Sailfish: A Dependency-Aware and Resource Efficient Scheduling for Low Latency in Clouds

Jinwei Liu*, Yingjie Lao†, Ying Mao‡, Rajkumar Buyya§
*Department of Computer and Information Sciences, Florida A&M University, Tallahassee, FL 32307, USA
†Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634, USA
‡Department of Computer and Information Science, Fordham University, Bronx, NY 10458, USA
§School of Computing and Information Systems, The University of Melbourne, Parkville, VIC 3010, Australia
*jinwei.liu@famu.edu, †ylao@clemson.edu, ‡ymao41@fordham.edu, §rbuyya@unimelb.edu.au

2023 IEEE International Conference on Big Data (BigData) | 979-8-3503-2445-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/BigData59044.2023.10386947

***Abstract*—Efficiently scheduling jobs in clouds is critical for job performance, system throughput and resource utilization. The growing importance of parallel applications in clouds introduces challenges in scheduling data-parallel jobs. Production data-parallel jobs increasingly have complex dependency structure, i.e., complex task dependencies expressed as directed acyclic graphs (DAGs), and heterogeneous resource demands. NP-hard problems are introduced by relaxing either of these challenges (i.e., scheduling of homogeneous tasks with dependency constraints or independent and heterogeneous tasks) for scheduler design. It is challenging to design a scheduler for simultaneously achieving low latency and high resource utilization due to the complex dependency structure and job heterogeneity. In this paper, we propose Sailfish, a dependency-aware and resource efficient scheduling for low latency in clouds. Sailfish first uses the machine learning algorithm to classify jobs into two categories (high priority jobs and low priority jobs) based on the extracted features. Next, Sailfish splits the jobs into tasks and distributes the tasks to the master nodes based on the dependency of tasks and the load of master nodes. Then, Sailfish utilizes the dependency information of tasks to determine tasks' priority, and packs tasks by leveraging the complementary of tasks' requirements on different resource types and task dependency. Finally, the master nodes leverage the proposed mutual reinforcement algorithm to distribute tasks to workers in the system based on the resource demands of tasks, the available resources of workers and task dependency. Extensive experimental results based on a real cluster and experiments using real-world Amazon EC2 cloud service show that Sailfish can improve the average resource utilization (by up to 40%) and reduce the latency (the average job completion time) significantly (by up to 91%) compared to the existing schedulers.**

***Index Terms*—scheduling, task dependency, heterogeneity, resource utilization, latency**

I. Introduction

Cloud frameworks tailored for managing and analyzing big datasets are powering ever larger clusters of computers. Batch processing frameworks for data parallel clusters (e.g., MapReduce [1], Dryad [2]) are increasingly used in business environments as part of near real time production systems at Facebook [3] and Microsoft. Data-parallel jobs in clouds have complex dependency structure [4], [5]. Figure 1 shows an example of production data-parallel jobs with complex task dependencies in clouds, and this example shows how to utilize task dependency for improving the system performance. In

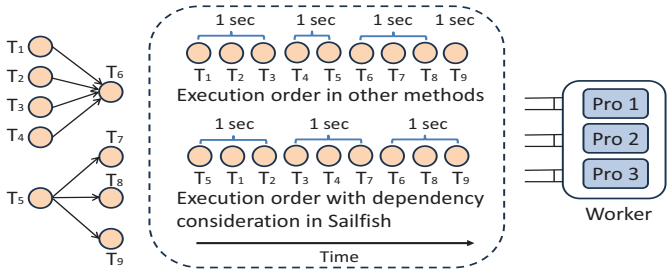


Fig. 1: A motivating example: (1) complex dependencies among tasks of production data-parallel jobs, and (2) leveraging task dependency for improving system performance.

Figure 1, tasks T_1, T_2, T_3, T_4 and T_5 are precedent tasks, and their dependent tasks (e.g., tasks T_6, T_7, T_8, T_9) cannot start running until they finish execution. Inappropriate execution order of tasks may drag down the system performance, but judicious selection of execution order of tasks can help improve the system performance (e.g., improving throughput and reducing latency). Task T_5 has more dependent tasks than tasks T_1, T_2, T_3 and T_4. Executing T_5 at first can enable more dependent tasks to start running after the precedent task T_5 completes, which can help improve the throughput and reduce latency. For simplicity, we assume the execution time of each task is 1 second. The worker can process 3 tasks simultaneously. In Figure 1, we see it requires 4 seconds to process these 9 tasks based on the execution order in other methods because the worker can only process 2 tasks (T_4 and T_5) after finishing executing the tasks T_1, T_2 and T_3 due to the task dependency. However, it only requires 3 seconds to process these tasks based on the execution order in Sailfish. The dependency constraints of tasks challenge the design of schedulers for simultaneously achieving low latency and high resource utilization [4].

Heterogeneous hardware configurations and workloads are increasingly common in modern computer clusters such as Spark [6] and Hadoop [7]. The heterogeneity poses a significant challenge for scheduling in cloud datacenters [8]–[10]. A heterogeneous workload typically consists of many latency-sensitive short jobs (e.g., operations of user-facing services) and a small portion of long batch jobs (e.g., data analytics) that dominate in terms of resource usage [8]–[12]. Duration of jobs

in Google Cluster can vary from tens of seconds to essentially the entire duration of the trace (i.e., 29 days), and most jobs are short jobs [11], [13], [14]. The short jobs are latency sensitive and typically they are assigned a higher priority in scheduling. The long jobs tend to be latency-insensitive, and are usually assigned a lower priority. However, the long jobs consume the bulk of the resources, requiring a high-quality scheduling (e.g., improvement of resource utilization and load balance). The heterogeneity not only poses a challenge for improving latency and throughput, but also challenges resource utilization improvement. Therefore, the scheduler has to be heterogeneity-aware, but it remains a critical challenge to design a scheduler for simultaneously achieving low latency and high resource utilization [8]–[10]. Scheduling heterogeneous DAGs introduces hard algorithmic problems whose optimal solutions are intractable [15], [16].

To address the problem, we propose Sailfish: a dependency-aware and resource efficient scheduling for low latency in clouds. Our goal is to simultaneously improve resource utilization and reduce latency. Sailfish outperforms previous schedulers in that it can well handle the scheduling of heterogeneous jobs with dependency constraints, and simultaneously achieve high resource utilization and low latency by leveraging task dependency to determine task priority, the mutual reinforcement algorithm to predict tasks' waiting time for reducing latency and the task packing strategy to reduce resource fragmentation. We summarize the contribution of this work below.

