 7), which can be represented by multiple rectangular blocks executed in parallel. For example, the three execution paths in LDA are {*Stage 1*}, {*Stage 2, Stage 3*}, and {*Stage 4*}, and the execution of the last *Stage 5* is *blocked* by the parallel stages in LDA execution paths above, as shown in Fig. 11. Note that the job completion time in Fig. 10 is actually the sum of the makespan of parallel stages (*i.e.,* the longest execution time of execution paths) and the execution time of sequential stages (*e.g., Stage 5* of LDA). Leveraging the predicted stage execution time t_k by our Spark performance model within an error of 9.1% (as illustrated in the Appendix A.2 of our technical report [23]), *DelayStage* is able to delay the appropriate stages to improve resource utilization and speed up the job execution.

Specifically, we observe from Fig. 11 that, the stock Spark starts to submit and execute stages once the stage is ready for launch. Accordingly, the resource contention significantly prolongs the execution of the long-running execution path by 29.4% for CosineSimilarity and 23.8% for LDA, respectively. In contrast, *DelayStage* improves the job performance simply by delaying the execution

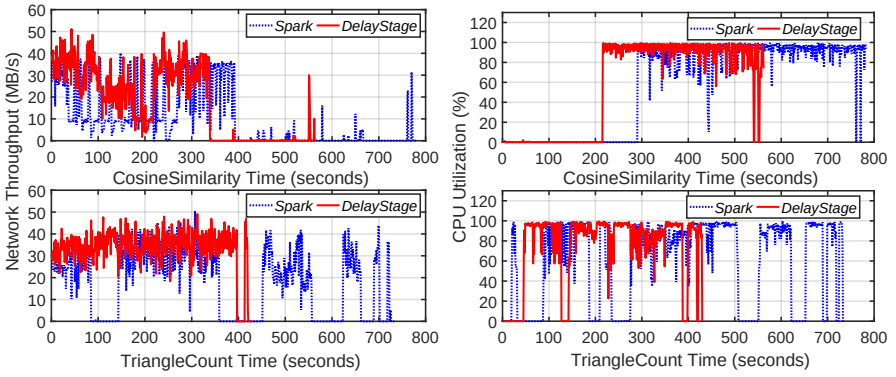


Figure 12: (a) Network throughput and (b) CPU utilization of a worker node running CosineSimilarity and TriangleCount under the stock Spark and *DelayStage* during the job execution.

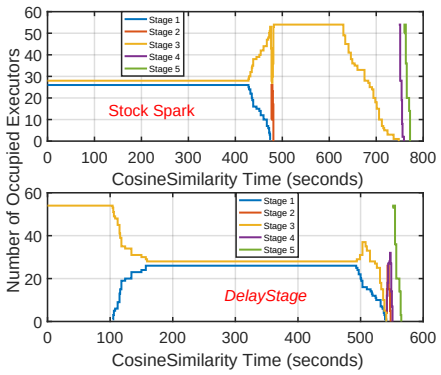


Figure 13: Comparison of the executor occupation across the stages of CosineSimilarity achieved by the stock Spark and *DelayStage*.

of several parallel stages (*i.e.,* *Stage 1* and *Stage 2* in both CosineSimilarity and LDA). The "magic" of our *DelayStage* strategy is that simply delaying the stages can achieve resource interleaving of CPU, network bandwidth and disk I/O across parallel stages, so that the long-running execution path (*e.g.,* {*Stage 3, Stage 4*} in CosineSimilarity, {*Stage 4*} in LDA) can be processed faster than the stock Spark.

Though AggShuffle improves the data locality on the reduce stage by proactively transferring the data during the shuffle phase, the performance benefits of AggShuffle actually depend on the heterogeneity of map tasks as discussed before. As illustrated in Fig. 11, AggShuffle can *possibly* increase the execution time of stages (*e.g.,* *Stage 1* and *Stage 2* in LDA), because the *Stage 1* and *Stage 2* of LDA contains almost the same tasks. In addition, the size of shuffle input data on the reduce stage is larger than the amount of the intermediate data before the shuffle phase. In more detail, the ratio of the shuffle input data size to the intermediate data size of *Stage 1* is large than 1 (*i.e.,* 1.3) in LDA, which requires more CPU computation works and thus prolongs the execution time of *Stage 1* accordingly.

