Improving Learning-Based DAG Scheduling by Inserting Deliberate Idle Slots

Yubin Duan and Jie Wu

Abstract

The increasing demands of computing capabilities make it expensive to operate a large-scale cloud cluster. A good scheduling algorithm should be able to reduce the average job completion time (JCT), which is the time duration between a job's arrival and its termination. However, when considering the precedence constraint of stages in each job, and when jobs arrive online, designing a scheduler to minimize the average JCT is challenging. Counterintuitively, we find that inserting idle time before some jobs might reduce the JCT, which is ignored by many schedulers. The state-of-the-art scheduler, which uses reinforcement learning (RL) techniques to solve scheduling problems, does not consider deliberate idle time. We integrate our observations to the RL agent and let the agent learn the best length of idle time. We carefully design the features used in RL. The shape of each job DAG is captured by the critical path length and the average width, and the detailed precedence constraints in each job DAG are extracted by graph neural networks. The experiment results on both synthetic and real-world datasets show that inserting the deliberate idle time could reduce the average JCT. Also, the results illustrate the significant contribution made by our proposed features.

Introduction

With the development of cloud computing, improving the efficiency of cloud clusters has become critical for cluster operators. The large demands of computing capabilities make it expensive to operate a large-scale cloud cluster such as Amazon Web Services and Google Cloud. For example, Alibaba's data clusters need to process more than 70 million transactions per second during peak hours. If we could improve the job processing efficiency of such large-scale data clusters, even by a small percentage, the operator could save a large amount of cost. The job processing efficiency of a cloud cluster heavily depends on the job scheduler, which decides the job processing sequence and the number of machines allocated to each job. Improving the scheduling algorithm has become the main concern.

A good scheduling algorithm should be able to reduce the average job completion time (JCT) [1, 2]. The completion time, or processing time, for each job is defined as the duration between the job's arrival and its completion. The average JCT is the total completion time divided by the number of jobs that have been processed. In this article, we consider a set of online arriving jobs. Each job is made of multiple stages, and these stages have precedence constraints (i.e., some stages cannot start unless the precedent stages are finished). The dependence relation among stages within a job is given by a directed acyclic graph (DAG). These jobs need to be scheduled on cloud clusters consisting of identical machines. The homogeneous setting means that a job could be processed by any machine in the cluster. Each machine's ability is limited and can only process one stage in a job at a time. In addition, we assume that each stage is non-preemptive, which means that when a stage starts being processed by a machine, it cannot be preempted. Our objective is to build an online job scheduler that could minimize the average job completion time for cloud clusters.

However, designing such a scheduler is challenging. The precedence constraint of stages in each job and the job's online arrival bring challenges for finding the optimal schedule. Recent research shows that reinforcement learning (RL) is a powerful tool for scheduling [3]. However, the existing RL-based scheduler does not consider the addition of deliberate idle time. We find that inserting deliberate idle time to some jobs could efficiently reduce the average JCT. Therefore, we propose to integrate this observation into an existing RL-based scheduler. Also, we carefully investigate the features that could be used in the RL. Namely, we note that the shape of each job DAG could be captured by the critical path length and the average width.

Inserting deliberate idle time might reduce the average JCT when the variance of job length is large. Figure 1 illustrates an example in which inserting idle time could reduce the JCT efficiently. Without inserting the idle time, as illustrated in Fig. 1a, job 1 would be processed when it arrives. Its completion time is 20 s. As for job 2, when it arrives, it has to wait for job 1 to finish, which leads to a waiting time of 18 s. Then it uses 4 s to finish. The job completion time of job 2 is 22 s. The average JCT of job 1 and job 2 is 21 s. In contrast, if we could insert the 2 s idle time slot to job 1, which means we force job 1 to wait for 2 s, job 2 could be processed first. Job 2's completion time becomes 4 s. The JCT of job 1 is 26 s. The average JCT could be reduced to (4 s + 26 s)/2 = 15 s. This is much smaller than the 21 s average JCT in the case when no idle time is inserted.

Designing a heuristic scheduling algorithm is challenging. We investigate the learning-based approach in which the scheduling policies are trained from the actual workload history. In par-

This research was supported in part by NSF grants CNS 2128378, CNS 2107014, CNS 1824440, CNS 1828363, CNS 1757533, CNS 1629746, and CNS 1651947.

The authors are with Temple University.

