EdgeTuner

Fast Scheduling Algorithm Tuning for Dynamic Edge-Cloud Workloads and Resources

Han, Rui; Wen, Shilin; Liu, Chi Harold; Yuan, Ye; Wang, Guoren; Chen, Lydia Y.

DOI
10.1109/INFOCOM48880.2022.9796792

Publication date
2022

Document Version
Final published version

Published in
INFOCOM 2022 - IEEE Conference on Computer Communications

EdgeTuner: Fast Scheduling Algorithm Tuning for Dynamic Edge-Cloud Workloads and Resources

Rui Han, Shilin Wen, Chi Harold Liu, Ye Yuan, Guoren Wang
Beijing Institute of Technology, Beijing, China
Beijing, China
{hanrui, 3120185530, chiliu, yuan-ye, wanggr}@bit.edu.cn

Lydia Y. Chen
TU Delft
Delft, Netherlands
{lydiaychen@ieee.org

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796792

***Abstract*—Edge-cloud jobs are rapidly prevailing in many application domains, posing the challenge of using both resource-strenuous edge devices and elastic cloud resources. Efficient resource allocation on such jobs via scheduling algorithms is essential to guarantee their performance, e.g. latency. Deep reinforcement learning (DRL) is increasingly adopted to make scheduling decisions but faces the conundrum of achieving high rewards at a low training overhead. It is unknown if such a DRL can be applied to timely tune the scheduling algorithms that are adopted in response to fast changing workloads and resources. In this paper, we propose EdgeTuner to effectively leverage DRL to select scheduling algorithms online for edge-cloud jobs. The enabling features of EdgeTuner are sophisticated DRL model that captures complex dynamics of Edge-Cloud jobs/tasks and an effective simulator to emulate the response times of short-running jobs in accordance to dynamically changing scheduling algorithms. EdgeTuner trains DRL agents offline by directly interacting with the simulator. We implement EdgeTuner on Kubernetes scheduler and extensively evaluate it on Kubernetes cluster testbed driven by the production traces. Our results show that EdgeTuner outperforms prevailing scheduling algorithms by achieving significant lower job response time while accelerating DRL training speed by more than 180x.**

***Index Terms*—Edge-cloud workloads; scheduling algorithm; DRL; run-time tuning; Kubernetes**

I. Introduction

Due to recent development of Internet of Things (IoT) and edge computing technology, traditional cloud-based approaches face performance issues due to their high transmission latency and expensive bandwidth cost. At the same time, edge devices have limited or even insufficient resources to execute expensive vision and machine learning tasks [9], [11], [19], [25]. The jobs that process some tasks on both cloud and some on edge devices thus fast increase and edge-cloud workloads becomes very critical for various applications. Representative workloads include *cloud-edge workloads* (e.g. smart home applications [18], [39]), *edge-cloud workloads* (e.g. smart healthcare [22], anomaly detection [38], and object recognition [16]), and *edge-cloud-edge workloads* (e.g. autonomous driving services [5] and intelligent photo management [32]). To process jobs in such workloads, an edge device collaborates with cloud nodes [12], [33], [37] and thus the jobs' performances are considerably affected by the resource allocation among them.

Example. Figure 1 illustrates an example of allocating three tasks (from two jobs) to two cloud nodes using a Kubernetes scheduler. The job performance is influenced by three factors: (1) *Workload dynamicity*. Jobs arrive continuously and their tasks demand different amounts of resources (Figure 1(a)); (2) *Resource dynamicity*. The cloud resource available to an edge device dynamically changes (Figure 1(b)); and (3) *scheduling algorithm*. When applying different scheduling algorithms, e.g. Balanced Resource Allocation (BRA), Most Requested Priority (MRP), and Least Requested Priority (LRP) [4] in Kubernetes, jobs have considerably different latencies. In Figure 1(c)'s example, BRA achieves the lowest latency because its scheduling mechanism uses the resources most efficiently for this specific scenario.

Hence, the cluster's performance largely depends on the configurations of scheduling algorithms that respond to workload and resource dynamicity. Deep reinforcement learning (DRL) is a widely used technique that learns optimal resource allocation online as the system runs [17], [20], [28]. However, applying this technique to tune scheduling algorithms requires a dauntingly large number of data samples for training DRL agents. Moreover, cloud-edge jobs tend to be short running (over 90% of job durations range from dozens of seconds to a few minutes). When constructing time-variant states (randomly arrival jobs, various resource demands of tasks, and elastic node resources) in a DRL environment, a training sample needs rather *long time* (e.g. 10 seconds) to generate. This is because it needs to differentiate the states of two consecutive time-steps and at least a few seconds are needed before some workload or resource changes happen. A DRL training may need millions of samples to converge and hence the training process is bottlenecked by the **time-consuming sampling phase** (may take dozens of hours). *At run-time, an outdated DRL agent may lead to significant deviations from the optimal scheduling policy and incur job performance degradation*. Some recent techniques develop simulation platforms to support the DRL training in an offline fashion [24], [35], [36]. Most of them share the limitation that their designs target for *long-running jobs* in performance computing (HPC) data centers, and implicitly assume *a fixed scheduling algorithm* and *pre-defined available* resources.

In this paper, we propose EdgeTuner, an online approach that effectively uses DRL agents to select scheduling algorithms for edge-cloud jobs. To overcome the daunting training overhead, we develop a cluster simulator that emulates the

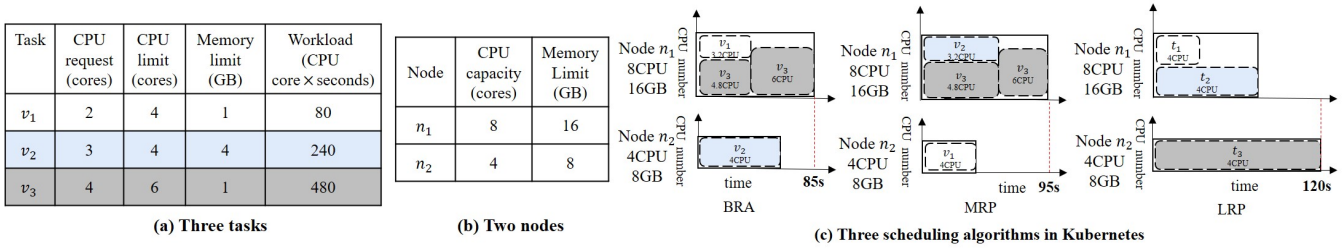

Fig. 1. An example of job scheduling using three Kubernetes scheduling algorithms

volatile and complex state space of edge-cloud jobs, their resource demands, and cluster resource utilization. The simulator effectively captures the on-line adaption across different scheduling algorithms as well as the dynamicity of short-running jobs. As such, the training of DRL agent can be effectively and quickly converged via this offline simulation phase. Note that EdgeTuner differs from traditional hyper-heuristic approaches that find an optimal scheduling algorithm for pre-specified cloud workflow [34] or batch jobs [27], [29]. In contrast, EdgeTuner needs no prior knowledge about the jobs to be scheduled and provides fast scheduling algorithm tuning for continuously arrival jobs in the cluster.