- We propose Sailfish, a dependency-aware and resource efficient scheduling, which can increase resource utilization and throughput and reduce latency in clouds.
- Sailfish is heterogeneity-aware, and it uses a machine learning based method for job classification based on the extracted features. Sailfish splits jobs into tasks, and assigns the tasks that do not depend on each other to different workers (or different cores of a worker) so that these tasks can be processed in parallel and the latency can thus be reduced. Sailfish leverages task dependency to determine task priority and adequately considers task dependency in scheduling to further reduce latency.
- Sailfish adequately considers the complementary of tasks' requirements on different resource types, and presents a task packing strategy to reduce the resource fragmentation and improve the resource utilization.
- We extract task-level features and worker-level features, and propose a mutual reinforcement algorithm to accurately predict tasks' waiting time so that it can further reduce latency and increase throughput.
- We conduct workload analysis based on the real trace data from a large production cluster, and the analysis results demonstrate that the production cluster has low resource utilization.

The remainder of this paper is organized as follows. Section II shows the workload analysis based on the production workloads. Section III describes the system model used in this paper. Section IV presents the system design and architecture

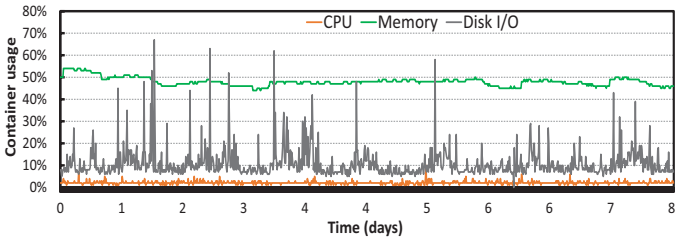


Fig. 2: Resource utilization of containers within 8 days.

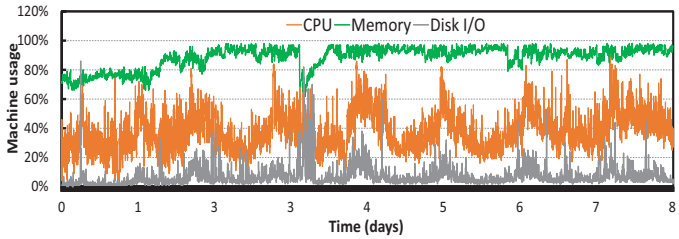


Fig. 3: Resource utilization of machines within 8 days.

of our job scheduling system Sailfish. Section V presents the performance evaluation for Sailfish. Section VI reviews the related work. Section VII concludes the paper with remarks on our future work.

II. Workload Analysis

To verify that the existing systems have low resource utilization, we conducted workload analysis.

A. Low Resource Utilization

We collected the trace data from Alibaba Cluster. The Alibaba Cluster Trace (cluster-trace-v2018) [4] is a datacenter trace for virtual machines (VMs) with batch workloads and DAG information. Cluster-trace-v2018 includes about 4,000 machines in a period of 8 days with CPU and MEM allocation. To measure the resource utilization, we randomly sampled 400 machines and 800 containers.

Figure 2 shows the average resource utilization of containers in Alibaba Cluster Trace. In Figure 2, we see that the resource utilization follows CPU<Disk I/O<Memory, and the average CPU utilization is around 2.12%. The containers in general have low resource utilization. Figure 3 shows the average resource utilization of machines in Alibaba Cluster Trace. In Figure 3, we see that the resource utilization follows Disk I/O<CPU<Memory, and the average Disk I/O utilization is around 4.94%. The machines in general have low resource utilization (CPU and Disk I/O).

III. System Model

In this section, we first introduce the concepts and assumptions, then we formulate our problem based on the concepts and assumptions. Finally, we present our proposed algorithm for improving resource utilization.

A. Concepts and Assumptions

In a cloud system, a job is supposed to be split into m tasks, and the tasks are allocated to workers based on the their requirements on resources. We assume each worker has a

TABLE I: Features for predicting jobs' execution time.

Feature	Description
Job-level features	
Required CPU	Amount of CPU resource required by the job
Required MEM	Amount of MEM resource required by the job
Required storage	Amount of storage resource required by the job
Required GPU	Amount of GPU resource required by the job
# of tasks	Number of tasks which the job contains
System-level features	
CPU utilization	VM's CPU utilization
MEM utilization	VM's MEM utilization
Storage utilization	VM's storage utilization
GPU utilization	VM's GPU utilization

buffer queue which is used for queueing tasks when a worker is allocated to more tasks than it can run concurrently. We assume a dependent task can be executed immediately after the predecessor task. The *completion time* of a task is the time from the submission of the job that contains the task to the time when the task finishes execution. A job completes when all of its tasks finish. *Throughput* is the total number of jobs that complete their execution per time unit.

Problem Statement: Given a certain amount of resources (e.g., CPU, MEM and Storage, etc.) in terms of VMs, resource demands of each job, the dependency constraints of the tasks in each job, and resource capacity constraints of VMs, how to allocate the VM resources to the heterogeneous jobs to achieve high resource utilization and low latency?

Scheduling of tasks in cloud computing environment has been proved to be an NP-hard problem and has a high computational complexity [17], [18]. Therefore, we propose a heuristic method called Sailfish. Sailfish first classifies the jobs into long jobs and short jobs by estimating the run time of jobs [11]. Next, Sailfish splits the jobs into tasks and distributes the tasks to the master nodes based on task dependency and the load of master nodes. Then, Sailfish utilizes the dependency information of tasks to determine task priority (Tasks with more dependent tasks have higher priority), and packs packs complementary tasks. Finally, the master nodes use the proposed mutual reinforcement algorithm to distribute tasks to workers in the system based on the resource demands of tasks, the available resources of workers and task dependency.

B. Job Classification and Job Priority Determination

To classify jobs, Sailfish extracts both job-level features and system-level features (The system-level features reflect the environment in which the jobs will be running, and we extracted the system-level features) [19]–[22], and it uses Support Vector Machine (SVM) to classify jobs into long jobs and short jobs. Specifically, Sailfish uses the extracted features shown in Table I to predict the execution time of a job. If the predicted execution time of the job is longer than the threshold (T_{th}), then the job is a long job; otherwise, the job is a short job. Sailfish sets two priority levels (high priority and low priority) in the job level. In Sailfish, long jobs are considered as low priority jobs, and short jobs are considered as high priority jobs (Long jobs such as graph analytics can tolerate long latencies and tend to be latency-insensitive. However, short jobs such as queries are latency sensitive. Thus, we consider long jobs as low priority jobs and short jobs as high priority jobs in this paper).

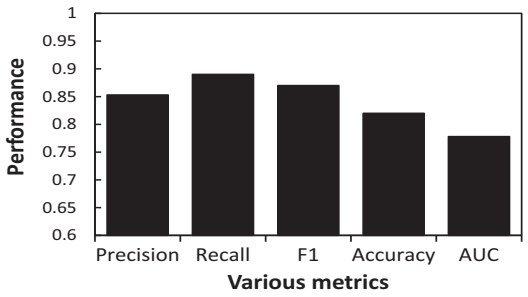


Fig. 4: Performance on different evaluation metrics of Sailfish.