Digital Object Identifier: 10.1109/MNET.001.2100231

0890-8044/21/$25.00 © 2021 IEEE

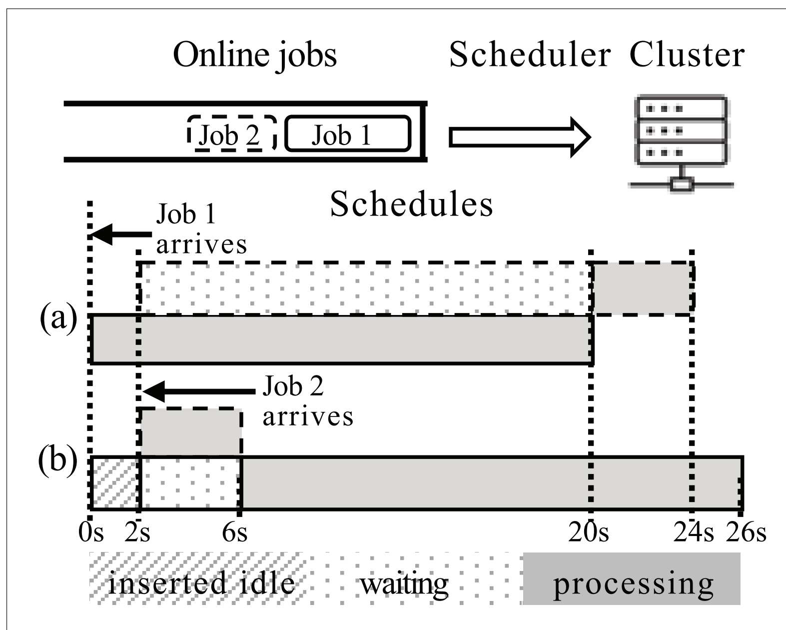

FIGURE 1. Job scheduling in cloud clusters.

ticular, we propose to integrate our observations into the RL agent, including inserting idle time slots. An advantage of learning-based schedulers is that they can adaptively adjust scheduling policies for different types of workload. However, concerns of learning-based schedulers arise in their interpretability, especially for RL. It is also challenging for a learning-based scheduler to provide a performance guarantee for the worst cases. We attempt to increase the interpretability of RL models using a perturbation-based approach. Specifically, besides training an RL agent based on our model, we also trained RL agents in which our proposed features are removed. Then we illustrate the contribution of our features by comparing the performance of these RL agents. The comparison shows that the features we design could help improve the performance of the RL agent.

In this work, we propose to minimize the average JCT of online DAG jobs in cloud clusters. We find that inserting deliberate idle time can reduce the average JCT. As it is challenging to build such scheduling because of the precedence constraint in general DAGs and the online arrival, we investigate the RL approach, which learns the length of idle time in its experience. Experiments on both synthetic and real-world datasets show that inserting deliberate idle time could reduce the average JCT; our features improve the efficiency of the scheduler.

Preliminaries

The job scheduling problem with the objective to minimize the average completion time is usually NP-hard. Scheduling for online arrival jobs is challenging [4]. Reference [5] has shown that there is no online algorithm that could achieve a bounded competitive ration. Besides the online arrival property, the precedence constraint in each job makes the scheduling more challenging.

The job scheduling problem has been investigated by many researchers [6, 7]. Theoretical analysis, such as [7], usually focuses on simple cases. Reference [7] gives the state-of-the-art theoretical result for minimizing the average completion time or the average flow time. However, they only consider the simple case where each stage has one sequential task. We consider a more general case where each stage could have sequential and/or parallel tasks. Optimally scheduling arbitrary structured DAGs on more than two machines is intractable [4]. Therefore, many researchers start looking for heuristics for practical scenarios. Reference [6] considers heterogeneous DAGs, which is more practical in real-world cluster traces, and proposes a heuristic for job scheduling and resource packing. They identify troublesome jobs in scheduling and packing, and propose to first deal with these troublesome jobs. Reference [8] discusses stage delay scheduling with the objective of minimizing the makespan. They did not consider adjusting the job processing order by inserting deliberate idle time.

Noticing that finding optimal schedules in polynomial time for general structured DAGs is intractable, an RL-based heuristic has been applied for scheduling [3]. Reference [3] introduces an RL-based DAG scheduler. Instead of using off-the-shelf RL techniques, they propose a batch training approach to deal with the online arrival of jobs. The existing RL-based scheduler would process a stage once an executor becomes available. However, our observation shows that inserting idle time for some stages would improve the performance of schedulers in terms of the average JCT.

Improving the interpretability of machine learning techniques has attracted more attention recently. Reference [9] summarizes the techniques to interpret machine learning models. Those techniques are grouped into two categories: intrinsic and post hoc interpretability. Intrinsic interpretability refers to self-explanatory models such as decision tree models and attention models. Post hoc interpretability needs to use another model to explain an existing model.