In particular, the contributions of this paper are as follows:

▷ **Complex Edge-Cloud job scheduling modelling**. We formulate the tuning of scheduling algorithms for edge-cloud jobs as a sequential decision making process (MDP) to leverage the DRL technique. To incorporate various scheduling scenarios, we define general *state* representation of complex nodes and workloads, use *action* to reflect optional scheduling algorithms, and define *reward* function to estimate job performance.

▷ **DRL training acceleration**. We develop a cluster simulator that emulates a scheduling algorithm's resource allocation mechanism and its influence factors (available resources, and waiting and running tasks). At each scheduling interval, the simulator takes the agent's state and action as inputs and outputs the reward instantly (this process takes at least a few seconds in real clusters). The training can be performed offline by directly interacting with the simulator in the usually adopted online learning scheme.

▷ **Implementation and evaluation**. We incorporate our controller on the popular Kubernetes scheduler and evaluate it using workloads driven from the Alibaba cluster trace [2]. We evaluate our approach in real clusters and the extensive evaluations against 11 Kubernetes scheduling algorithms show: (i) by applying DRL agents on edge devices, our approach outperforms representative scheduling algorithms by achieving 15.78% reductions in job latencies; (ii) our approach accelerates the DRL training speeds by more than 180x.

The remainder of this paper is organized as follows: Section II formulates the problem, Section III explains our approach. and Section IV evaluates it. Section V introduces the related work, and finally, Sections VI summarizes the work.

II. Problem Formulation

We now formalize the tuning for scheduling algorithms of edge-cloud workloads as a RL problem. An edge-cloud *job* consists of multiple *tasks*, some of which are executed on edge devices and the rest of tasks are co-executed on cloud nodes with other jobs. For the resource demands, we consider CPU and memory. A *scheduling algorithm* decides how to best co-locate the execution of tasks in the cloud in accordance to their CPU and memory resources such that the response times of jobs are minimized. Here, we specifically consider scenarios that scheduling algorithms can be dynamically chosen during the system runtime. We model the tuning of scheduling algorithm as RL problem. in which an *agent* (tuner) learns to *act* (selecting a scheduling algorithm) in an *environment* (cluster) in order to maximize a scalar reward signal [26]. At each discrete time-step (episode) t = 0, 1, 2..., the cluster provides the tuner with an observation s_t, the tuner responds by selecting a scheduling algorithm a_t and obtains the feedback of reward $R(s_t, a_t)$ and next state s_{t+1} from the environment. This interaction is formalized within the framework of markov decision process (MDP), which is a controlled stochastic process defined by the state space $\mathcal{S}$, action space $\mathcal{A}$, transition dynamics $0 \leq P(s_{t+1}|s_t, a_t) \leq 1$, and reward function $R(s, a)$.

State. The modeling of state considers two factors that determine jobs' running performances: the resource utilizations in nodes and the waiting and running tasks in the cluster. Similar to previous cluster scheduling, a job corresponds to a directed acyclic graph (DAG) of tasks. In scheduling, waiting tasks are the targets that a scheduling algorithm needs to manage; running tasks occupy resources and then release them after completion. Formally, s_t=(N, V^w, V^r) denotes the node and task information during a scheduling interval.

- A node $n \in N$ is denoted as a 6-tuple $(cpu^u, cpu^r, cpu^c, mem^u, mem^r, mem^c)$: (1) cpu^u represents the actual usage of CPU cores; (2) cpu^r represents the $requested$ (reserved by tasks) CPU cores; (3) cpu^c represents the capacity of CPU cores in the node; (4) mem^u represents the actual memory usage; (5) mem^r represents the requested memory by tasks; and (6) mem^c represents the node's memory capacity.
- A waiting task $v^w \in V^w$, it is denoted as a 7-tuple $(cpu^r, cpu^l, mem^r, mem^l, work, p, j^{id})$: (1) cpu^r represents the requested CPU by the task; (2) cpu^l represents

the CPU limit of the task; (3) mem^r represents the requested memory by the task; (4)mem^l represents the task's memory limit; (5) $work$ represents the workload of task, e.g. workload 200 means the task needs 100 seconds to complete when running in 2 CPU cores; (6) p represents the task's platform type (cloud or edge); and (7) j^{id} is the job ID the task belongs to.

- A running task $v^r \in V^r$ is denoted as a 5-tuple $(work, node, cpu^l, et, j^{id})$: (1) $work$ represents the task's workload (that is, number of CPU cores multiplied by execution time); (2) $node$ represents the node the task is allocated; (3)cpu^l represents the task's CPU limit; (4) et represents the elapsed time when the task starts running; and (5) j^{id} is its job ID.

Dimensionality of state s_t. We note that in practical job scheduling, $|N|$ is the number of nodes in the edge-cloud cluster, but the numbers of waiting and running tasks continuously change at different time steps. We hence set these numbers as sufficiently large values (e.g. 6 or 10 that is larger than the total number of tasks). Given that the scheduling interval is short (e.g. 10 seconds), such values do not increase the training complexity. We note that the state space grows significantly with the number of job/task arrivals and the size of cluster.

Action. Given a scheduler, action a_t represents one possible scheduling algorithm an agent can select. For example, when implementing DRL in Kubernetes, three typical scheduling algorithms are: (1) BRA: this algorithm balances the utilization of CPU and memory resources in different nodes. (2) LRP: this algorithm calculates the amount of resources and the number of tasks allocated to different nodes, and prefers to allocate tasks to nodes with more available resources. (3) MRP: this algorithm prefers to allocate tasks to nodes with less available resources, thus running the same tasks with the least number of nodes.

Transition dynamics. In a MDP, transition dynamics $P(s_{t+1}|s_t, a_t)$ reflects the time-variant dynamics of cluster. Such dynamics are determined by three factors: the tasks $V_t^{allocate}$ that obtain resource allocations; the completed tasks $V_t^{complete}$ at time-step t; and the newly arrival jobs/tasks V_{t+1}^{arrive} at time-step t+1. We note that both $V_t^{allocate}$ and $V_t^{complete}$ are influenced by the scheduler algorithm set by action a_t, and they determine the three elements in state s_{t+1} at time step t+1:

$$V_{t+1}^w = V_t^w \backslash V_t^{allocate} \cup V_{t+1}^{arrive} \tag{1}$$

$$V_{t+1}^r = V_t^r \cup V_t^{allocate} \backslash V_t^{complete} \tag{2}$$

Reward function. At a time-step t, a reward r_t denotes the job latency (or job completion time (JCT) [8]) when using scheduling algorithm a_t. Given that there are only a small number of jobs during a scheduling interval, we consider completion times of both jobs and their tasks to accelerate the convergency of RL training. Specifically, let J be the set of jobs completed within period $(t-1, t]$ and JCT_i be the completion time of a job $j_i \in J$. The reward of job j_i is

$$r_i^{job} = \alpha_1 * JCT_i + \beta_1 \tag{3}$$

Similarly, let V^c be the set of tasks completed within period $(t-1, t]$ and TCT_i be the completion time of a task $v_i^c \in V^c$. The reward of task v_i^c is

$$r_i^{task} = \alpha_2 * TCT_i + \beta_2 \tag{4}$$

The reward r_t of time-step t is the summation of jobs and tasks' rewards.

$$r_t = \sum_{i=1}^{|J|} r_i^{job} + \sum_{i=1}^{|V^c|} r_i^{task} \tag{5}$$

In RL training, we set negative values of α_1 and α_2, and positive values of β_1 and β_2 in Equations 3 and 4. These settings ensure the reward is inversely proportional to the completion times of jobs and tasks.