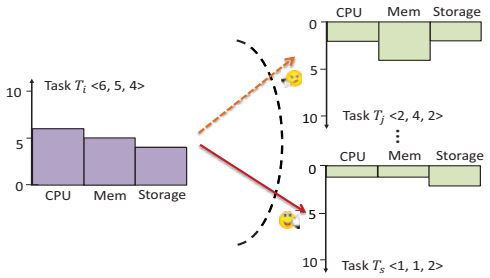


Fig. 5: An example of finding task T_i's complementary task from the given list of tasks to pack with T_i.

In the experiment, Sailfish uses the historical data from google cluster trace [14] to estimate the run time of jobs. Sailfish extracts the numerical values of the features from the historical data for predicting the jobs' execution time. Sailfish considers jobs with execution time no more than 10 minutes as short jobs [23] and jobs with execution time more than 10 minutes as long jobs. Sailfish trains the SVM classifier using the sigmoid kernel function, and it repeatedly divides the dataset into training and testing sets using 10-fold cross-validation with percentage split and performs classification. Figure 4 shows the result of the performance on different evaluation metrics of Sailfish's job classification. In Figure 4, we see that most of the values of the evaluation metrics are higher than 0.82. Therefore, the SVM algorithm can be used to determine job priority by classifying jobs into long jobs and short jobs based on the extracted features.

C. Task Priority Determination

Leveraging task dependency to determine task priority can help reduce the latency. In order to avoid starvation of long running tasks and reduce the waiting time of tasks, we follow the method in the work [24] to consider the priority of task T_{ij} as a function of its remaining time t_{ij}^{rem} and its waiting time t_{ij}^{w} based on the dependency relations among T_{ij} and its children (i.e., the tasks depending on T_{ij}) and determine task priority.

D. Improving Resource Utilization

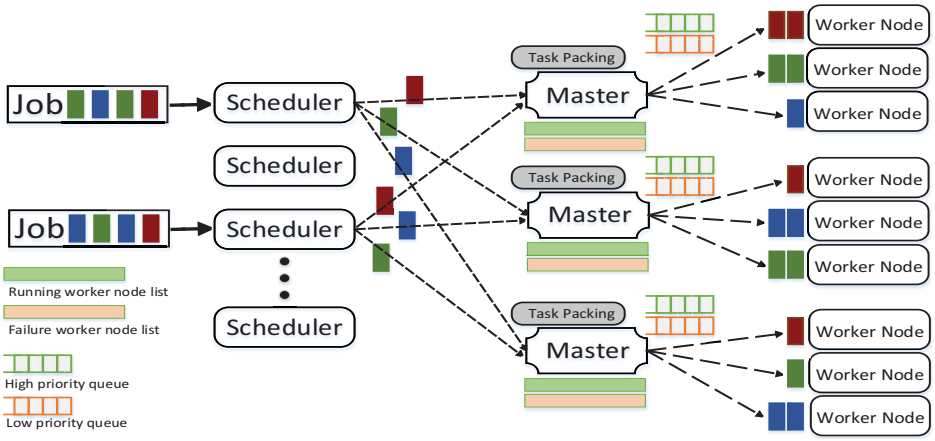


Fig. 6: Architecture of Sailfish.

Resource fragmentation leads to low resource utilization. To reduce resource fragmentation and improve resource utilization, Sailfish presents a task packing strategy which packs the tasks with complementary dominant resources (A *dominant resource* is defined as the resource type on which the task has the highest demand) such that the summation of the deviation of the two tasks' resource demands on each resource type is the largest (see Eq. (1)). To find T_i's complementary task, Sailfish uses Eq. (1) to calculate its deviation with every other task T_j whose dominant resource is different from that of T_i.

$$DV(j,i) = \sum_{k=1}^{l}((d_{jk} - \frac{d_{jk}+d_{ik}}{2})^2 + (d_{ik} - \frac{d_{jk}+d_{ik}}{2})^2), \quad (1)$$

where d_{ik} is T_i's resource demand on resource type k. Finally, the task with the highest deviation value is the complementary task of T_i. This method can also be applied to task packing for more tasks (e.g., three tasks). Figure 5 shows an example of finding task T_i's complementary task from the given list of tasks to pack with T_i based on Eq. (1). In this example, task T_i's resource demands on CPU, memory and storage are 6, 5 and 4, respectively; task T_j's resource demands on CPU, memory and storage are 2, 4 and 2, respectively; task T_s's resource demands on CPU, memory and storage are 1, 1 and 2, respectively. After calculating its deviation with the other tasks T_j and T_s, Sailfish chooses task T_s to pack with T_i because $DV(s,i) = 22.5 > DV(j,i) = 10.5$. If the task T_i's complementary task cannot be found from the given list of tasks, in this case, Sailfish assigns the task entity comprising a single task T_i to a server.

E. Resource Allocation

After packing tasks using the task packing strategy in Section III-D, Sailfish assigns each task entity (packed tasks or a task) to a server that has the least remaining resources (called most matched server) and can meet the resource demand of the task entity, which can help further improve the resource utilization.

IV. System Design and Architecture

This section presents the system design and architecture of Sailfish. Figure 6 shows the architecture of Sailfish. The system comprises multiple distributed schedulers, masters and workers. After a user submits a job to the cloud system, the system searches the scheduler that is not heavily loaded and has the smallest geographic distance to the user, and delivers the job to the scheduler. The scheduler first splits the job into tasks, then it randomly selects r masters that are not heavily loaded by taking into account task dependency and distributes the tasks to the selected masters (see Algorithm 1). After receiving the tasks, the masters pack complementary tasks using the task packing strategy in Section III-D. Each master maintains two task queues (high priority queue and low priority queue): the high priority queue stores tasks of short jobs and the low priority queue stores tasks of long jobs. Next, masters use the mutual reinforcement algorithm in Section IV-A and the resource allocation algorithm to assign task entity to workers, and the workers calculate the priority of tasks assigned to them and run tasks. In Sailfish, each scheduler has a unique key, and each master records the resource information of the workers associated with master.

Algorithm 1: Job_Processing()

```
   Input: A submitted job
1  A Job is submitted
2  sort(s[]) // Sort schedulers by distances between
       schedulers and the user submitting the job
3  for i ← 1 to size(s) do
4      if s[i] is lightly loaded then
5          Assign the job to s[i] // Assign the job to the
               scheduler
6          s[i] split the job into m tasks
7          s[i] distributes the tasks to r randomly selected masters
             that are not heavily loaded by taking into account task
             dependency
```

A. Task Scheduling

1) Fine-grained Waiting Time Prediction: Many previous studies using sample-based techniques assign tasks to workers based on the queue length at workers [25], [26]. However,

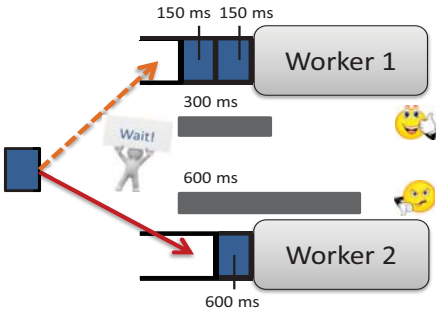


Fig. 7: An example of coarse-grained estimate of waiting time based on queue length.