Models

Problem Formulation

Assume we have k identical executors and n jobs. Let J denote the set of jobs, and $J = \{j_1, j_2, ..., j_n\}$. Each job j_i consists of multiple stages with dependency relationships. The arrival time of job j_i is a_i, and its finish time is b_i. Then the completion time of a job is $c_i = b_i - a_i$. Note that the completion time includes both waiting time and the processing time of the job. We focus on the online arrival of jobs. Formally, the arrival time a_i of each job j_i is a stochastic variable, and its value is unknown to the scheduler before j_i's arrival. However, we assume job DAGs are well annotated; that is, the precedence relations and stage lengths are immediately known by the scheduler when a job arrives. At any time, each executor can process only one job stage. Once a job stage is assigned to an executor, it must be processed without preemption until the stage terminates.

We use a DAG to model the stage dependency relationships for each job. Specifically, we use S_i to denote the set of stages in job j_i. $S_i = \{s_{i1}, s_{i2}, ..., s_{im}\}$, where m is the number of stages in job j_i. Each stage s_{ij} has a length l_{ij} representing its processing time on one executor. Allocating more executors to a stage could reduce its processing time. However, the speedup is nonlinear. There

are precedence constraints for stages in S_i. The partial order $s_{ix} \prec s_{iy}$ means that stage s_{iy} cannot be processed unless stage s_{ix} has finished. A DAG is used to represent the partial orders of stages in a job, where directed edges in the DAG show the precedence relationship between stages.

We aim to minimize the average JCT. Specifically, the average JCT is defined as $\Sigma_{ji \in J} c_i / |J|$, where $\Sigma_{ji \in J} c_i$ is the overall job completion time and $|J|$ is the size of the job set. The job terminates only if all stages in the job are processed. Scheduling offline DAGs with general structures on an arbitrary number of executors is NP-complete [10]. Since any offline instance could be reduced to an online instance, online scheduling is also NP-complete.

We follow the list scheduling approach: a DAG is flattened into an ordered list consisting of all stages. For job j_i, the scheduler needs to choose a processing order O_i from feasible topology sorts of stages in j_i. In addition, for each stage $s_{ij} \in S_i \in j_i$, the scheduler needs to decide the number of executors p_{ij} assigned to it and the length of the deliberate idle time d_{ij} inserted before it.

Dynamic View on Stochastic Scheduling

Inserting the deliberate idle time cannot bring any benefit for the offline scheduling, but might be useful in the online scenario. In the offline scheduling, the scheduler could avoid the case in which a shorter job waits for a longer job to finish by simple greedy heuristics, such as short-job-first scheduling. These heuristics cannot be applied in online scheduling since the numbers and lengths of unprocessed jobs are stochastic. If the scheduler assigns all executors to arrived jobs, a newly arrived job has to wait even if it is extremely small. This introduces an unreasonably long waiting time for those relatively small jobs. Inserting deliberate idle time could help the scheduler avoid this case by letting the scheduler keep a small number of available executors for future jobs. It could reduce the average JCT, especially when the scheduler can predict the size of future jobs in the rough.

To deal with the online scheduling problem, we use an RL framework, as shown in Fig. 2, to dynamically update job schedules. The RL agent would observe the number of available executors and the remaining job set as its input state and generate the job schedule (O_i, p_{ij}, d_{ij}) as actions. Recall that the list scheduling assigns a priority for each job stage and selects a stage to execute based on priority. The dynamic list scheduling means that the priority p_{ij} of a job stage could be adjusted only if the stage has not been processed. Furthermore, our scheduler also dynamically adjusts the O_i and d_{ij} when determining p_{ij}. The values of (O_i, p_{ij}, d_{ij}) are determined by an RL agent. We consider building our scheduler based on RL for two major reasons. One is that the workloads in data center clusters can be predicted. Existing research, such as [11, 12], have shown several machine learning techniques for workload prediction. This shows that there are some patterns in cluster workloads that can be learned and used by RL agents. The other reason is that RL techniques are usually used to solve sequential decision making problems, which fits the list scheduling approach. Starting from a random policy, the RL agent would explore possible actions and improve its policy using the reward values of environmental feedback. Also, the RL agent could dynamically adjust the scheduling when a new job arrives or some executors become available.

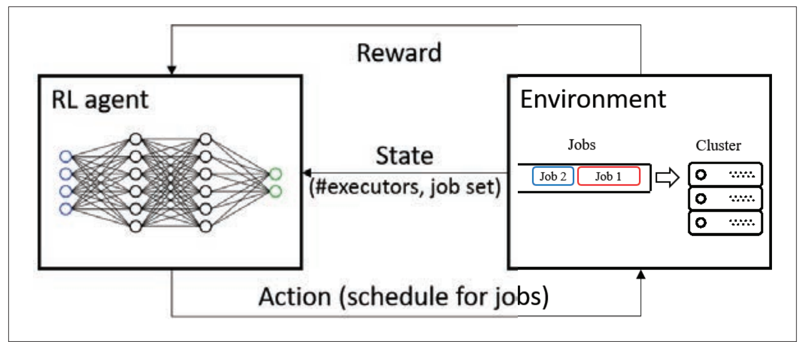

FIGURE 2. Reinforcement learning structure.