III. EdgeTuner

A. Overview

EdgeTuner is designed with two objectives.

1) Hot swapping scheduling algorithms for dynamic workloads and resources. The core component of EdgeTuner, the *DRL-based agent*, is external to the cluster scheduler and just operates on its scheduling algorithms. This design ensures minimum modifications to the scheduler, and more importantly, making it possible to replace any of them at runtime without shutting down the system. Specifically, the agent observes the state (cluster status) periodically (e.g. 10 seconds) and selects a scheduling algorithm for the cluster scheduler.

2) Simulator-based DRL training acceleration. We note that under *diverse workloads*, the whole training process needs a lot of experience (e.g. several million samples) to converge. However, in real job scheduling scenarios, the actor takes at least a few seconds to evaluate the effectiveness of an action (that is, the selection of a scheduling algorithm) and obtains a sample from the environment (the cluster). Even using the latest DRL training techniques (e.g. IQN+Ape-X [28]), the training may take dozens of hours to complete due to the long sampling phase. Moreover, when the cluster resource changes, the training process needs to be re-executed and the long training time makes the DRL agent infeasible for online scheduling algorithm tuning. Given this motivation, we develop a cluster simulator and use it as the environment for the actor. We explain how to train DRL agent under dynamic workloads and resources in Section III-B.

We incorporated the proposed approach with the Kubernetes schedulers [4] (Section III-C). Similar to other mainstream resource negotiation systems, such as Borg [31], Kubernetes provides access to various information regarding to resources, jobs, and scheduling constraints. When a job is submitted, Kubernetes also provides interfaces to obtain its submission time, resource demand, and task information. When an agent

generates an action according to the above state information, it is pushed to the Kubernetes scheduler that supports run-time adjustment of its scheduling algorithms.

B. Simulator-based Training of DRL Agent

The training of a DRL agent has two phases: in the **sampling** phase, the actor collects experience training samples by interacting with the simulator. The **learning** phase starts when a pre-specified number of samples is collected. Similar to other simulation platforms [35], [36], our simulator is driven by workload traces and it can generate an experience sample instantly, thus considerably accelerating the sampling and training process.

Algorithm 1 details the steps of the actor. It first initializes the environment by obtaining the latest network parameters (line 1) and getting initial state from environment (line 2). Subsequently, it iteratively obtains samples and adds them to the replay memory (lines 3 to 14). At each iteration, the actor first selects an action (a scheduling algorithm) and applies it to the environment (line 4). It then triggers environment.SimulateOneStep(s_{t-1},a_{t-1}) to obtain state s_t and reword r_t constructed using information of nodes, jobs, and tasks in the environment (line 5). Finally, it gets episode termination signal (line 6) and adds the sample data to the local buffer (line 7). When the buffer size is larger than the maximal size B, the actor calculates priorities for the current experience and triggers the remote call to add experience to the replay memory (lines 8 to 12). The actor also periodically obtains the latest network parameters (line 13).

Algorithm 1 Actor

Require: B: the maximal size of local buffer;
T_s: the number of sampling steps.

1. $\theta_0 \leftarrow$ learner.Parameters();
2. $s_0 \leftarrow$ environment.InitialState();
3. **for** t = 1 to T_s **do**
4. $a_{t-1} \leftarrow \pi_{\theta_{t-1}}(s_{t-1})$;
5. $s_t, r_t \leftarrow$ environment.SimulateOneStep(s_{t-1}, a_{t-1});
6. $\gamma_t \leftarrow$ environment.HasDone();
7. localBuffer.Add($(s_{t-1}, a_{t-1}, r_t, \gamma_t)$);
8. **if** localBuffer.Size() $> B$ **then**
9. $\tau \leftarrow$ localBuffer.Get(B);
10. $p \leftarrow$ ComputePriorities(τ);
11. replay.Add(τ, p);
12. **end if**
13. Periodically($\theta_t \leftarrow$ learner.Parameters()).
14. **end for**

The steps of function environment.SimulateOneStep(a_{t-1}) are explained in Algorithm 2. This function first gets the set N of nodes, and the waiting tasks V^w and running tasks V^r from state s_{t-1} (line 1). It then simulates resource allocations using a list of iterations (lines 3 to 7). At each iteration, the function sequentially allocates resources to waiting tasks using scheduling algorithm a_{t-1} (line 4) and checks the completion of running tasks at the current simulation time t^s (line 5). The

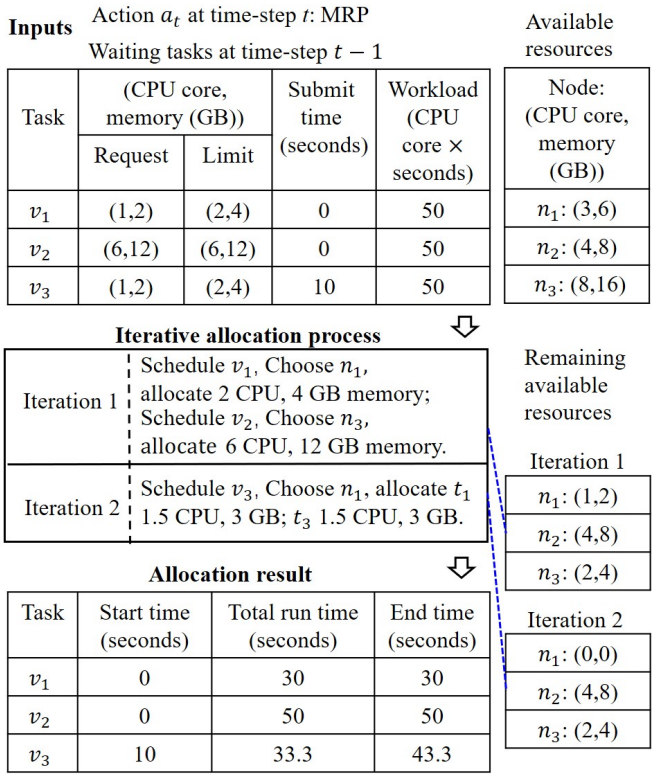

Fig. 2. Example of one time-step in the simulator

status of nodes, and waiting, running and completed tasks are updated before moving to the next iteration. The iterations end when the simulation time exceeds the scheduling interval. Finally, the function converts the information of nodes and tasks into state s_t and reward r_t of time-step t, and returns them (lines 8 to 11).

