2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS) | 978-1-6654-0386-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/RTAS52030.2021.00019

Developing Real-Time Scheduling Policy by Deep Reinforcement Learning

Zitong Bo*†, Ying Qiao*, Chang Leng*, Hongan Wang*, Chaoping Guo* and Shaohui Zhang‡
*Beijing Key Lab of Human-Computer Interaction, Institute of Software, Chinese Academy of Sciences
†University of Chinese Academy of Sciences
‡Beijing National Speed Skating Oval Operation Co., Ltd
zitong2019@iscas.ac.cn

***Abstract*—Designing scheduling policies for multiprocessor real-time systems is challenging since the multiprocessor scheduling problem is NP-complete. The existing heuristics are customized policies that may achieve poor performance under some specific task loads. Thus, a new design pattern is needed to make the multiprocessor scheduling policies perform well under various task loads. In this paper, we investigate a new real-time scheduling policy based on reinforcement learning. For any given real-time task set, our policy can automatically derive a high performance by online learning. Specifically, we model the real-time scheduling process as a multi-agent cooperative game and propose multi-agent self-cooperative learning that overcomes the curse of dimensionality and credit assignment problems. Simulation results show that our approach can learn high-performance policies for various task/system models.**

***Index Terms*—real-time scheduling, reinforcement learning, multiprocessor system, deep neural network**

I. Introduction

Developing scheduling policies for aperiodic tasks running in multiprocessor real-time systems is always challenging work. The multiprocessor scheduling problem is NP-complete [1], which means that the optimal scheduling policy cannot be implemented in polynomial time unless P=NP. Therefore, some heuristic policies have been investigated, such as GEDF [2], PFair [3]–[5], EDF-BF [6], EDHS [7],etc. These policies are simple to implement since they determine priorities directly by task parameters. Only PFair is optimal for periodic tasks when system utilization is less than the number of processors [4]. However, the runtime overhead of PFair is too high from the implementation point of view. Furthermore, all of the above heuristic policies may achieve poor performance in real-time systems executing aperiodic loads. Traditional heuristics are customized by the expert depending on his domain knowledge. In practice, there are two significant drawbacks to this traditional design pattern. Firstly, the complete prior knowledge of task behaviors is unlikely to be available, so that employing experts to design customized scheduling policy could be very costly. Secondly, real-time systems in the real world often violate the assumptions of the customized scheduling policy. For example, a system may have non-negligible high preemption overhead or migration overhead. Therefore, a new design pattern for developing real-time scheduling policies is urgently needed.

One ideal approach to the above issues is to develop an 'artificial expert' on real-time scheduling, i.e., implementing a method that can automatically generate customized policies for various multiprocessor systems with given system and task information.

We find that reinforcement learning (RL) is a promising approach to develop sterling scheduling policies. With the advent of AlphaGo [8], reinforcement learning has gradually become a hotspot in the artificial intelligence (AI) research area. The integration of the deep learning has greatly promoted successful applications of reinforcement learning in solving real-world complex applications [9], such as chips design [10], robotic manipulation [11] and video games [12]. Real-time scheduling problems have one common key feature with the above applications, i.e., they are massive search problems where exhaustive or heuristic-based methods cannot scale. The considerable success of the above applications has shown that reinforcement learning is good at solving such massive search problems. Therefore, we deem that reinforcement learning is a promising way to develop real-time scheduling policies.

However, using RL to learn real-time scheduling policy is not trivial work. In most real-time systems, aperiodic tasks may have more than one active instance in some time steps so that the number of instances continuously changes during system runtime. Hence, the number of the system states grows exponentially with the number of instances, resulting in poor convergence for RL. This phenomenon is called the curse of dimensionality [13], which exists extensively in machine learning. Besides, since real-time systems are sensitive to time overheads, the learned scheduling policies should have low overheads, which further increases the designing difficulty of the policy architecture. Moreover, the reward function design for RL has a significant influence on the performance of the learned scheduling policy. Thus, designing a suitable policy architecture and reward function is also challenging work.

In this paper, we investigate a novel multi-agent deep reinforcement learning framework to learn real-time scheduling policies having high success ratio for multiprocessor systems. Our approach keeps computing proper priorities for task instances, which should be strongly desired by real-time multiprocessor systems that deal with unpredictable frequent task instance arrivals. The proposed framework can also be easily extended to many applications having different system

978-1-6654-0386-3/21/$31.00 ©2021 IEEE
DOI 10.1109/RTAS52030.2021.00019

models. Our main contributions are summarized as follows:

(1) We formulate the real-time scheduling process as Markov game and investigate a new real-time scheduling method based on reinforcement learning, i.e., a deep reinforcement learning framework to learn scheduling policies online, improving the scheduling performance of real-time multiprocessor systems. Instead of directly designing heuristics for specific task sets, the proposed reinforcement learning approach can derive high-performance scheduling policies through continuously improving the policy online. This framework can be extended to other real-time systems with various characteristics.

(2) We propose a multi-agent self-cooperative learning method, a novel reinforcement learning algorithm that solves the credit assignment problem in the Markov game. The proposed method is based on multi-agent actor-critic, and we also adopt the framework of centralized training with decentralized execution.