Idle-Aware Job Scheduler

For dynamic list scheduling, we need to determine the frequency of updating the schedule for existing jobs. It is critical to select an appropriate location for idle time insertion. Should it be inserted before each job (job-level insertion) or before each stage (stage-level insertion)? All of those factors should be considered when designing an RL agent for scheduling. We first investigate the impact of those factors, including the relationship between the speedup of each job and the number of executors assigned, the frequency of updating the scheduling list, and the comparison between job-level insertion and stage-level insertion. Then we introduce our RL-based job scheduler. Finally, we investigate the detailed structures of the RL agent.

The speedup of the execution of a job positively correlates with the number of executors assigned to it, while it is nonlinear and difficult to model. Amdahl's law [13] gives a formula for the theoretical speedup. Specifically, a program is divided into a serial part and a parallel part. The execution time of the serial part is fixed, and that of the parallel part is inversely proportional to the number of executors. However, it is hard to determine the percentage of the serial part for each job. Also, this percentage varies for different types of jobs. Therefore, instead of formulating the speedup, we use machine learning techniques to find the proper number of executors for different types of jobs.

In the dynamic list scheduling approach, schedules can be updated during the processing of these jobs. Instead of setting a fixed interval between two updates, we choose to trigger the update based on certain events. Specifically, we propose to update the schedules when i) a new job DAG arrives, ii) a job is finished and the executors assigned to it become available, or iii) the deliberate idle slot reaches its end. Updating the schedules besides the occurrence of those events is not very helpful since we consider the non-preemptive scheduling. The executors assigned to a job cannot be retrieved until the job is finished. Therefore, we choose to update the schedules at trigger events.

To improve the performance of scheduling, we also need to carefully determine the granularity of the idle time insertion. Inserting idle time before each job (job-level insertion) reduces the searching space, but also might miss the optimal solutions. It

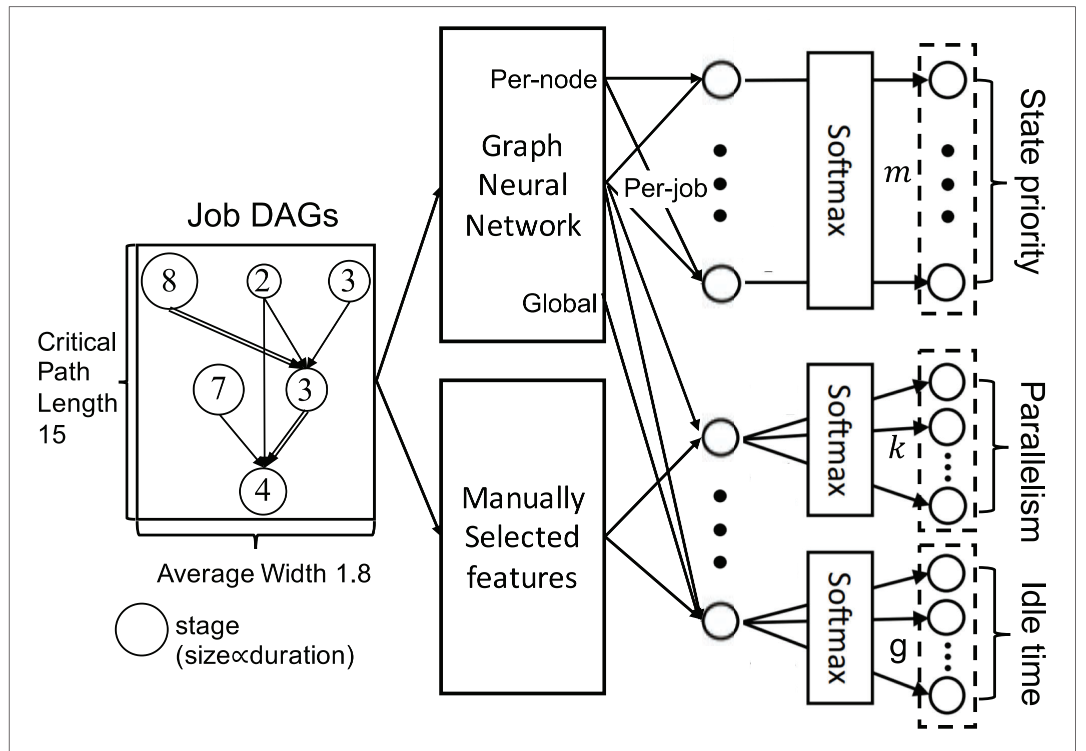

FIGURE 3. The policy network structure.

is not necessary to insert idle time at each stage when a DAG arrives since some stages cannot be executed until their predecessors are all finished. In this article, we propose to mix these two insertion methods. Specifically, we repeatedly apply job-level insertion on remaining job DAGs, where the remaining job DAGs consist of all stages that have not been processed. When a stage is completed or a new job DAG arrives, we re-evaluate the length of idle time for remaining job DAGs. To avoid the starvation of jobs (e.g. always insert an idle slot before the same stage), we restrict each stage to only being delayed at most once. The scheduler would maintain a table to record whether a stage has been delayed or not.