Algorithm 2 environment.SimulateOneStep(s_{t-1}, a_{t-1})

Require: t^s: the simulation time;
Δt: the duration of one iteration in simulation;
T^w,V^r,V^c: the sets of waiting, running, and completed tasks;
N: the set of nodes.

1. Obtain N, V^w, V^r from state s_{t-1};
2. t^s=0;
3. **while** t^s is smaller than the scheduling interval **do**
4. $N, V^w, V^r \leftarrow$ environment.AllocateResource $(N, Task^w, a_{t-1})$;
5. $N, V^r, V^c \leftarrow$ environment.CheckCompletion $(N, Task^r)$;
6. t^s=$t^s + \Delta t$;
7. **end while**
8. $s_t \leftarrow$ stateBuilder.Build(N, V^w, V^r);
9. $J^c \leftarrow$ GetCompletedJobs(V^c);
10. $r_t \leftarrow$ rewardBuilder.Build(J^c, V^c);
11. **return** s_t, r_t.

Example. Figure 2 displays an example of simulating resource allocations in a Kubernetes cluster at time-step t. In state s_{t-1}, there exist three nodes of different resource capacities, three waiting tasks v_1 to v_3, and no running task. The MRP algorithm is applied to allocate resources with two iteration. At iteration 1, it allocates node n_1's resources to task v_1 and node n_3's resources to task v_2 according to these tasks' resource limits. At iteration 2, it assigns task v_3 to the node n_1 with the least available resources. Given that there are insufficient resources to meet the resource limits of both tasks, the amounts of resources allocated to v_1 and v_3 are in proportion to their resource requests.

In the **learning** phase, the learner starts when the replay memory has L samples and trains the model using T^t iterations. At each iteration, the learner first samples a prioritized batch of experience (training samples), applies the learning rule, and updates the model parameters. Subsequently, it calculates and updates the priorities for experience and removes old experience from replay memory.

C. Implementation of Cluster Simulator

Our cluster simulator is implemented using Golang and it can support different operating systems such as Linux and Max-Os. Figure 3 illustrates the implementation of simulator-based DRL training and it consists of three major parts.

- **Simulator-based training environment**. At each sampling or training time-step t, the "Action executor" module is responsible for accepting an action from the agent, converting it into the corresponding scheduling algorithm, and forwarding it to the simulator. Subsequently, the "State builder" and "Reward builder" modules receive job and node information from the simulator and construct state s_t and reward r_t in the experience.
- **Http server**. The "Initial state" module is used to initialize the simulation environment, including initial node information, tasks waiting to be scheduled, and tasks that have been executed. Similarly, the "Interactive state" module stores the state information during the training process. The "Action" module provides scheduling algorithms such as LRP, MRP, BRA in the simulator.
- **Cluster simulator**. Each sampling/training time-step t (e.g. an episode) corresponds to multiple iterations in the simulator. At each iteration, the simulator first judges if there exists tasks/jobs to be scheduled in the waiting queue and if there are sufficient resources. If the available resources exceed the requested resources by the waiting tasks, the simulator applies a scheduling algorithm to assign the tasks to edge and cloud, and updates the node and task statuses; otherwise it updates the simulation time t^s. The simulation completes if t^s is longer than the scheduling interval; otherwise it continues scheduling the remaining tasks at the next iteration.

Applicability to other job scheduling scenarios. Our simulator implements the Kubernetes scheduler and its 10 scheduling algorithms at present. Within the context of DRL training, its simulation mechanism can be applied to other job scheduling scenarios: (1) *state*. The simulator generates state information by taking real cluster traces (e.g. Google traces [3] or Alibaba traces [1]) as input. Specifically, it first derives information of nodes and tasks, and adds tasks to priority queues. Subsequently, it emulates the process of resource allocation and task execution, during which the resource granularity can be containers or CPU/memories. After each time-step, it transforms the information of nodes and tasks into the state information that the DRL agent can understand. We note that the above generation process naturally applies to a wide range of job scheduling scenarios. (2) *Action*. The simulator includes a general scheduler that provides the interfaces of predicate and priority. It is convenient to implement a new job scheduling algorithm by extending the general scheduler with new predicates and priorities. (3) *Reward*. The current reward function (Equation 5) considers job and task latencies. It can be extended to incorporate other metrics in job scheduling, such as resource utilization or energy consumption.

Applicability to scheduling algorithms. Our simulator can replay the behaviours of resource allocation and task execution for most of the Kubernetes scheduling algorithms. The only exception is Equal Priority (EP), which randomly allocates resources to tasks. Our simulator cannot replay the behaviour of this algorithm because it may have two different resource allocations even when handling the same workload. We note that a DRL agent also cannot take such algorithms as actions, because they result in different rewards under the same state and thus disturb the optimization process in DRL training.

Discussion of system uncertainties. We note that in real clusters, a job's performance is also influenced by random background activities such as system maintenance or garbage collection of operating systems. These activities are not incorporated in our simulator for two reasons. First, although background activities can create considerable CPU or network load (in particular when resource are saturated), this work focuses on comparing job performances across different scheduling algorithms and implicitly assumes that the performances are estimated under the same factors (that is, different algorithms suffer from the same performance interferences). Second, in many practical scenarios (when systems have available resources for allocation), the performance impact of background activities is much smaller (e.g. 100 times smaller) than that caused by applying different scheduling algorithms.

IV. Evaluation

In this section, we evaluate the proposed approach with two major criteria: (1) its robust performance under diverse scheduling scenarios of different workloads and available resources (Section IV-B); and (2) its effectiveness in significantly accelerate DRL training (Section IV-C) and how it is influenced by DRL training settings (Section IV-D)

A. Evaluation settings

Evaluation platform. The experiments of job scheduling were performance in a Kubernetes cluster of 2 edge devices