CPU and network resource utilization: To examine whether our *DelayStage* strategy can improve the resource utilization, we further take a closer look at the CPU utilization and network throughput of a worker node during the execution of CosineSimilarity and TriangleCount workloads. Due to the page limit, the resource utilization of the other two workloads (*i.e.,* ConnectedComponents, LDA) are provided in the Appendix A.3 of our technical report [23].

As shown in Fig. 12(a), the *DelayStage* strategy fills most of the idle periods during the job execution, and thus achieves higher network throughput of a worker node as compared to the stock Spark. For example, *DelayStage* fills the network low-utilized period from around 40 to 160 seconds when running CosineSimilarity in the stock Spark shown in Fig. 12(a). This is because we simply delay the execution of *Stage 1* of CosineSimilarity for about 110 seconds during the shuffle reading process of *Stage 3*, so as to alleviate the network contention across the two stages, thereby increasing the utilization of network bandwidth. Similar observations can be obtained with *DelayStage* when running TriangleCount with 11 stages. By delaying *Stage 1* and *Stage 3* for 43 seconds and 107

Table 3: The #average (#standard deviation) of network throughput and CPU utilization of a worker node when running CosineSimilarity (Cos.), ConnectedComponents (Con.), LDA, and TriangleCount (Tri.) under the stock Spark and the *DelayStage* strategy.

	Network (MB/s)		CPU (%)	
	Spark	*DelayStage*	Spark	*DelayStage*
Cos.	10.2 (13.2)	17.1 (9.1)	56.6 (44.7)	60.7 (42.4)
Con.	11.5 (17.1)	13.6 (14.6)	49.5 (43.1)	53.4 (37.6)
LDA	21.3 (10.4)	26.7 (6.9)	46.6 (34.7)	59.7 (28.8)
Tri.	18.1 (15.1)	32.9 (9.9)	60.4 (42.5)	74.1 (35.8)

seconds, respectively, as illustrated in Appendix A.1 of our technical report [23], *DelayStage* can fully utilize the network bandwidth resource during the network idle period from 85 to 143 seconds running TriangleCount in the stock Spark.

In addition, the CPU resource of a worker node under the *DelayStage* strategy is better utilized compared with the stock Spark shown in Fig. 12(b). As an example, the CPU idle period from around 216 to 291 seconds running CosineSimilarity in the stock Spark can be totally filled by *DelayStage*. This is because the shuffle input data of *Stage 3* can be fetched faster than the stock Spark as we have delayed the submission of *Stage 1*, and thus the computation of *Stage 3* can be brought forward for around 75 seconds. To verify that, we further look into the execution occupation by stages of CosineSimilarity. As shown in Fig. 13, we observe that *Stage 3* can fully utilize the executor bandwidth resources to read its stage input data during the period of the submission delay of *Stage 1* (*i.e.,* from 0 to around 110 seconds) with *DelayStage*. As a result, the execution time of *Stage 1* and *Stage 3* can both be shortened by 8.4% and 28.8%, respectively, due to the improved resource utilization.

In more detail, we calculate the average value and the standard deviation of the network throughput and CPU utilization of a worker node during the execution of the four workloads, which are summarized in Table 3, respectively. Obviously, *DelayStage* achieves higher and more stable CPU utilization and network throughput, in comparison to the stock Spark. In more detail, *DelayStage* improves the average network throughput by 18.3%-81.8% (*e.g.,* from

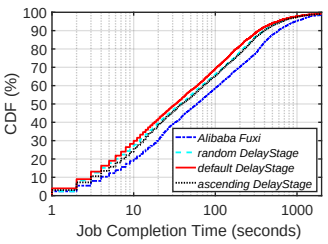


Figure 14: Performance comparison of production jobs in Alibaba trace running with Fuxi [30] and *DelayStage*.

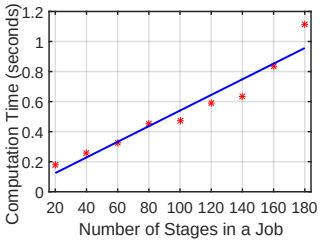


Figure 15: Computation time of *DelayStage* running on an EC2 `m4.large` instance with Alibaba production jobs.

18.1 MB/s to 32.9 MB/s for TriangleCount). Similarly, *DelayStage* also increases the average CPU utilization by 7.2%-28.1% (*e.g.,* from 60.4% to 74.1% for TriangleCount). In particular, the small standard deviation of resource utilization further validates the low variance of job completion time achieved by the *DelayStage* strategy.