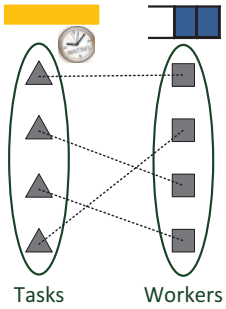


Fig. 8: Bipartite graph for tasks and workers.

queue length at workers is not sufficient for learning about the waiting time because queue length provides only a coarse-grained estimation of waiting time. Figure 7 shows an example in which a scheduler tries to assign a task to one of two workers. There are two 150 ms tasks waiting for execution in the queue of Worker 1, and there is one 600 ms task in the queue of Worker 2. The schedulers placing tasks based on queue length will assign the task to Worker 2 though it will result in a 300 ms longer waiting time. In addition, many previous methods of placing tasks based on the queue length rely on the global queue-length information. However, tracking the global queue-length information is both time and resource consuming [27]. To handle these issues, we propose a fine-grained approach to predict tasks' waiting time, and we extract two types of features: task-level features and worker-level features.

Mutual Reinforcement Algorithm: We propose a graph-based Semi-supervised Learning (SSL) algorithm called Mutual Reinforcement Label Propagation (MRLP) to accurately predict tasks' waiting time. Previous studies [28]–[31] show that cluster workloads contain a large number of recurring jobs (More than 60% of the jobs are recurring in the production clusters at Microsoft [12]). We assume an initial estimate of task waiting time based on previous executions [12].

Let $\mathcal{T}$ be the set of tasks that will be placed at the end of queues of workers. Let $\mathcal{W}$ be the set of workers. The identified features showed in Table II form the feature space of tasks ($\mathcal{X}(\mathcal{T})$) and workers ($\mathcal{X}(\mathcal{W})$). Sailfish uses a graph-based model to learn task waiting time. Sailfish models the relation between tasks and workers by using a bipartite graph $G_{ij} = \{N_{ij}, E_{ij}\}$ (see Figure 8), where N_{ij} is the set of

TABLE II: Summary of features extracted from tasks and workers.

Feature	Description
Task-level features	
# of predecessor tasks	# of predecessor tasks in the queue of a worker
# of dependent tasks	# of dependent tasks in the queue of a worker
Job priority	Priority of the job from which the task is
CPU demand	Amount of CPU resource required by the task
MEM demand	Amount of MEM resource required by the task
storage demand	Amount of storage resource required by the task
GPU demand	Amount of GPU resource required by the task
Worker-level features	
# of running tasks	# of running tasks at a worker
# of cores	# of cores of a worker
Amount of MEM	Amount of memory of the worker
Amount of storage	Amount of storage of the worker
Amount of GPU	Amount of GPU resource of the worker
CPU utilization	Worker's CPU utilization
MEM utilization	Worker's MEM utilization
Storage utilization	Worker's storage utilization
GPU utilization	Worker's GPU utilization

nodes and E_{ij} is the set of undirected edges. There is an edge between a task node and a worker node if the task completed execution on the worker node. In the model, each edge linking a task and a worker represents the waiting time mapped to a worker with a queue length. Below we explain how the model works. Suppose there are m tasks that are assigned to n worker nodes. Let T be the vector $(n \times 1)$ of tasks' waiting time, and Q be the vector $(m \times 1)$ of the queue lengths of worker nodes. Define an $m \times n$ matrix E, where $e_{ij} = 1(i \in [1, m], j \in [1, n])$ means task t_i is added to the queue q_j, otherwise $e_{ij} = 0$. Then we can get E' from E

$$E'_{ij} = e_{ij} / \sum_{k=1}^{n} e_{ik}. \tag{2}$$

For the task part of the bipartite graph, an edge connects any two tasks (i.e., t_i, t_j) if they are in the same category (Both belong to the same job type, e.g., short/long job). The weight for the edge linking t_i and t_j is represented by $w_t(t_i, t_j)$, which is calculated based on the cosine similarity between the features of two tasks x_i^t and x_j^t:

$$w_t(t_i, t_j) = exp(-\|x_i^t - x_j^t\|^2 / \lambda_t^2), \tag{3}$$

where λ_t is a weighting parameter, $w_t(t_i, t_j)$ is set to be 0 if t_i and t_j belong to two different categories. In addition, $w_t(t_i, t_i) = 0$.

Define an $n \times n$ probabilistic transition matrix N:

$$N_{ij} = P(t_i \to t_j) = w_t(t_i, t_j) / \sum_{k=1}^{n} w_t(t_i, t_k), \tag{4}$$

where N_{ij} is the probability of transit from t_i to t_j. An edge connects any two workers (i.e., w_i, w_j) that have completed tasks in the same category for worker part of the graph. The weight for the edge linking q_i and q_j is represented by $w_q(q_i, q_j)$, which is calculated based on the cosine similarity between the features of two workers (worker-level features) x_i^q and x_j^q:

$$w_q(q_i, q_j) = exp(-\|x_i^q - x_j^q\|^2 / \lambda_q^2), \tag{5}$$

where λ_q is a weighting parameter, $w_q(q_i, q_j)$ is set to be 0 if the workers w_i (with the queue q_i) and w_j (with the queue q_j) have not completed tasks belonging to the same category. In addition, $w_q(q_i, q_i) = 0$.