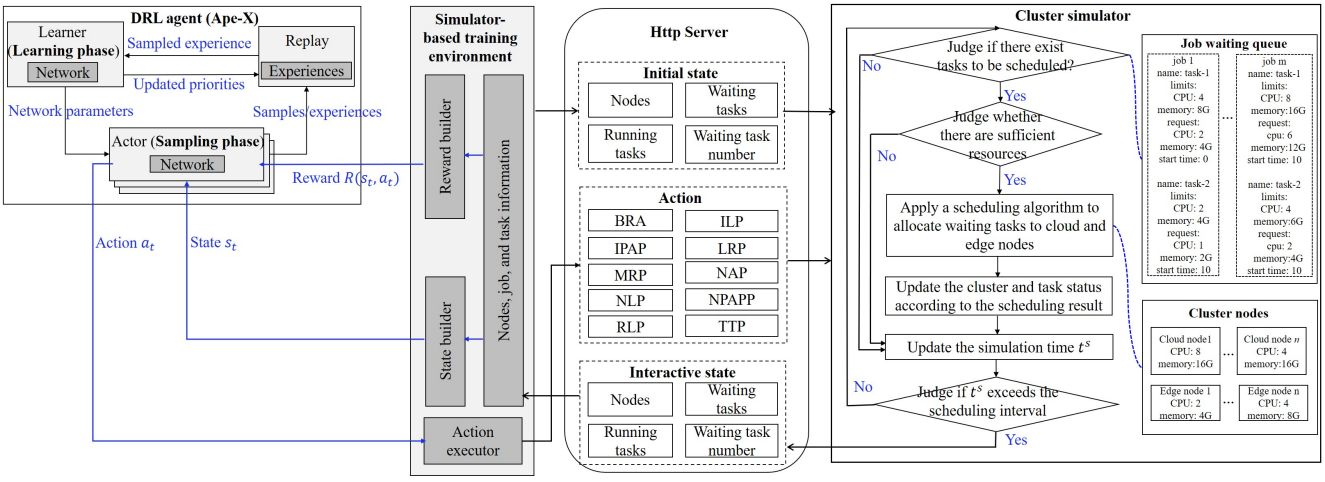

Fig. 3. Implementation of Cluster Simulator for DRL training

and 4 cloud nodes. Each edge device is Raspberry Pi 4B of 4-core 1.5GHz Cortex-A72 (ARM v8) and 4 GB memory. Each cloud node is equipped with 8 Intel E5-2660 v4 processor cores, and 8 GB of DRAM. The training of DRL agents was conducted in a GPU node of two Intel(R) Xeon(R) Silver 4216 processors, 48-GB Quadro RTX 8000 Graphics Card, and 256 GB memory. All nodes run Linux Ubuntu 18.04 LTS. In the Kubernetes cluster, the Python, Go, Docker, and Kubernetes versions are 3.8.5, 1.14.12, 19.03.8, and v1.19.3 respectively.

Real-trace driven scheduling scenarios. In evaluation, we generate three workload patterns of edge-cloud applications (edge-cloud, cloud-edge, and edge-cloud-edge workloads) following the 4034-node and 8-day Alibaba cluster trace 2018 [2]. Specifically, we derive the information of job arrival pattern, the number of tasks in a job, the resource (CPU and memory) request and resource limit of each task, and the workload (CPU core×seconds) from the trace. In job scheduling, we also consider two typical scenarios of daytime (6:00 to 24:00) and night (0:00 to 6:00). In the trace, 19,508 and 28,290 jobs, 6.92 and 7 million tasks are submitted in the daytime and at night, respectively.

DRL training setting. We implemented the proposed approach based on Google DeepMind's RainBow tool [13]. In DQN training, we use the latest technique [28] that combines Implicit Quantile Networks (IQN) [6] and Ape-X [15]. The training settings of three DRL elements are as follows:

- *State*. The interval between two time-steps is 10 seconds, hence the number of waiting and running tasks in a state is set to 10. In Equations 3 and 4, the values of α_1, β_1, α_2, and β_2 are set to -0.04, 10, -0.004, 1, respectively.
- *Actor*. The maximal number of time-steps is 50 millions and the experience replay memory capacity is set to 10 million. In sampling, the number of actors is 8, the history length (the number of consecutive states processed) is set to 4, and the frequency of sampling from memory is 4.
- *Learner*. The training phase starts after 10K time-steps of the sampling phase. In training, the network hidden node size is set to 64, the batch size is 32, the network is updated every 1000 steps, and the importance sampling weight in prioritised experience replay is 0.4.

Evaluation metrics. Our evaluation considers both job performance and training efficiency under dynamic workloads and resources. The *job performance* is measured by the average job latency and the *training efficiency* is measured by the sampling time and the training time in DRL training.

B. Improvement of Job Performance Under Dynamic Workloads and Resources

In this section, we evaluate the effectiveness of EdgeTuner in reducing job latencies by adaptively selecting its scheduling algorithms under dynamic workloads and resources. Here we compare against baselines with 11 representative algorithms in the Kubernetes cluster scheduler [4]: LRP, MRP, BRA, EP, Resource Limits Priority (RLP), Taint Toleration Priority (TTP), ode Affinity Priority (NAP), Image Locality Priority (ILP), Node Prefer Avoid Pods Priority (NPAPP), Node Label Priority (NLP), and Inter Pod Affinity Priority (IPAP). For the same set of jobs (tasks), these algorithms lead to different resource allocations and thus result in different job performances.

Dynamic workloads. This evaluation tests 6 workloads, covering 3 edge-cloud workload patters and 2 periods (daytime and night). Figure 4 uses box plots to illustrate each workload's distribution of job latencies, including their minimum and maximum values, the first quartile, median, and third quartile. We can observe that in most of the cases, our approach achieves lower latencies than other approaches, indicating the DRL agent selects proper scheduling algorithms for different workloads in the system. Specifically, most of jobs complete within a few seconds to dozens of minutes. This means the waiting and running tasks continuously change at different scheduling time-steps and the agent selects the algorithm that brings the largest reward (namely the shortest completion times of jobs and tasks).

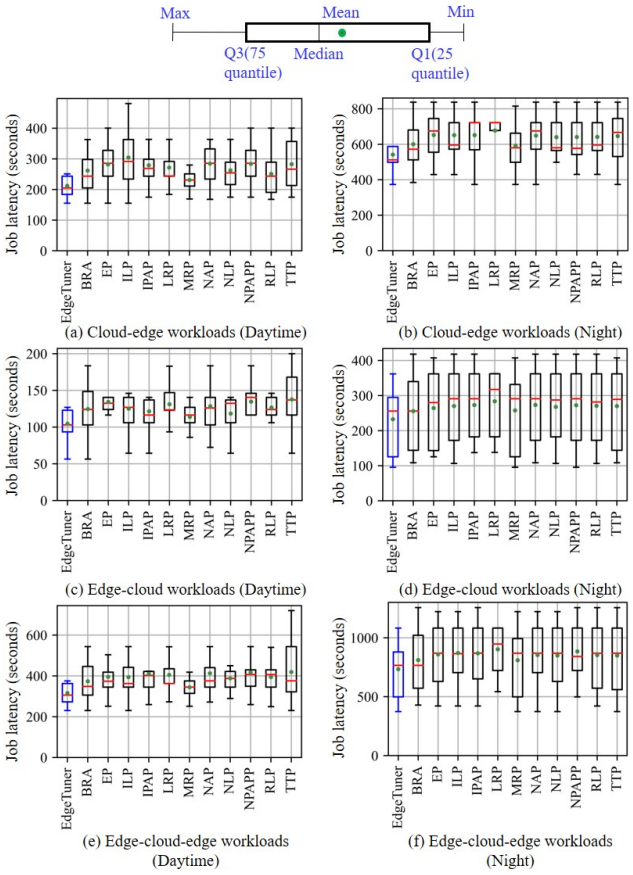

Fig. 4. Comparison of job latencies under dynamic workloads

Dynamic resources. We take the two cloud-edge workloads as example and further test dynamic resource changes. In four evaluations of either workload (daytime or night), there are 50% decrease, 25% decrease, 25% increase, and 50% increase in its cloud resources, respectively. Figure 5 displays the comparison results. The results show for the same workload (Figures 5(a) to (d) or Figures 5(e) to (h)), less available resources result in higher job latencies and larger discrepancies in job latencies. In different scenarios, our approach consistently brings the lowest job latencies because of its dynamic scheduling algorithm tuning mechanism. Note that when the cluster resource changes, the DRL agent needs to be re-trained because the node information changes in its state. In EdgeTuner, this training can be performed offline by setting different resources in its simulator for the same workload, thus avoiding the time-consuming online learning process.