5.3 Large-Scale Simulations Driven by Alibaba Cluster Trace

To evaluate the effectiveness and benefits of our *DelayStage* strategy with multiple jobs at scale, we conduct complementary simulations driven by the latest Alibaba cluster trace v2018 [2]. Specifically, we compare the job completion time using our *DelayStage* strategy with that using Alibaba Fuxi scheduler [30], by replaying 2,775,025 jobs on 4,000 machines in our simulation. The numbers of stages range from 4 to 186 for the production jobs in the trace. Each machine is configured with multiple executors by setting the executor number ϵ^w as the number of CPU cores per machine by default. For simplicity, we only consider the network bandwidth heterogeneity in our simulation. We set the network bandwidth $B^{i,w}$ as the value in the range from 100 Mbps to 2 Gbps, and the disk bandwidth D^w is statically set as 80 MB/s. In our simulation, the resources are evenly partitioned among multiple jobs that are concurrently running in the cluster.

Specifically, we compare our default *DelayStage* strategy ($\mathcal{P}_m$ in the descending order) with Alibaba Fuxi [30], and *DelayStage* with the other two kinds of $\mathcal{P}_m$ sequence (in the random order and the ascending order), which are referred to as "random *DelayStage*" and "ascending *DelayStage*", respectively. As shown in Fig. 14, we observe that our default, random, and ascending *DelayStage* strategies as well as Alibaba Fuxi complete each production job with 871, 945, 996, and 1,373 seconds on average, respectively. This indicates that: *First,* our *DelayStrategy* can work well with multiple jobs to reduce the average completion time of production jobs. *Second,* our three *DelayStage* strategies can reduce the average job completion time by 36.6%, 31.2%, and 27.5%, respectively, in comparison to Alibaba Fuxi scheduling strategy. Though Alibaba Fuxi distributes the task execution uniformly to available workers to balance computation and network utilization among workers [30], our *DelayStage* strategy can moderately increase the average CPU and network utilization by 25.4% and 24.8%, respectively, compared to Alibaba Fuxi as shown in Table 4. In addition, our simulation results further validate the effectiveness of the *descending* $\mathcal{P}_m$ sequence adopted in the default *DelayStrategy* as discussed in Sec. 4.1.

Table 4: Average CPU and network utilization of a worker node running production jobs with Alibaba Fuxi [30] and the random, ascending, and default *DelayStage* strategies.

	Fuxi	random	ascending	default
CPU (%)	36.2	43.4	42.2	45.4
Network (%)	42.7	49.1	48.3	53.3

5.4 Runtime Overhead of *DelayStage*

We first evaluate the computation time of our *DelayStage* strategy (*i.e.,* Alg. 1) deployed on one EC2 `m4.large` instance. Specifically, the strategy execution time for ConnectedComponents, CosineSimilarity, LDA, and TriangleCount are 58, 76, 107, and 164 milliseconds, respectively. By repeatedly replaying the Alibaba production jobs with different number of stages, we also record the strategy computation time in simulation. As shown in Fig. 15, the computation time of *DelayStage* shows a roughly linear increase along with the number of stages in a job, which further validates the complexity analysis of our strategy in Sec. 4.1. In addition, 90% of production jobs contain less than 15 stages as analyzed from Alibaba cluster trace in Fig. 2, and the corresponding strategy computation only takes less than 0.2 seconds for these jobs. We further examine the profiling overhead for the four DAG-style data analytics jobs on a sampled input data (*e.g.,* 10%) [27] in our experiment. The profiling time for ConnectedComponents, CosineSimilarity, LDA, and TriangleCount are 104, 143, 45, and 79 seconds, respectively. As the DAG-style data analytics jobs are executed repeatedly in production clusters [27], each job requires profiling only once to obtain the essential parameters. As a result, the overall runtime overhead of *DelayStage* above is acceptable in practice.

6 RELATED WORK

Task scheduling: There have been a number of works on *scheduling tasks within a cluster* to reduce the job response time, by scheduling sub-second tasks [19], task speculation [22], and balancing task loads [30]. To jointly achieve data locality and fairness, Quincy [15] designs a queue-based greedy scheduler to place tasks with locality constraints. Zaharia *et al.* [28] simply delay the map tasks to achieve data locality with slightly relaxed fairness. While sharing the similar idea, *DelayStage* computes the best schedule to answer which stages and how much time to delay for parallel stages, with the aim of achieving high resource utilization and minimizing the job completion time. A more recent work [18] designs Monotasks, which leverages a fine-grained task scheduling model to exploit the resource pipelining and thus avoid resource contention. Monotasks requires modifying the resource allocation in Spark, while *DelayStage* is simple and lightweight by only delaying the submission time of stages.