Then, we define an $m \times m$ probabilistic transition matrix

$$M_{ij} = P(q_i \to q_j) = w_q(q_i, q_j) / \sum_{k=1}^{m} w_q(q_i, q_k). \quad (6)$$

Given some known (labeled) examples of T and Q. The following equation can be used to estimate the workers' queue lengths from their neighbors and their tasks' waiting times.

$$Q_{c+1} = \alpha M Q_c + (1 - \alpha) E' T_c. \quad (7)$$

Correspondingly, the equation below can be used to estimate tasks' waiting time from their neighbors and workers' queue lengths.

$$T_{c+1} = \beta N T_c + (1 - \beta) E^T Q_{c+1}. \quad (8)$$

Repeating a certain number of times, all tasks' waiting times can be estimated. The steps for iteratively finding waiting time and queue length are shown in Algorithm 2 as follows:

Algorithm 2: Pseudocode for iteratively finding waiting time and queue length

Input: waiting time feature vector T_0, queue length feature vector Q_0, weighting coefficients α, β, some manual labels of T_0 and/or Q_0
Output: Waiting time T and queue length Q.
1 Set c=0
2 **while** *not convergence* **do**
3 Propagate waiting time. $T_{c+1} \leftarrow \beta N T_c + (1 - \beta) E^T Q_{c+1}$ // Formula (8)
4 Propagate queue length. $Q_{c+1} \leftarrow \alpha M Q_c + (1 - \alpha) E' T_c$ // Formula (7)
5 Clamp the labeled data of T_{c+1} and Q_{c+1}
6 Set $c = c + 1$
7 **return** Q, T

Algorithm 2 shows the pseudocode for iteratively finding tasks' waiting time and queue length. The algorithm first propagates tasks' waiting time by estimating tasks' waiting time from their neighbors and the queue length of workers (line 3). Second, the algorithm propagates queue length of workers by estimating queue length from their neighbors and tasks' waiting time (line 4). Then, it clamps the labeled data of tasks' waiting time and queue length of workers (line 5). After repeating a certain number of times, the algorithm can estimate all tasks' waiting time and the queue length of workers.

2) Task Scheduling: Algorithm 3 shows the pseudocode of task scheduling. In Algorithm 3, Sailfish first packs tasks that have complementary resource requirements based on the task packing strategy. Then, the masters assign task entity to workers based on Algorithm 2.

Algorithm 3: Task_Scheduling()

Input: A set of tasks
1 Pack tasks that have complementary resource requirements based on the task packing strategy
2 Masters assign task entity to workers based on the available resources of workers and the mutual reinforcement algorithm `// Algorithm 2`

V. Performance Evaluation

In this section, we present our trace-driven experimental results. We first conducted experiments on a large-scale real cluster [32]. Then, we conducted experiments on the real-world Amazon EC2 [33] to further evaluate the performance of our method.

TABLE III: Parameter settings.

Parameter	Meaning	Setting
n	# of servers	30-50
h	# of jobs	100-2,500
m	# of tasks of a job	10-2,000
L_{th}^{s}	Threshold for heavily loaded scheduler	0.85
L_{th}^{m}	Threshold for heavily loaded master	0.85
α	A certain coefficient $\in (0, 1)$	0.5
β	A certain coefficient $\in (0, 1)$	0.5
γ	A certain coefficient $\in (0, 1)$	0.5 [24]
ω_1	Weight for task's remaining time	0.5
ω_2	Weight for task's waiting time	0.3
ω_3	Weight for task's allowable waiting time	0.2

To show the performance of Sailfish, we compared the results of our method and the other three methods Pigeon [8], Eagle [10] and Sparrow [25]. We first deployed our testbed in a large-scale cluster. We implemented our method and other three methods in our testbed.

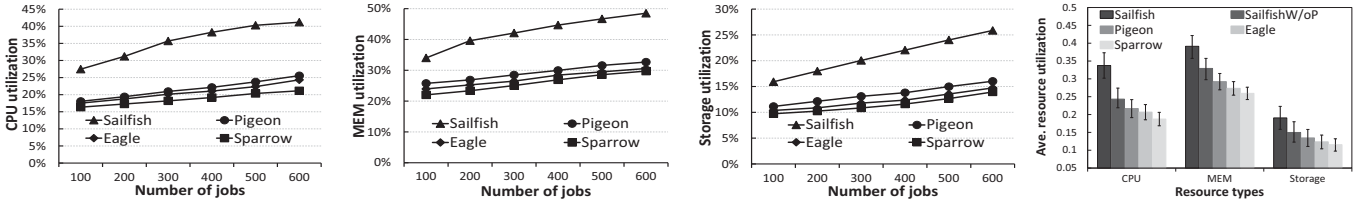
Pigeon [8]. Pigeon is a distributed hierarchical job scheduler for datacenters, which aims to ensure low latency. Pigeon divides workers in a cluster into groups, and the master in each group centrally manages all the tasks handled by the group.

Eagle [10]. Eagle is a hybrid data center scheduler for data-parallel programs, and a centralized scheduler handles long jobs and distributed schedulers handle short jobs.

Sparrow [25]. Sparrow is a stateless decentralized scheduler that provides near optimal performance using two key techniques: batch sampling and late binding.

We first deployed our testbed on 50 servers in a large-scale real cluster. Users' job arrival rates follow a Poisson distribution [34]. The servers in the real cluster are from Sun X2200 servers (AMD Opteron 2356 CPU, 16GB memory). We then conducted experiments on 30 instances in the real-world Amazon EC2 and the instances in EC2 are from commercial product HP ProLiant ML110 G5 servers (2660 MIPS CPU, 4GB memory). We considered each instance as a server. In both real cluster and EC2 experiments, we set the bandwidth capacity and the disk storage capacity of each server (instance) to 1GB/s bandwidth and 720GB, respectively.

In each experiment, we varied the number of heterogeneous jobs from 100 to 600 with step size of 100. The Google cluster trace data [14] was used to set the parameters. The Google cluster trace [14] records resource usage on a cluster of about 11,000 machines from May 2011 for 29 days. We randomly chose tasks from the jobs in the period between May



(a) CPU (b) Memory (c) Storage (d) Average

Fig. 9: Utilizations of different resource types vs. number of jobs of different methods on a real cluster.

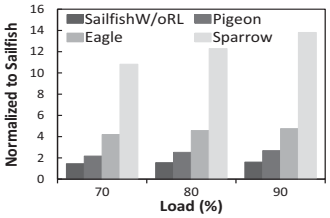


Fig. 10: Job completion times normalized to Sailfish on a real cluster.

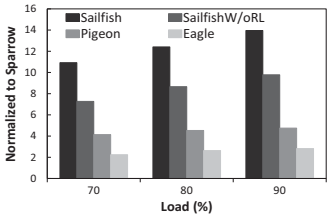


Fig. 11: Throughput normalized to Sparrow on a real cluster.

1 to May 7. We set the CPU and memory consumption, and execution time for each task based on the Google cluster trace, and we set the disk and bandwidth consumption for each task to 0.02MB [35] and 0.02MB/s [36], [37], respectively. In the experiment, we created the dependency relationship between tasks based on tasks' starting time and ending time from the trace. When there is no overlap between the execution times of two tasks of a job, we can create a dependency relationship between the two tasks. We constrained the number of levels in a created dependency DAG within five and the number of dependent tasks on a task within fifteen [15]. Table III shows the parameter settings in our experiment unless otherwise specified.