Table I summarizes the percentages of reduced job latency, when comparing EdgeTuner against the baseline scheduling algorithms. We can see that these reductions vary across different workloads and different available resources depending on a variety of factors such as types of workloads, amounts of resources and scheduling algorithms. When considering all evaluation cases, our approach achieves an average of 15.78% reductions in job latencies.

C. Acceleration of DRL training

The effectiveness of tuning scheduling algorithms relies on efficiently training DRL agents. This sections evaluation tests an agent's training time consisting of two parts: (1) the major training time comes from collecting samples in the actor (Algorithm 1). Each training needs several million samples to converge and each sample needs at least a few seconds to obtain in real clusters; (2) using the collected samples in the replay memory, the learner trains the DQN model. We compare EdgeTuner (collecting samples with the Kubernetes simulator in Algorithm 2) and the real Kubernetes cluster following the experiment settings of the previous section. After convergency, EdgeTuner and the real cluster have negligible differences in their rewards.

Table II lists the sampling times and training times under different workloads and resources (the same 14 scenarios as Table I). We can see that in all cases, the sampling phase takes a long time (more than 90 hours) in the real Kubernetes cluster and EdgeTuner considerably reduce this time to a few seconds (acceleration by up to 35630.04x). Similarly, in Kubernetes, the training phase also completes in dozens of hours to several hundreds hours due to the time-consuming sampling process. In contrast, EdgeTuner completes the training phase within a few hours. We can also observe that the training time also varies across different states (i.e. different jobs, tasks, and nodes) and our approach consistently reduces training time by 122.57x. When considering both sampling and training phases, our approach accelerates DRL training by more than 180x and more importantly, because it enables the training to be performed offline and can provide the adapted DRL agent timely.

D. Discussion of DRL Training Settings

We now take the cloud-edge workload as an example and design experiments to discuss the *three major factors* that influence DRL training efficiency. Five metrics are used in evaluation: sampling time, training time, the total number of samples, the total number of sampling and training iterations, and the total time of sampling and training phases.

DRL training techniques. In DRL training, this work employs the latest Rainbow tool [14] combined with two model training techniques: IQN for distributional reinforcement learning [6], and Ape-X for distributed sampling and prioritized experience replay [15]. Figure 6(a) shows that IQN incurs the longest sampling time because it only uses one actor. In contrast, Ape-X supports multiple actors and considerably reduces the sampling time when collecting the same number of samples. In addition, Figure 6(b) shows that Ape-X has the longest training time because the IQN technique accelerates the convergence speed. To verify this claim, Figures 6(c) and (d) show that Ape-X needs the largest samples and training iterations, and thus takes the longest time to complete the training process (Figure 6(e)).

Different numbers of actors. This evaluation considers three different numbers of actors: 8 (used in EdgeTuner in previous evaluations), 4, and 16. Figure 7(a) shows that more

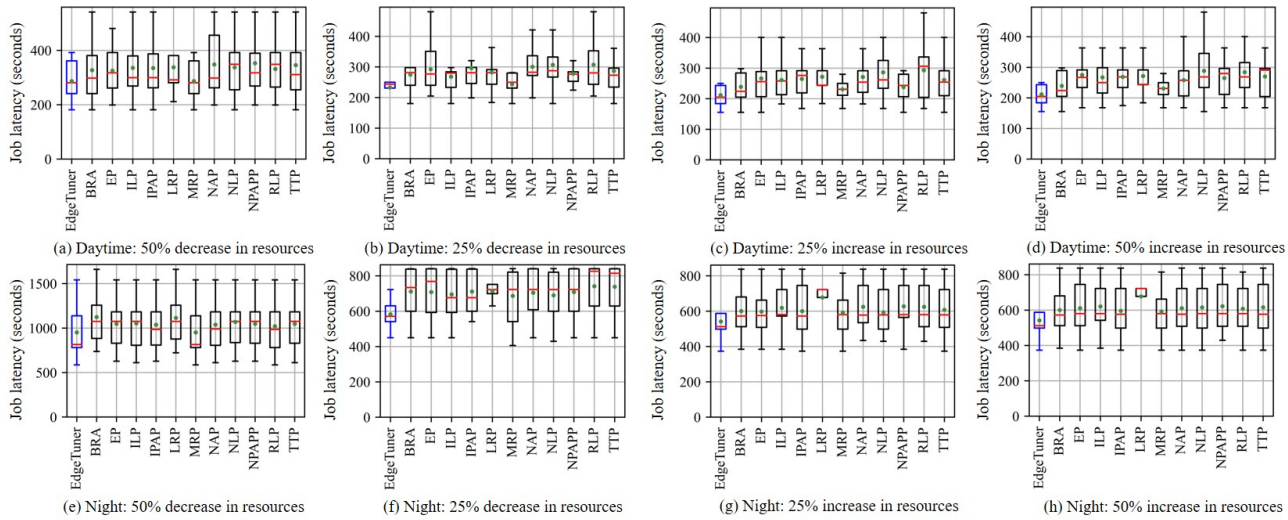

Fig. 5. Comparison of job latencies under dynamic available resources

TABLE I
PERCENTAGES OF REDUCED JOB LATENCIES UNDER DYNAMIC WORKLOADS AND RESOURCES

Scenario	Job scheduling scenarios			BRA	EP	ILP	IPAP	LRP	MRP	NAP	NLP	NPAPP	RLP	TTP
1	Cloud-edge	daytime	Original resources	19.16	25.18	30.59	24.37	22.14	8.66	25.70	19.47	25.44	15.94	25.18
2	workloads	night	Original resources	9.83	17.02	16.90	16.90	20.09	8.46	16.64	15.47	15.60	15.60	16.25
3	Edge-cloud	daytime	Original resources	12.90	19.40	16.92	7.69	17.56	7.69	10.74	10.74	12.20	14.96	27.03
4	workloads	night	Original resources	9.02	12.12	14.07	15.02	18.02	9.73	15.02	13.11	14.71	14.07	14.07
5	Edge-cloud-	daytime	Original resources	15.28	20.00	19.80	22.74	21.98	8.14	23.30	18.56	24.22	19.80	24.58
6	edge workloads	night	Original resources	9.51	14.67	15.75	15.55	18.74	9.39	14.07	13.66	17.08	14.07	13.66
7	Cloud-edge workloads	daytime	50% decrease	12.54	12.00	14.63	14.37	15.13	0.00	17.82	15.13	18.75	13.60	17.10
8			25% decrease	15.33	20.55	13.43	21.62	17.73	4.53	22.67	24.18	16.55	24.43	18.88
9			25% increase	11.34	20.68	18.85	20.08	22.14	8.26	21.85	26.22	10.97	27.99	19.16
10			50% increase	11.72	22.99	20.97	21.27	22.14	8.26	18.22	26.74	20.08	25.70	21.56
11		night	50% decrease	15.39	9.17	9.60	8.29	14.63	0.00	8.38	11.04	9.17	6.76	9.08
12			25% decrease	18.14	17.80	16.26	18.14	18.49	15.16	17.33	15.65	17.80	21.46	21.14
13			25% increase	9.83	9.38	12.46	9.83	20.09	8.46	13.30	8.77	13.72	13.30	10.87
14			50% increase	9.83	11.46	12.74	9.08	20.09	8.46	11.31	12.03	12.88	10.87	12.03