Recently, *task placement for geo-distributed* analytics (*i.e.,* GDA) has received much attention. To shorten the GDA job completion time, Iridium [20] distributes tasks and their input data among geo-distributed datacenters to achieve the minimal data transfer time. Tetrium [13] further considers the multi-dimensional resource (*i.e.,* CPU, memory, and network) constraints when placing the GDA tasks. To minimize the intra-datacenter traffic, Clarinet [24] designs a WAN-aware query optimizer to place tasks across datacenters.

A recent work [6] proposes a task placement algorithm to achieve max-min fairness in the completion time across GDA jobs. Orthogonal to these task placement-based solutions which answer *where* to execute tasks, *DelayStage* answers *when* to execute the stage by delaying its execution. In addition, we plan to extend *DelayStage* to the geo-distributed setting and examine its effectiveness.

Stage scheduling: Several recent works have been proposed to *schedule stages* by considering the dependencies in job's DAG. To minimize the average job completion time, Tetris [8] places the map or reduce stages from different jobs to satisfy multi-dimensional resource demands. CARBYNE [9] designs an altruistic scheduling of stages which vacates resources for other jobs before or after the stage execution. Graphene [10] identifies the *troublesome* stages of multiple jobs and assigns them to the machines first. While these works focus on the stage placement on workers, *DelayStage* focuses on interleaving stage executions *over time*. A more recent work named AggShuffle [16] leverages the *task-level* network resource interleaving, by proactively aggregating the output data of map tasks to the target datacenters. In contrast, *DelayStage* explores the *stage-level* resource interleaving on *multiple* resources (*i.e.,* CPU, network, and disk) with a larger optimization space. We plan to incorporate *DelayStage* into the prior works above and examine the joint effectiveness in shortening the job completion time.

Job scheduling: There have been works on *arranging the job sequence* to reduce the average job response time. For instance, LAS_MQ[12] leverages multi-level queues to schedule jobs according to the job queue priority. Rasley *et al.* [21] further propose the task queue management on each worker node to reduce the job feedback delays. SWAG [14] re-orders the sub-jobs from multiple jobs in each datacenter according to the job makespan. Similar to the priorities of job queues above, *DelayStage* organizes the *parallel stages* into multiple execution paths with the descending order, and schedules the stage in the long-running execution path with high priority. In addition, our work can be easily extended to reducing the average job completion time in the multi-job environment, as validated by our trace-driven simulation in Sec. 5.3.

7 CONCLUSION

To speed up the performance of DAG-style data analytics jobs, this paper presents the design and implementation of *DelayStage*, a simple stage delay scheduling strategy by interleaving the cluster resources across parallel stages. With a job's DAG, *DelayStage* computes the best time schedule for executing parallel stages to greedily minimize the *makespan* of parallel stages (*i.e.,* the longest path execution time in the DAG) in a job. We implement a prototype of *DelayStage* based on Apache Spark, and conduct extensive prototype experiments on Amazon EC2 as well as large-scale simulations driven by the latest Alibaba cluster trace v2018. Our experiment results demonstrate that *DelayStage* is able to reduce the completion time of DAG-style jobs by up to 41.3% with practically acceptable runtime overhead, compared with the stock Spark and the state-of-the-art stage scheduling strategies.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: A System for Large-Scale Machine Learning. In *Proc. of OSDI*. 265–283.

[2] Alibaba. 2018. Alibaba Cluster Trace Program. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018

[3] Amazon. 2019. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/

[4] Apache. 2019. Apache Flink - Stateful Computations over Data Streams. https://flink.apache.org

[5] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press.

[6] Li Chen, Shuhao Liu, Baochun Li, and Bo Li. 2018. Scheduling jobs across geo-distributed datacenters with max-min fairness. In *Proc. of INFOCOM*. 1–9.

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[8] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. In *Proc. of SIGCOMM*. 455–466.

[9] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *Proc. of OSDI*. 65–80.

[10] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. of OSDI*. 1393–1404.