A. Experimental Results on A Real Cluster

Figures 9(a), 9(b) and 9(c) show the relationship between the resource (CPU, Memory and Storage) utilization and the number of jobs on a real cluster. In these figures, we see that the resource utilization follows Sailfish>Pigeon>Eagle>Sparrow. The resource utilization in Sailfish is higher than that in Pigeon because Sailfish leverages complementarity of tasks' demands on different resource types and uses a task packing strategy to reduce the resource fragmentation and improve the resource utilization. Also, Sailfish leverages task dependency to determine task priority and adequately considers task dependency in scheduling, which can enable more (dependent) tasks to run in parallel and thus improve the resource utilization. The resource utilization in Pigeon is higher than that in Eagle and Sparrow because all the workers in a group are shared among tasks from short jobs in a work-conserving manner, while all the low priority workers are shared by tasks from long jobs. The resource utilization in Eagle is higher than that in Sparrow because Eagle utilizes Succinct State Sharing (SSS) to improve resource utilization by dynamically allowing short jobs to run in the general partition.

We also measured the average resource utilization of different resource types in different methods. To test the effectiveness of task packing in increasing resource utilization, we also evaluated the performance of SailfishW/oP, a variant of Sailfish in which job packing is not used. Figure 9(d) shows the average resource utilization of different resource types in different methods. We observe that the average resource utilization follows Sailfish>SailfishW/oP>Pigeon>Eagle>Sparrow due to the same reasons.

We evaluated the overhead of different methods by measuring the job completion time of each method on the real cluster. To test the effectiveness of the mutual reinforcement algorithm (MRLP) in reducing job completion time, we also evaluated the performance of SailfishW/oRL, a variant of Sailfish in which the mutual reinforcement algorithm is not used. We followed the work [8], [10] to measure the job completion time of SailfishW/oRL, Pigeon, Eagle and Sparrow normalized to Sailfish under different cluster loads. Figure 10 shows different methods' job completion time normalized to Sailfish under different cluster loads. In Figure 10, we see that the job completion time follows Sailfish<SailfishW/oRL<Pigeon<Eagle<Sparrow. This is because Sailfish splits jobs into tasks by taking into account the constraints (e.g., the dependency among tasks) of tasks, and assigns the tasks that do not depend on each other to different machines so that these tasks can be processed in parallel and the latency (job completion time) can thus be reduced. Also, Sailfish leverages task dependency to determine task priority and adequately considers task dependency in scheduling to further reduce latency. Moreover, Sailfish uses the mutual reinforcement algorithm to accurately predict task duration, which can further reduce the latency. As Sparrow does not distinguish between short jobs and long jobs, it incurs up to 13.9, 8.6, 5.2 and 2.9 times longer job completion time than Sailfish, Pigeon and Eagle, respectively.

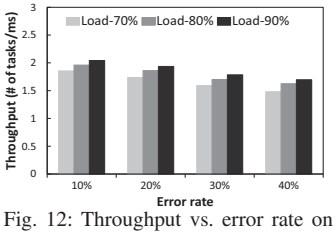
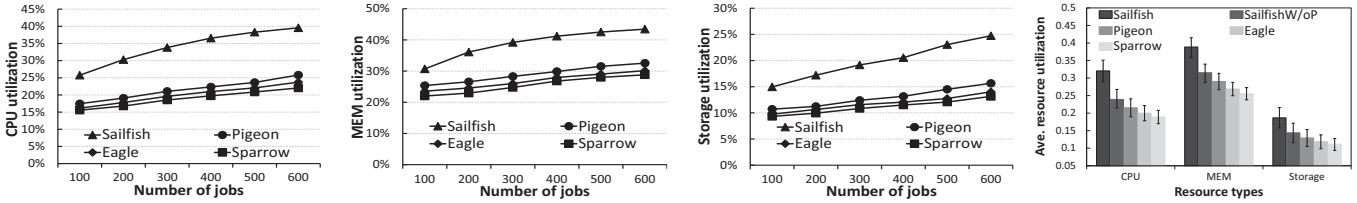


Fig. 12: Throughput vs. error rate on a real cluster.

We also measured the throughput of different methods on the real cluster. To test the effectiveness of the mutual reinforcement algorithm in increasing throughput, we also evaluated the performance of SailfishW/oRL. We measured the throughput of Sailfish, SailfishW/oRL, Pigeon and Eagle normalized to Sparrow under different cluster loads. Figure 11 shows different methods' throughput normalized to Sparrow under different cluster loads.



(a) CPU (b) Memory (c) Storage (d) Average

Fig. 13: Utilizations of different resource types vs. number of jobs of different methods on Amazon EC2.

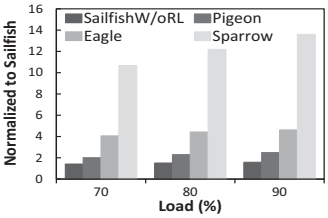


Fig. 14: Job completion times normalized to Sailfish on Amazon EC2.

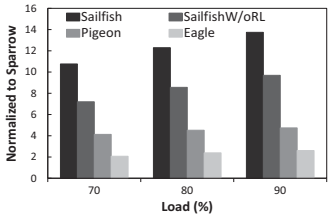


Fig. 15: Throughput normalized to Sparrow on Amazon EC2.

In Figure 11, we see that the throughput follows Sailfish>SailfishW/oRL>Pigeon>Eagle>Sparrow. This is because Sailfish splits jobs into tasks by taking into account the constraints (e.g., the dependency among tasks) of tasks, and assigns the tasks that do not depend on each other to different machines so that these tasks can be processed in parallel and the throughput can thus be increased. Also, Sailfish leverages task dependency to determine task priority and adequately considers task dependency in scheduling to further increase throughput. Moreover, Sailfish uses the mutual reinforcement algorithm to accurately predict task duration, which can further increase the throughput. As Sparrow does not distinguish between short jobs and long jobs, it incurs up to 14.0, 9.7, 4.8 and 2.8 times lower throughput than Sailfish, SailfishW/oRL, Pigeon and Eagle, respectively.

To test the effectiveness of the mutual reinforcement algorithm, we also examined the relationship between throughput and the prediction accuracy of the reinforcement learning algorithm on the real cluster. Figure 12 shows the relationship between Sailfish's throughput and the error rate of the mutual reinforcement algorithm under different cluster loads. In Figure 12, we see that the throughput decreases as the error rate increases. This is because the prediction accuracy decreases as the error rate increases, and lower prediction accuracy can increase the waiting time of tasks in the queue of workers.