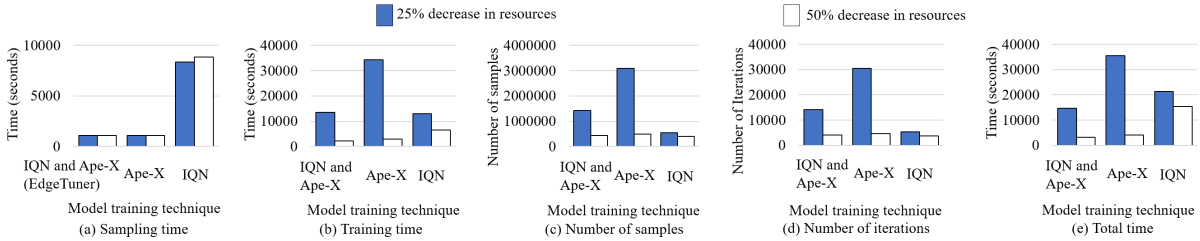

Fig. 6. Comparison of DRL training overheads under different model training techniques

actors indeed reduce sampling time. However, when the actor number is 16, the sampling speed exceeds the training speed. This means the 16 actors need to wait for the learner while occupying resources, thus delaying the whole training process (Figure 7(b)). This claim is verified in Figures 7(c) and (e)'s results. We can also observe that the scenario of 50% decrease in resources is simpler because it has less available resources (that is, smaller state space). Hence in this scenario, the model training needs fewer samples (Figure 7(c)) and iterations (Figure 7(d)), and converges more quickly (Figure 7(e)).

History length. In DRL training, history length decides the number of time-steps used to construct a state in the environment. Conceptually, the longer the history length, the more information the agent can learn from a state. This evaluation considers three history lengths: 4 (used in EdgeTuner in previous evaluations), 1, and 8. The results in Figure 8 display that when the history length is 1 (that is, the states in different iterations are independent of each other), the training needs the largest numbers of samples and the longest time to converge. In contrast, when the history length is 8, the training needs the smallest number of samples (Figure 8(c)), but its sampling time is still longer than that of history length 4 in the scenario

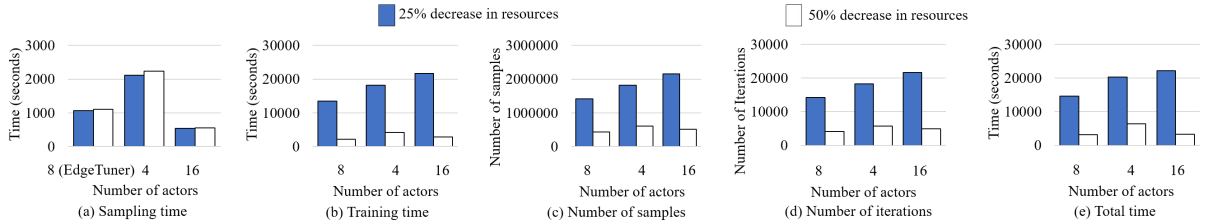

Fig. 7. Comparison of DRL training overheads under different actors

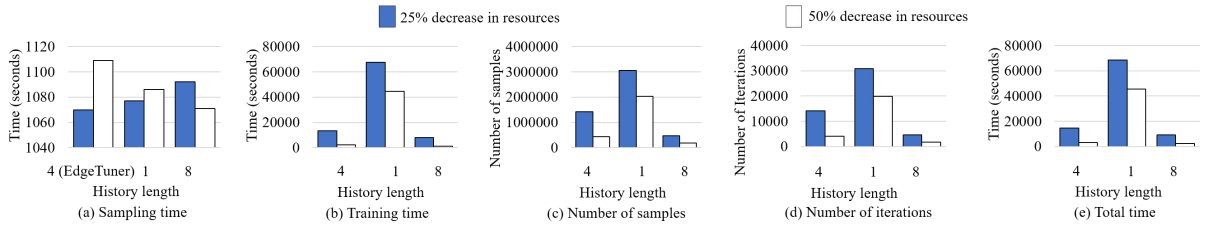

Fig. 8. Comparison of DRL training overheads under different history lengths

TABLE II
DRL TRAINING TIME BREAKDOWN UNDER DYNAMIC WORKLOADS AND RESOURCES

Scenario	Sampling time (seconds)		Training time (seconds)	
	EdgeTuner/Kubernetes	Reductions	EdgeTuner/Kubernetes	Reductions
1	11/336160	30560x	5824/744708	127.87x
2	7/325463	46494.71x	3689/448439	121.56x
3	9/312343	34704.78x	9425/1107046	117.46x
4	8/314422	39302.75x	2222/264988	119.26x
5	8/281755	35219.38x	13051/1386341	106.22x
6	6/288622	48103.67x	4793/521044	108.71x
7	12/337417	28118.08x	8053/1020424	126.71x
8	10/335230	33523x	6947889505	128.04x
9	11/336568	30597.1x	9125/1149461	125.97x
10	12/334561	27880.08x	13003/1647970	126.74x
11	8/338561	42320.13x	2128/273040	128.31x
12	10/337088	37454.22x	13534/1672159	123.55x
13	10/338766	33876.6x	4563/573715	125.73x
14	11/337326	30666x	3455/448519	129.82x

of 25% decrease in resources (Figure 8(a)). This is because the state of history length 8 is two times larger than that of history length 4 and hence each sample's collection time is longer in the former setting.

V. Related Work

In modern cloud data centers, cluster resource management systems (e.g. YARN [30] and Kubernetes [4]) provide multiple scheduling algorithms/policies to control resource allocation to their jobs [7], [10], [21], [23], [40]. This work focuses on edge-cloud jobs with diverse workload characteristics including stochastic arrival rate, different resource demands and durations. Our approach is built upon existing configurable schedulers and employ DRL to tune their scheduling algorithms online. Early work in this area applies reinforcement learning (RL) techniques to schedule jobs at particular time slots and minimize their latencies [20]. Later techniques employ state-of-the-art DRL techniques to accelerate the training speed [17], [28]. However, in this edge-cloud job scheduling scenario, they still suffer from **time-consuming sampling phase** because: (i) a large number of samples (e.g. over 1 million) are needed; (ii) a sample can only be obtained every few seconds (as shorter intervals cannot reflect the changes in workloads).