[11] Zhiming Hu, Baochun Li, Chen Chen, and Xiaodi Ke. 2018. Flowtime: Dynamic scheduling of deadline-aware workflows and ad-hoc jobs. In *Proc. of ICDCS*. 929–938.

[12] Zhiming Hu, Baochun Li, Zheng Qin, and Rick Siow Mong Goh. 2017. Job scheduling without prior information in big data processing systems. In *Proc. of ICDCS*. 572–582.

[13] Chienchun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. 2018. Wide-area analytics with multiple resources. In *Proc. of EuroSys*. 12–12.

[14] Chien Chun Hung, Leana Golubchik, and Minlan Yu. 2015. Scheduling jobs across geo-distributed datacenters. In *Proc. of SoCC*. 111–124.

[15] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proc. of SOSP*. 261–276.

[16] Shuhao Liu, Hao Wang, and Baochun Li. 2017. Optimizing Shuffle in Wide-Area Data Analytics. In *Proc. of ICDCS*. 560–571.

[17] Bernard Marr. 2018. How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. https://www.forbes.com/sites/bernardmarr/

[18] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proc. of SOSP*. 184–200.

[19] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proc. of SOSP*. 69–84.

[20] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low latency geo-distributed data analytics. In *Proc. of SIGCOMM*. 421–434.

[21] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient queue management for cluster scheduling. In *Proc. of EuroSys*. 36–36.

[22] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. 2015. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proc. of SIGCOMM*. 379–392.

[23] Wujie Shao, Fei Xu, Li Chen, Haoyue Zheng, and Fangming Liu. 2019. *Stage Delay Scheduling: Speeding up DAG-style Data Analytics Jobs with Resource Interleaving*. Technical Report. East China Normal University. https://github.com/icloud-ecnu/delaystage/blob/master/delaystage-report.pdf

[24] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. 2016. CLARINET: WAN-Aware Optimization for Analytics Queries. In *Proc. of OSDI*. 435–450.

[25] Ashish Vulimiri, Carlo Curino, Philip Godfrey Brighten, Thomas Jungblut, Jitu Padhye, and George Varghese. 2015. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proc. of NSDI*. 323–336.

[26] Wikipedia. 2019. Wikipedia:Database download. http://en.wikipedia.org/wiki/Wikipedia:Database_download

[27] Fei Xu, Haoyue Zheng, Huan Jiang, Wujie Shao, Haikun Liu, and Zhi Zhou. 2019. Cost-Effective Cloud Server Provisioning for Predictable Performance of Big Data Analytics. *IEEE Transactions on Parallel and Distributed Systems* 30, 5 (2019), 1036–1051.

[28] Matei Zaharia, Dhruba Borthakur, Sarma Joydeep Sen, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of EuroSys*. 265–278.

[29] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, et al. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* (2016), 56–65.

[30] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proc. of VLDB*. 1393–1404.

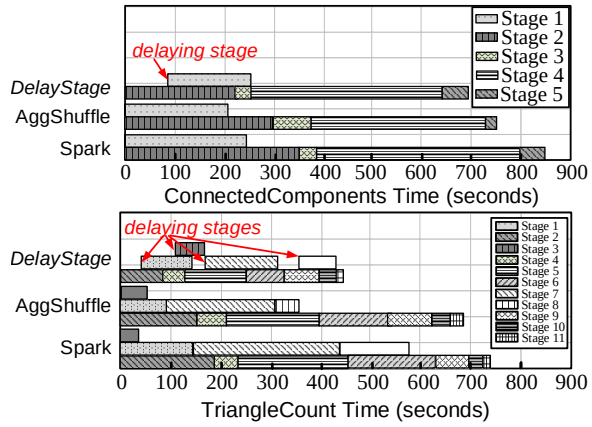


Figure 16: Stage execution time breakdown for ConnectedComponents with 5 stages and TriangleCount with 11 stages.

A SUPPLEMENTARY EXPERIMENT RESULTS

A.1 Stage Execution Time Breakdown for TriangleCount and ConnectedComponents

As shown in Fig. 16, we observe that *DelayStage* simply delays *Stage 1* for ConnectedComponents and a set parallel stages including *Stage 1*, *Stage 3*, *Stage 7*, and *Stage 8* for TriangleCount, respectively, so that the execution time of the longest execution path of parallel stages is shortened by 28.2% for ConnectedComponents and 42.0% for TriangleCount in comparison to the stock Spark. Similar to the explanations of Fig. 11, selectively delaying the execution of stages can exploit multi-dimensional *resource interleaving* across multiple parallel stages, in order to mitigate the resource contention and speed up the execution of the longest stage execution path.