The scheduler needs to assign a priority to all stages in each job, and selects a stage to execute based on the priority when there are available executors. There are multiple heuristics to determine the priority of each stage, such as based on the critical path or based on the node degrees [14]. However, none of these heuristics works perfectly for general DAG scheduling with an arbitrary number of executors. Each heuristic is designed to label a special DAG structure, and cannot adaptively adjust the stage priority during processing. We rely on the RL approach to find a proper policy to label the stage priority for general structured DAGs. The RL agent could learn the proper number of executors and the length of idle slots for each job based on its experience.

At each trigger event, the RL agent uses the number of available executors, the annotated job DAG set (including flags indicating whether stages have been delayed), and the numbers of executors currently assigned to each job as the state. It would generate an action that consists of the next job stage to be executed (s_{ij}), the maximum number of executors to be assigned to the selected job (p_i), and the length of the deliberate idle slot to be inserted (d_{ij}). Based on the action, the scheduler would set a timer for stage s_{ij} with length d_{ij}, and label s_{ij} as a delayed stage. When the timer is up, the scheduler would assign available executors to the selected stage s_{ij} such that the total number of executors assigned to j_i becomes p_i. If the number of available executors is not large enough, the scheduler would assign all available executors to j_i.

Note that the RL agent limits the maximum number of executors assigned to the job, j_i, instead of the stage, s_{ij}. Furthermore, to meet the requirement of the non-preemptive scheduling, the value of p_i is non-decreasing.

We use the accumulated JCT as the reward parameter. Assume two adjacent actions are generated at times t_k and t_{k+1}, and the number of jobs during $[t_k, t_{k+1})$ is u_k. The reward is $-(t_{k+1} - t_k)u_k$. Maximizing the reward is equivalent to minimizing the overall JCT $\sum_{j_i \in J} c_i$ or the average JCT $\sum_{j_i \in J} c_i / |J|$.

The structure of the RL agent is adapted from [3]. The policy network of the RL agent is shown in Fig. 3. The features for training contain two parts. The first part is the features embedded by the graph convolutional neural network, and the second is the feature we manually selected.

The sizes (numbers of stages and stage lengths) of job DAGs vary, which brings challenges for encoding the DAGs. We need to encode the DAG with different sizes into a vector with a fixed size. Reference [3] proposed to overcome the challenge by using a graph neural network (GNN). The GNN could encode the job DAGs into a set of fixed-length vectors. There are three types of vectors: per-node embedding vectors, pre-job embedding vectors, and global embedding vectors. The per-node embedding vectors are used to determine the priority of each stage. The per-job

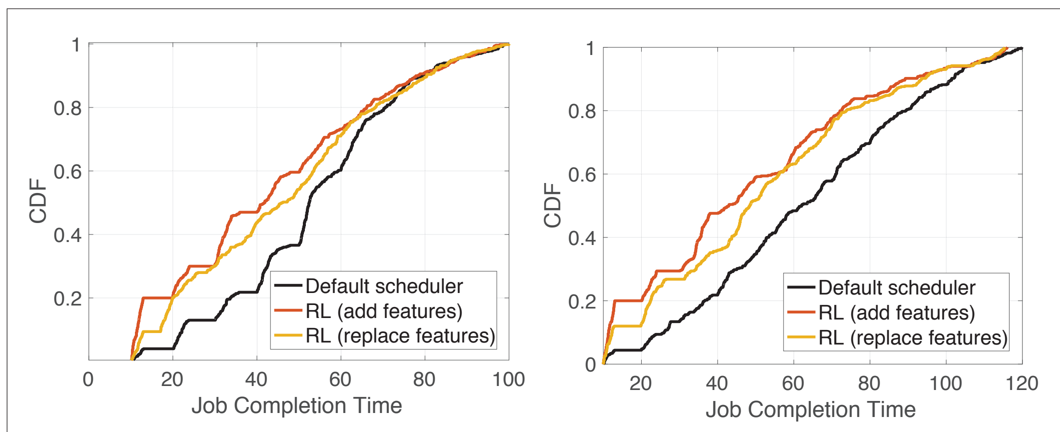

FIGURE 4. The CDF of job completion time: a) workload = 60 percent; b) workload = 80 percent.

embedding vectors are the job-level encoding of DAG and are used to determine the priority of each stage, the parallelism of each job, and the idle time length. The global embedding vectors accumulate the encoding for multiple DAGs and are mainly used to determine the idle time length.