B. Experimental Results on Amazon EC2

To fully test the performance of our method, we also conducted experiments on the real-world Amazon EC2. Figures 13(a), 13(b) and 13(c) show the relationship between the resource (CPU, Memory and Storage) utilization and the number of jobs on Amazon EC2. We observe that the resource utilization follows Sailfish>Pigeon>Eagle>Sparrow. The resource utilization in Sailfish is higher than that in Pigeon because Sailfish leverages complementarity of tasks' demands on different resource types and uses a task packing strategy to reduce the resource fragmentation and improve the resource utilization. The resource utilizations in Pigeon is higher than that in Eagle and Sparrow because all the workers in a group are shared among tasks from short jobs in a work-conserving manner, while all the low priority workers are shared by tasks from long jobs. To further test the effectiveness of the reinforcement learning algorithm in reducing job completion time and verify the performance of resource utilization of different methods, we also evaluated the performance of SailfishW/oP and measured the average resource utilization of different resource types in different methods. Figure 13(d) shows the average resource utilization of different resource types in different methods. We observe that the average resource utilization follows Sailfish>SailfishW/oP>Pigeon>Eagle>Sparrow due to the same reasons.

We also measured the job completion time of SailfishW/oRL, Pigeon, Eagle and Sparrow normalized to Sailfish under different cluster loads on Amazon EC2. Figure 14 shows different methods' job completion time normalized to Sailfish under different cluster loads. In Figure 14, we see that the job completion time follows Sailfish<SailfishW/oRL<Pigeon<Eagle<Sparrow. As Sparrow does not distinguish between short jobs and long jobs, it incurs up to 13.6, 8.5, 5.4 and 2.8 longer job completion time than Sailfish and Pigeon, respectively. The result in Figure 14 is consistent with that in Figure 10, and the reasons are the same as that explained in Figure 10.

We also measured the throughput of Sailfish, SailfishW/oRL, Pigeon and Eagle normalized to Sparrow under different cluster loads on Amazon EC2. Figure 15 shows different methods' throughput normalized to Sparrow under different cluster loads. In Figure 15, we see that the throughput follows Sailfish>SailfishW/oRL>Pigeon>Eagle>Sparrow due to the same reasons explained in Figure 11. As Sparrow does not distinguish between short jobs and long jobs, it incurs up to 13.7, 9.6, 4.7 and 2.6 times lower throughput than Sailfish, SailfishW/oRL, Pigeon and Eagle, respectively.

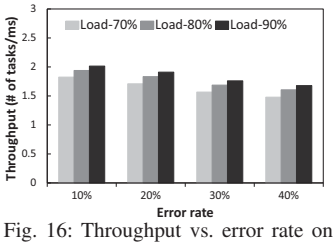


Fig. 16: Throughput vs. error rate on Amazon EC2.

We also examined the relationship between throughput and the prediction accuracy of the mutual reinforcement algorithm on Amazon EC2. Figure 16 shows the relationship between Sailfish's throughput and the error rate of the reinforcement

learning algorithm under different cluster loads. In Figure 16, we observe similar result due to the same reasons explained in Figure 12.

VI. Related Work

The related work is split into two categories: job/task scheduling for low latency (or short job completion time) and job/task scheduling for high resource utilization. First, we review prior work falling into these two categories. Then, we indicate the advantages of Sailfish compared to the previous methods.

A. Scheduling for Reducing Latency or Job Completion Time

Many methods have been proposed to handle job/task scheduling for low latency (or short job completion time or high throughput). Ousterhout *et al.* [25] proposed Sparrow, a distributed, low latency scheduling that provides near optimal performance using two key techniques: batch sampling and late binding. To compensate for the occasional poor scheduling decisions made by the distributed job scheduling such as Sparrow, Delgado *et al.* [11] proposed Hawk, a hybrid scheduler, staking a middle ground between centralized and distributed schedulers for improving the runtime of jobs. To overcome the limitation (only partially avoiding head-of-line blocking) of hybrid schedulers such as Hawk [11] and Mercury [38], Delgado *et al.* [10] proposed Eagle, a hybrid data center scheduler for data-parallel programs. Eagle dynamically divides the nodes of the data center in partitions for the execution of long and short jobs to avoid head-of-line blocking, and introduces sticky batch probing to achieve better job-awareness. Mohan *et al.* [39] proposed Synergy, a resource sensitive scheduler for shared GPU clusters. Synergy infers the sensitivity of DNNs to different resources using optimistic profiling, and performs such multi-resource workload-aware assignments across a set of jobs scheduled on shared multi-tenant clusters using a new near-optimal online algorithm. However, they neglect the dependency between tasks and thus cannot utilize the dependency information to reduce the latency to the minimum by running tasks that are independent of each other in parallel. To reduce job completion time, Wang *et al.* [8] proposed Pigeon, a distributed, hierarchical job scheduler that employs a divide-and-conquer approach in task scheduling. In Pigeon, workers are divided into groups. Each group has a master worker which centrally manages all the tasks handled by the group. Jajoo *et al.* [40] proposed SLEARN, a task-sampling-based approach, to learn job properties in the spatial dimension for Cluster Job Scheduling. SLEARN exploits the similarity among runtime properties of the tasks of the same job. SLEARN first proactively samples and schedules a small fraction of the tasks of each job, then it uses the observed runtime properties of those tasks to estimate those of the whole job. However, Pigeon and SLEARN neglect the complementary resource requirements of tasks, and they cannot fully increase the resource utilization. Also, they do not leverage task dependency information to determine task priority for reducing the latency.

B. Scheduling for Improving Resource Utilization

There are many studies on job/task scheduling for high resource utilization. But the resource constraints are ignored in many research studies. To improve resource utilization, some studies [30], [41] pack tasks/jobs to machines based on their requirements of all resource types for reducing resource fragmentation. Grandl *et al.* [41] proposed Tetris, a multi-resource cluster scheduler that packs tasks to machines based on their requirements along multiple resources. Specifically, Tetris first selects the set of tasks whose peak usage of each resource type can be accommodated on a machine when the resources of the machine become available. For each task in the set, Tetris computes an alignment score (a weighted dot product between the vector of machine's available resources and the task's peak usage of resources) to the machine, and the task with the highest alignment score is scheduled and allocated its peak resource demands. Grandl *et al.* [30] proposed an online altruistic multi resource DAG scheduler CARBYNE which packs tasks for improving resource utilization. However, these methods do not consider job heterogeneity (job heterogeneity in both execution time and resource demands for task packing. Zhao *et al.* [42] proposed Muri, a multi-resource cluster scheduler for DL workloads. Muri exploits multi-resource interleaving of DL training jobs to achieve high resource utilization and reduce job completion time). To maximize interleaving efficiency, Muri uses a scheduling algorithm based on Blossom algorithm for multi-resource multi-job packing.

Unlike the existing approaches, Sailfish judiciously utilizes task dependency information to determine task priority by prioritizing tasks that will subsequently enable the execution of more dependent tasks, which helps reduce the latency. Also, Sailfish extracts task- and worker-level features, and presents a mutual reinforcement algorithm to accurately predict task waiting time for further reducing the latency and increasing throughput. In addition, Sailfish considers the complementary of tasks' requirements on different resource types, and packs complementary tasks, and then allocates resources to the packed tasks, which helps reduce the resource fragmentation and improve the resource utilization.