To address the above problems, recent approaches train DRL agents in an offline manner [35], [36]. The training is driven by a neural network based computational model, which predicts system states and generates training samples based on history traces. Similarly, DeepEE develops a simulation platform to emulate dynamic IT workloads and cooling systems [24]. These techniques target *long-running and compute-intensive jobs* in HPC data centers and differ from this work's scenario in the following two aspects. First, the long-running jobs follow an arrival queue and they are dispatched to proper servers according to **a fixed job scheduling algorithm**. Second, latency is not a key concern of compute-intensive jobs. In DRL training, these techniques implicitly assume **fixed available resources** in the cluster.

VI. Conclusion

In this paper, we present EdgeTuner to optimize resource allocation and scheduling algorithms for dynamic edge-cloud workloads. The core component of EdgeTuner is a DRL agent for run-time scheduling algorithm tuning and a cluster simulator to accelerate the lengthy DRL training process. EdgeTuner is implemented on Kubernetes and evaluated using production-system workloads on real clusters.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 61872337 and No. 62132019), the National Research and Development Program of China (No. 2019YQ1700), and the Swiss National Science Foundation NRP75 project 407540_167266. Equal contributors: Rui Han and Shilin Wen. Corresponding author: Rui Han.

REFERENCES

[1] Alibaba cloud trace. https://github.com/alibaba/clusterdata.

[2] Alibaba cluster trace. https://github.com/alibaba/clusterdata.

[3] Google cluster trace. https://github.com/google/cluster-data.

[4] Google kubernetes. http://kubernetes.io.

[5] Djabir Abdeldjalil Chekired, Mohammed Amine Togou, Lyes Khoukhi, and Adlen Ksentini. 5g-slicing-enabled scalable sdn core network: Toward an ultra-low latency of autonomous driving service. *IEEE Journal on Selected Areas in Communications*, 37(8):1769–1782, 2019.

[6] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. In *ICML'18*, pages 1096–1105. PMLR, 2018.

[7] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *EuroSys'18*, pages 1–13, 2018.

[8] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 455–466. ACM, 2014.

[9] Rui Han, Chi Harold Liu, Shilin Li, Shilin Wen, and Xue Liu. Accelerating deep learning systems via critical set identification and model compression. *IEEE Transactions on Computers*, 69(7):1059–1070, 2020.

[10] Rui Han, Chi Harold Liu, Zan Zong, Lydia Y Chen, Wending Liu, Siyi Wang, and Jianfeng Zhan. Workload-adaptive configuration tuning for hierarchical cloud schedulers. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2879–2895, 2019.

[11] Rui Han, Qinglong Zhang, Chi Harold Liu, Guoren Wang, Jian Tang, and Lydia Y Chen. Legodnn: block-grained scaling of deep neural networks for mobile vision. In *MobiCom'21*, pages 406–419, 2021.

[12] Zijiang Hao, Shanhe Yi, and Qun Li. Nomad: An efficient consensus approach for latency-sensitive edge-cloud applications. In *INFOCOM'19*, pages 2539–2547, 2019.

[13] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *AAAI'18*, 2018.

[14] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *AAAI'2018*, volume 32, 2018.

[15] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

[16] Tarun Kulshrestha, Divya Saxena, Rajdeep Niyogi, and Jiannong Cao. Real-time crowd monitoring using seamless indoor-outdoor localization. *IEEE Transactions on Mobile Computing*, 19(3):664–679, 2019.

[17] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *ICDCS'17*, pages 372–382, 2017.

[18] Yang Liu, Yuchen Zhou, and Shiyan Hu. Combating coordinated pricing cyberattack and energy theft in smart home cyber-physical systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(3):573–586, 2017.

[19] Xiao Ma, Ao Zhou, Shan Zhang, and Shangguang Wang. Cooperative service caching and workload scheduling in mobile edge computing. In *INFOCOM'20*, pages 2076–2085, 2020.

[20] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *HotNets'16*, pages 50–56, 2016.

[21] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *SIGCOMM'19*, pages 270–288. 2019.

[22] Shagufta Mehnaz and Elisa Bertino. Privacy-preserving real-time anomaly detection using edge computing. In *ICDE'20*, pages 469–480. IEEE, 2020.

[23] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *EuroSys'18*, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Y. Ran, H. Hu, X. Zhou, and Y. Wen. Deepee: Joint optimization of job scheduling and cooling control for data center energy efficiency using deep reinforcement learning. In *ICDCS'19*, pages 645–655, 2019.

[25] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[26] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[27] Boxiong Tan, Hui Ma, Yi Mei, and Mengjie Zhang. A cooperative coevolution genetic programming hyper-heuristic approach for on-line resource allocation in container-based clouds. *IEEE Transactions on Cloud Computing*, 2020.

[28] Marin Toromanoff, Émilie Wirbel, and Fabien Moutarde. Is deep reinforcement learning really superhuman on atari? *CoRR*, abs/1908.04683, 2019.

[29] Chun-Wei Tsai, Wei-Cheng Huang, Meng-Hsiu Chiang, Ming-Chao Chiang, and Chu-Sing Yang. A hyper-heuristic scheduling algorithm for cloud. *IEEE Transactions on Cloud Computing*, 2(2):236–250, 2014.

[30] Vinod Kumar Vavilapalli, Arun C Murthy, and et al. Apache hadoop yarn: yet another resource negotiator. In *SoCC'13*, page 5. ACM.

[31] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *EuroSys'15*, page 18. ACM, 2015.

[32] Ji Wang, Jianguo Zhang, Weidong Bao, Xiaomin Zhu, Bokai Cao, and Philip S Yu. Not just privacy: Improving performance of private deep learning in mobile cloud. In *SIGKDD'18*, pages 2407–2416, 2018.

[33] S. Wang, S. Yang, and C. Zhao. Surveiledge: Real-time video query based on collaborative cloud-edge deep learning. In *INFOCOM 2020*, pages 2519–2528, 2020.

[34] Qin-zhe Xiao, Jinghui Zhong, Liang Feng, Linbo Luo, and Jianming Lv. A cooperative coevolution hyper-heuristic framework for workflow scheduling problem. *IEEE Transactions on Services Computing*, 2019.

[35] D. Yi, X. Zhou, Y. Wen, and R. Tan. Toward efficient compute-intensive job allocation for green data centers: A deep reinforcement learning approach. In *ICDCS'19*, pages 634–644, 2019.

[36] D. Yi, X. Zhou, Y. Wen, and R. Tan. Efficient compute-intensive job allocation in data centers via deep reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1474–1485, 2020.

[37] Q. Zhang, Q. Zhang, W. Shi, and H. Zhong. Firework: Data processing and sharing for hybrid cloud-edge analytics. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2004–2017, 2018.

[38] Y. Zhang and V. S. Sheng. Fog-enabled event processing based on iot resource models. *IEEE Transactions on Knowledge and Data Engineering*, 31(9):1707–1721, 2019.

[39] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

[40] Zan Zong, Lijie Wen, Xuming Hu, Rui Han, Chen Qian, and Li Lin. Mespaconfig: Memory-sparing configuration auto-tuning for co-located in-memory cluster computing jobs. *IEEE Transactions on Services Computing*, 2021.