In addition to using the GNN, we propose to add some manually selected features to the feature vector. The manually selected features include the critical path length and the average width of a job, as shown in Fig. 3. The double lines in the figure indicate the critical path of the job, which is the longest path from the root to a leaf stage. The path length is the total duration of the stages that belong to the path. The length of the critical path shows the lower bound of the job processing time no matter how many executors are allocated to the job. The average width is the summation of the stage lengths divided by the critical path length. The average width of the job shown in Fig. 3 is 27/15 = 1.8. This value gives a clue for determining the proper number of executors that should be assigned to a job. By adding those features, we hope to manually force the RL algorithm to notice useful factors in scheduling. The manually selected features are easy to compute when the job DAG arrives. The length of the critical path and the average width are calculated by one round of traversal on the input DAG. During traversal, a variable is used to maintain the accumulated length of each stage, and another variable is used to record the depth of traversal. After the traversal, the length of the critical path is determined by the maximum depth. The average width could be determined by dividing the summation of stage lengths by the critical path length. Assembling the vector of manually selected features along with the feature vectors generated by the GNN, we get the feature space that could be used to train the policy network.

The output layer of the policy network consists of three types of neurons. The first type contains m neurons whose outputs represent the probability that the stage is selected. The second type contains k neurons, where k is the total number of executors in the cluster. Each neuron represents a parallelism level. The output of each neuron represents the probability that the parallelism level is selected. The third type of neuron is used to determine the length of idle slots for the selected stage. Theoretically, the idle time $d_i \in \mathbb{R}$, and the amount of its possible values is infinity. To reduce the size of the action space, we discretize the possible values of d_i. For a stage with length l_{ij}, we divide l_{ij} into g pieces. The length d_i must be $r \cdot l_{ij}/g$, where $r = 0, 1, 2, \cdots, g$. Therefore, the third type contains g neurons, and each represents a possible value of r. The output of each neuron is the probability that the corresponding r is selected. The policy gradient algorithm [15] is used to train the RL agent. The idea of the policy gradient is to perform gradient descent on the policy network based on the rewards observed during training.

Experiment

Experiment Setting

We test our scheduler on both synthetic and real-world datasets. The synthetic dataset contains two types of jobs, long-term and short-term. First, we fix the length of the long-term jobs at 50 s, and the length of the short-term jobs at 10 s. We simulate the Poisson process for job arrivals. Specifically, the interarrival time of synthetic jobs obeys independent and identically distributed (IID) exponential distribution. The real-world dataset is extracted from TPC-H[1] queries. We randomly sample 10^3 jobs input sizes varying from 1 GB to 100 GB. Besides the synthetic and real-world datasets, we also use a mixed dataset in the experiment. The mixed dataset consists of jobs randomly sampled from the synthetic and real-world datasets. The percentage of synthetic data samples is controlled by $\alpha \in [0, 1]$. $\alpha = 1$ means that all of the data in the mixed dataset is synthetic data. In addition, to reduce the time consumption of training, we use a simulator to calculate the reward for the policy network.

Experiment Results

We examine the importance of the manually selected features: critical path length and average width. We compare the performance of the RL agent that has manually selected features with the agent in which those features are removed. This experiment is conducted on the synthetic dataset and the mixed dataset ($\alpha = 0.5$). Both RL agents are trained in the same device and have the same length of training time. The comparison result is shown in Table 1. In Table 1, we also record the average JCT of the scheduler that does not consider deliberate idle insertion. From the table, we find that removing the manually selected features would harm the