VII. Conclusions and Future Work

In this paper, we present a dependency-aware and resource efficient scheduling with low latency in clouds. Sailfish judiciously leverages task dependency information to determine task priority by prioritizing tasks that will subsequently enable the execution of more dependent tasks, which helps reduce the latency and increase throughput. Also, Sailfish assigns the tasks that are independent of each other to different workers/cores so that these tasks can be processed in parallel and the latency can thus be reduced. Moreover, Sailfish extracts task- and worker-level features, and presents a mutual reinforcement algorithm to accurately predict task waiting time for further reducing the latency and increasing throughput. In addition, Sailfish takes into account the complementary of tasks' requirements on different resource types, and packs complementary tasks, and then allocates resources to the

packed tasks based on their resource demands, which helps improve the resource utilization. We compare Sailfish with the existing methods under different scenarios using a real cluster and Amazon EC2 cloud service, and demonstrate that Sailfish outperforms the exiting methods (including a hierarchical scheduler, a hybrid data center scheduler and a decentralized scheduler) under both the real cluster and Amazon EC2 cloud service. In the future, we will use different cloud/cluster workloads to fully verify the performance of our method. Also, we will consider data locality, fairness, cross-job dependency, and the scenario that new tasks are dynamically added which extends the task-dependency graph for further optimizing the performance of Sailfish. In addition, we will consider fault tolerance and energy efficiency in designing a low-latency and resource-efficient scheduling system so that the system can handle node failures/crashes or straggler.

ACKNOWLEDGMENT

This research was supported in part by the Faculty Research Awards Program at Florida A&M University.

REFERENCES

[1] 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.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of Eurosys*, pages 59–72, March 2007.

[3] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A.Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at facebook. In *Proc. of SIGMOD*, 2011.

[4] H. Tian, Y. Zheng, and W. Wang. Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud. In *Proc. of SoCC*, Santa Cruz, 2019.

[5] A. Chung, S. Krishnan, K. Karanasos, C. Curino, and G. R. Ganger. Unearthing inter-job dependencies for better cluster scheduling. In *Proc. of OSDI*, 2020.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of NSDI*, 2012.

[7] Hadoop. http://hadoop.apache.org/ [accessed in August 2023].

[8] Z. Wang, H. Li, Z. Li, X. Sun, J. Rao, H. Che, and H. Jiang. Pigeon: an effective distributed, hierarchical datacenter job scheduler. In *Proc. of SoCC*, 2019.

[9] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proc. of SoCC*, 2018.

[10] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proc. of SoCC*, Santa Clara, 2016.

[11] P. Delgado, F. Dinu, A. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of ATC*, 2015.

[12] J. Rasley, K. Karanasosy, S. Kandulay, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Porc. of EuroSys*, London, 2016.

[13] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of SoCC*, San Jose, October 2012.

[14] Google trace. https://code.google.com/p/googleclusterdata/ [accessed in August 2023].

[15] 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*, Savannah, 2016.

[16] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proc. of SIGCOMM*, 2019.

[17] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[18] A. Gorbenko and V. Popov. Task-resource scheduling problem. *International Journal of Automation and Computing*, 9(4):429–441, 2012.

[19] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. Hacigümüs. Activesla: A profit-oriented admission control framework for database-as-a-service providers. In *Proc. of SoCC*, Cascais, October 2011.

[20] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proc. of SoCC*, Seattle, 2014.

[21] L. Shao, Y. Zhu, S. Liu, A. Eswaran, K. Lieber, J. Mahajan, M. Thigpen, S. Darbha, S. Krishnan, S. Srinivasan, C. Curino, and K. Karanasos. Griffon: Reasoning about job anomalies with unlabeled data in cloud-based platforms. In *Proc. of SoCC*, Santa Cruz, 2019.

[22] A. Ganapathi, Y. Chen, A. Fox, R. H. Katz, and D. A. Patterson. Statistics-driven workload modeling for the cloud. In *Proc. of SMDB*, pages 87–92, 2010.

[23] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. In *Proc. of VLDB Endowment*, volume 5(12), pages 1802–1813, 2012.

[24] J. Liu, H. Shen, A. Sarker, and W. Chung. Leveraging dependency in scheduling and preemption for high throughput in data-parallel clusters. In *Proc. of IEEE CLUSTER*, Belfast, United Kingdom, 2018.

[25] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. of SOSP*, Farmington, 2013.

[26] K. Jagannathan, M. Markakis, E. Modiano, and J. N. Tsitsiklis. Queue-length asymptotics for generalized max-weight scheduling in the presence of heavy-tailed traffic. *IEEE/ACM Trans. Netw.*, 20(4):1096–1111, 2012.

[27] C. Wang, C. Feng, and J. Cheng. Distributed join-the-idle-queue for low latency cloud services. *IEEE/ACM Trans. Netw.*, 26(5):2309–2319, 2018.

[28] S. Agarwal, S. Kandula, N. Bruno, M. Wu, I. Stoica, and J. Zhou. Reoptimizing data-parallel computing. In *Proc. of NSDI*, San Jose, CA, 2012.

[29] A. Fergusin, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proc. of EuroSys*, Bern, 2012.

[30] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proc. of OSDI*, 2016.

[31] 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*, 2016.

[32] Palmetto cluster. https://www.palmetto.clemson.edu/palmetto/about/ [accessed in August 2023].

[33] Amazon ec2. https://aws.amazon.com/ec2/ [accessed in August 2023].

[34] L. Zheng, C. Joe-Wong, C. G. Brinton, C. W. Tan, S. Ha, and M. Chiang. On the viability of a cloud virtual service provider. In *Proc. of SIGMETRICS*, Antibes Juan-Les-Pins, 2016.

[35] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proc. of FAST*, 2012.

[36] A. L. Shimpi. The SSD anthology: Understanding SSDs and new drives from OCZ, 2014.

[37] J. Liu, H. Shen, and L. Chen. CORP: Cooperative opportunistic resource provisioning for short-lived jobs in cloud systems. In *Proc. of IEEE CLUSTER*, 2016.

[38] K. Karanasos, S. Rao, C. Douglas, K. Chaliparambil, G. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. of ATC*, 2015.

[39] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram. Looking beyond gpus for dnn scheduling on multi-tenant clusters. In *Proc. of OSDI*, 2022.

[40] A. Jajoo, Y. C. Hu, X. Lin, and N. Deng. A case for task sampling based learning for cluster job scheduling. In *Proc. of NSDI*, Renton, 2022.

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

[42] Y. Zhao, Y. Liu, Y. Peng, Y. Zhu, X. Liu, and X. Jin. Multi-resource interleaving for deep learning training. In *Proc. of SIGCOMM*, Amsterdam, 2022.