A.2 Prediction Accuracy of Stage Execution Time by Spark Performance Model

As shown in Fig. 17, we observe that our Spark performance model in Sec. 3.1 can accurately predict the stage execution time for a representative Spark job (*i.e.,* LDA), with an error of 1.6%-9.1%. This is because our analytical performance model explicitly considers the stage parallelism during the execution and captures the task/stage execution from the multi-dimensional resources including CPU, network, and disk. In addition, our model is constructed using a set of system-level parameters, which are practically obtained by the job profiling on a single executor, as elaborated in Sec. 4.2. *DelayStage* benefits from an accurate Spark performance model in predicting the stage execution time and decides the scheduling of parallel stages accordingly.

A.3 CPU and Network Resource Utilization of ConnectedComponents and LDA

As shown in Fig. 18(a), the *DelayStage* strategy basically achieves higher network bandwidth utilization compared with the stock Spark, during the job execution of ConnectedComponents and LDA. For example, we simply delay the scheduling of *Stage 1* for ConnectedComponents so as to speed up the execution of the longest execution path of parallel stages (*i.e., Stage 2* and *Stage 3*). *DelayStage* can accordingly fully utilize the network bandwidth by reading the input data of *Stage 4*, during the network idle period from around 250 to 390 seconds running ConnectedComponents in the stock Spark. As *Stage 4* is a sequential stage (*i.e.,* there is no any stages running in parallel with *Stage 4*), the CPU utilization of *DelayStage* shows an idle period from around 270 to 420 seconds (reading the stage input data) as shown in Fig. 18(b). Even though, our strategy can still fill most of the CPU idle period of the stock Spark (*e.g.,* from around 420 to 590 seconds) to speed up the execution of ConnectedComponents. Similarly, by delaying the scheduling of *Stage 1* and *Stage 2* of LDA, the CPU utilization of workers can be maximized by resource interleaving across multiple parallel stages. As evidenced by Fig. 18(b), the CPU idle period of the stock Spark from around 90 seconds to 135 seconds can be filled by *DelayStage* to execute *Stage 2*, *Stage 3* and *Stage 4*.

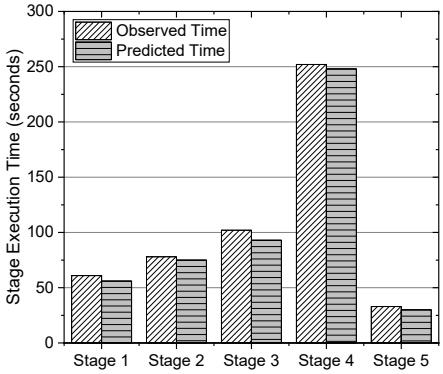
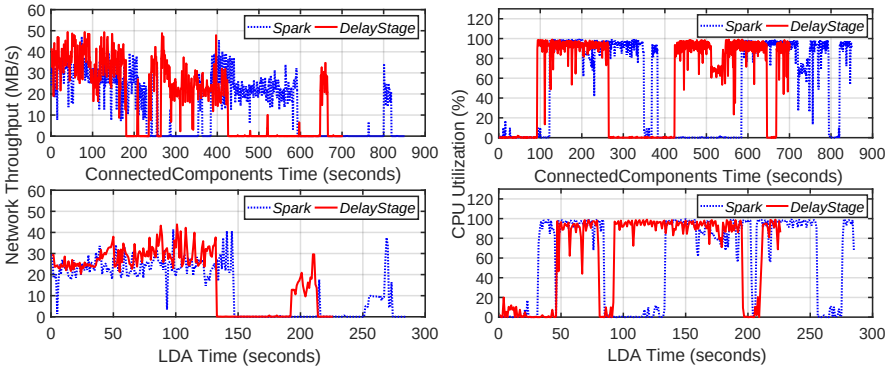


Figure 17: Observed and predicted stage execution time of LDA achieved by our Spark performance model.



(a) (b)

Figure 18: (a) Network throughput and (b) CPU utilization of a worker node running ConnectedComponents and LDA under the stock Spark and *DelayStage* during the job execution.