[1] http://www.tpc.org/tpch/

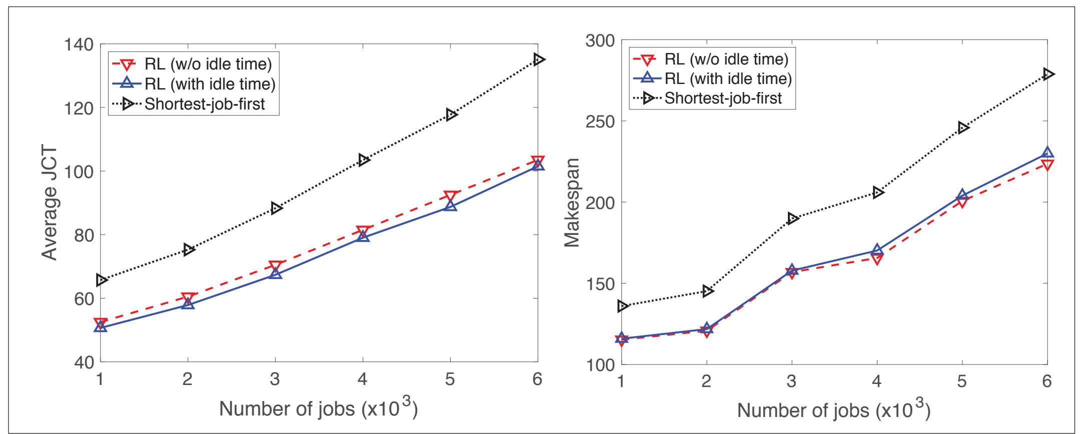

FIGURE 5. Comparison under different number of jobs: a) comparison of average JCT; b) comparison of makespan.

	Average JCT	Average JCT w/o selected features	Average JCT w/o idle time
Synthetic	46.3	52.7	53.5
Mixed	69.4	75.2	74.5

TABLE 1. Average JCT over synthetic and mixed dataset.

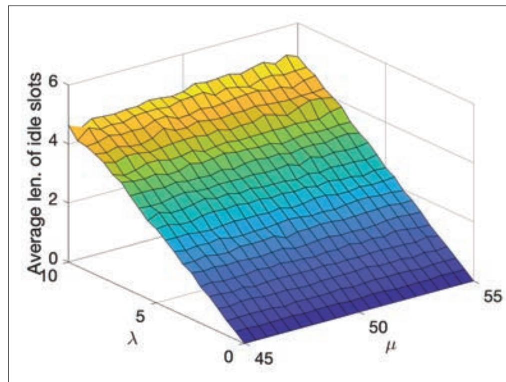

FIGURE 6. The inserted idle time on the synthetic dataset.

performance of the RL agent. The average JCT increases by about 13.8 percent on the synthetic dataset and increases by about 8.4 percent on the mixed dataset. The experiment results show that the features we selected are useful and could help to improve the performance of the RL agent.

We then take a closer look at the distribution of JCT. Figure 4 shows the cumulative distribution of JCT generated by a different scheduler. The default scheduler refers to the first-in first-out (FIFO) scheduler. Adding features means our manually selected features are added to the RL agent introduced in [3]. Replacing features uses manually selected features to randomly replace features used by the RL agent [3] and keeps the total number of features unchanged. In the CDF plot, a line on the left indicates a better scheduling policy. Comparing Figs. 4a and 4b, we can see that a larger workload is more likely to lead to a larger JCT. This is more obvious for the scheduler without inserting idle time. We can see that the difference between the CDF plot of the scheduler with idle time and that of the scheduler without idle time becomes larger when the workload increases from 60 to 80 percent. This shows that inserting deliberate idle time is more important if the workload is heavier. The number of overlaps between different jobs increases with the workload. With more overlaps, the number of cases also increases when the shorter jobs need to wait for larger jobs.

We also investigate the performance of our scheduler under different workloads. Specifically, the number of available workers in the cluster is fixed, and we change the number of jobs to simulate different workloads. The experiment results are shown in Fig. 5a. In the figure, we use the shortest-job-first algorithm as a baseline and compare the RL-based scheduler with and without inserting idle slots. From the figure, we find that the RL-based schedulers can significantly reduce the average JCT compared with shortest-job-first, which is a heuristic approach. The improvement is around 30 percent. In addition, we can observe around a 3.0–5.6 percent reduction of the average JCT if the deliberate idle slot is considered.

Additionally, we focus on the trade-off between the average JCT and makespan. Inserting deliberate idle time might reduce the average JCT but also might lead to a larger makespan. We show this trade-off in Fig. 5b. The figure shows the comparison over the makespan. Because we insert idle time for jobs, the makespan becomes larger. The increase is not large. However, considering that minimizing the makespan is not our objective, the small increase in the makespan does not affect the performance of our scheduler. If we modify the reward function of the RL agent, we would be able to reduce the makespan of the overall jobs. However, it is not the objective of our scheduler. In addition, compared to shortest-job-first, the RL agent that considers inserting deliberate idle slots can reduce the makespan by around 17–21 percent. This is because the RL-based scheduler can adaptively adjust the priority of different stages to improve the resource utilization when minimizing the average JCT.

Figure 6 shows the average length of the deliberate idle time on the synthetic dataset with different random parameter values. In this set of experiments, the longer job in the synthetic data set no longer has a fixed length. Instead, we sample its length from a normal distribution $\mathcal{N}(\mu, \sigma)$, where μ is the mean and σ is the standard deviation. With different μ, the value of σ is always set to $\mu/6$. The length of the shorter job in the synthetic dataset is kept at 10. λ is the parameter of

the Poisson process, and it controls the job arrival rate. From the figure, we can see that if the interval between two adjacent jobs becomes smaller or the average length difference between longer and shorter jobs becomes smaller, the average length of the inserted idle time is adaptively reduced. This shows that the RL agent could adjust its policy on different inputs. Compared to some fixed heuristic rules such as shortest-job-first, the RL agent could be applied to more general cases.

Conclusion

We focus on efficient scheduling for online arrival jobs with general DAG structures. The objective of our scheduler is to minimize the average job completion time (JCT). We adapt a reinforcement learning (RL) approach and integrate our observations that inserting deliberate idle time for relatively large jobs could reduce the average JCT. We follow a dynamic list scheduling approach to carefully design features used to train the policy network in the RL framework. The shape of each job DAG is captured by the critical path length and the average width, while the detailed precedence constraints in each job DAG are extracted by graph neural networks. We implement the scheduler with the deliberate idle time on both synthetic and real-world datasets. The experiment results show the efficiency of the addition of deliberate idle time. In addition, our perturbation-based method shows that the features, critical path length, and average width proposed in the article make large contributions to the RL agent. These features should improve the performance of the RL agent in terms of the average JCT.

Extending the RL-based scheduler for other objectives, such as minimizing makespan or providing fairness guarantees, would be interesting future works. We can adjust the reward function used by RL agents and change their preferences when assigning priorities. A well-designed reward function may be able to keep fairness while optimizing the makespan or average JCT. In addition, we can explore the performance of RL schedulers in different cluster scales. Additional efforts might be needed to train the RL agent for large-scale clusters that have an extremely large number of executors. Furthermore, how to improve the interpretability of RL techniques remains an open problem. In addition to adding manually selected features, developing attention mechanisms is an appealing approach for demystifying the decision making process of deep RL models.

References

[1] A. D. Ferguson *et al.*, "Jockey: Guaranteed Job Latency in Data Parallel Clusters," *ACM EuroSys*, 2012, pp. 99–112.

[2] J. Wu, *Distributed System Design*, CRC Press, 1998.

[3] H. Mao *et al.*, "Learning Scheduling Algorithms for Data Processing Clusters," *Proc. ACM SIG on Data Commun.*, 2019, pp. 270–88.

[4] A. Marchetti-Spaccamela *et al.*, "On the Complexity of Conditional Dag Scheduling in Multiprocessor Systems," *2020 IEEE Int'l. Parallel and Distributed Processing Symp.*, 2020, pp. 1061–70.

[5] N. Garg and A. Kumar, "Minimizing Average Flow-Time: Upper and Lower Bounds," *IEEE FOCS*, 2007, pp. 603–13.

[6] R. Grandl *et al.* "GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters," *12th USENIX Symp. Operating Systems Design and Implementation*, 2016, pp. 81–97.

[7] K. Agrawal *et al.*, "Scheduling Parallel Dag Jobs Online to Minimize Average Flow Time," *Proc. 27th Annual ACM-SIAM Symp. Discrete Algorithms*, 2016, pp. 176–89.

[8] W. Shao *et al.*, "Stage Delay Scheduling: Speeding Up Dag-Style Data Analytics Jobs with Resource Interleaving," *ICPP*, 2019, pp. 1–11.

[9] M. Du, N. Liu, and X. Hu, "Techniques for Interpretable Machine Learning," *Commun. ACM*, vol. 63, no. 1, 2019, pp. 68–77.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman San Francisco, vol. 174, 1979.

[11] S. Di, D. Kondo, and W. Cirne, "Host Load Prediction in a Google Compute Cloud with a Bayesian Model," *Proc. IEEE SC*, 2012, pp. 1–11.

[12] M. Ghorbani *et al.*, "Prediction and Control of Bursty Cloud Workloads: A Fractal Framework," *Proc. 2014 Int'l. Conf. Hardware/Software Codesign and System Synthesis*, 2014, pp. 1–9.

[13] T. G. Robertazzi and L. Shi, "Amdahl's and Other Laws," *Networking and Computation*, Springer, 2020, pp. 139–49.

[14] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, 1999, pp. 406–71.

[15] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, 2018.

Biographies

Yubin Duan (yubin.duan@temple.edu) received his B.S. degree in mathematics and physics from the University of Electronic Science and Technology of China, Chengdu, in 2017. He is currently a Ph.D. student in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania. His current research focuses on distributed systems.

Jie Wu [F] (jiewu@temple.edu) is the director of the Center for Networked Computing and Laura H. Carnell Professor at Temple University. His current research interests include mobile computing and wireless networks, network security, and cloud computing. He publishes in scholarly journals, conferences, and books. He is a Fellow of the AAAS.