(3) We have implemented the framework as well as a multi-agent self-cooperative learning algorithm, using Python and PyTorch [14]. Simulation results for randomly generated loads show that our RL approach has good convergence, mitigating the curse of dimensionality. Besides, the scheduling policy learned by reinforcement learning has low, stable overhead and high performance.

The rest of this paper is organized as follows. In section II, we introduce the system model as well as some concepts about reinforcement learning. Section III proposes the reinforcement learning architecture in detail. In section IV, we conduct some experiments to demonstrate the performance of the proposed approach. In section V, a use case is presented. In section VI, we review some related works. Summary, discussion, and future work will be given in section VII.

II. Preliminaries

A. System Model

The system considered in this paper has M identical processors, i.e., $P = \{P_1, P_2, ..., P_M\}$, and schedules a set of N real-time tasks, i.e., $T = \{T_1, T_2, ..., T_N\}$. Since we focus on designing a scheduler for aperiodic tasks, we only consider individual task instances, i.e., jobs. An active job is a task instance that has been released but has not been completed. The active job set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ consists of all the active jobs. Each job τ_i is associated with the following parameters: arrival time A_i; execution time $C_i \geq 0$; relative deadline $D_i > C_i$. The absolute deadline of τ_i is $d_i = A_i + D_i$. At time t, the remaining execution time of τ_i, denoted by $c_i(t)$, is defined as the execution time of τ_i minus the cumulative execution time that τ_i has consumed by t. τ_i is completed when $c_i(t) = 0$. The completion time of τ_i is denoted by f_i. Hence, $c_i(t) = 0$ if $t \geq f_i$. A job τ_j misses its deadline when $t > d_i$. The parameters of different jobs are independent.

Different jobs do not have precedence relations and do not share resources except the processor. The tasks are independent, i.e., a job's arrival of a task will not be affected by other tasks. Without loss of generality, we assume that each task releases at most one new job at any time instant. It should be noted that multiple instances of a single task may exist at some point. One can easily extend this paper's work to many other systems, which will be discussed in the next section.

B. Reinforcement Learning

Reinforcement learning is a separate branch of machine learning in which a policy learns to take optimal actions in an environment (either the real world or a simulation) to maximize a given reward function. During the learning process, the agent interacts with an environment by perceiving states, taking actions, and obtaining rewards [15], as shown in Fig.1. RL problems can be reformulated as a Markov Decision Process (MDP). The MDP relies on the Markov assumption, meaning that the next state s_{t+1} depends only on the current state s_t and is conditionally independent of the past.

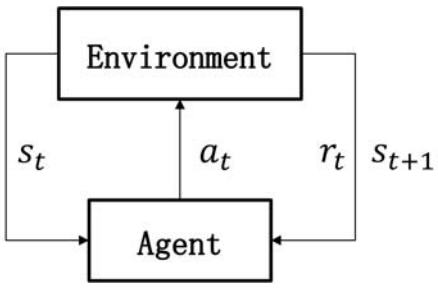


Fig. 1. In Reinforcement learning, an agent interacts with the environment.

MDP [16] consists of a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a transition system $\mathcal{P} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$, and a reward function $\mathcal{R} : \mathcal{S} \mapsto \mathcal{R}$. At each discrete decision epoch t, a controller observes the current MDP state s_t and selects an action a_t. The MDP then makes a transition to state s_{t+1} distributed according to $P(s_{t+1}|s_t, a_t)$ and incurs reward $r(s_{t+1})$. $r_t = r(s_t, a_t) = \mathbb{E}_{s_{t+1} \sim P(s_t, a_t)}[r(s_{t+1})]$

A solution to an MDP is a policy which is a state-to-action mapping. The stochastic policy a probability distribution of action and the deterministic policy is an unique mapping from state to action $\pi(s) = a$. The value function of a policy is a prediction of the expected, accumulative, future reward $R_t = \sum_{k=t}^{\infty} r_k$, measuring policy quality. The future reward is often discounted by γ, i.e., $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$. The state value function $V^{\pi}(s) = \mathbb{E}[R_t|s_t = s]$ is the expected return for following policy π from state s. $V^{\pi}(s)$ decomposes into the Bellman equation: $V^{\pi}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V^{\pi}(s')]$. An optimal state value function $V^*(s) = max_{\pi} V^{\pi}(s) = max_a Q^*(s, a)$ is the maximum state value achievable by any policy for state s. $v^*(s)$ decomposes into the Bellman equation: $V^*(s) = max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_*(s')]$. The action value function $Q^{\pi}(s, a) = \mathbb{E}[R_t|s_t = s, a_t = a]$ is the expected return for selecting action a in state s and then following policy π. $Q^{\pi}(s, a)$ decomposes into the Bellman equation: $Q^{\pi}(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_a \pi(a'|s') Q_{\pi}(s', a')]$. An optimal action value function $Q^*(s, a) = max_{\pi} Q^{\pi}(s, a)$ is

the maximum action value achievable by any policy for state s and action a. $Q^*(s,a)$ decomposes into the Bellman equation: $Q^*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma max_{a'} Q^*(s',a')]$.

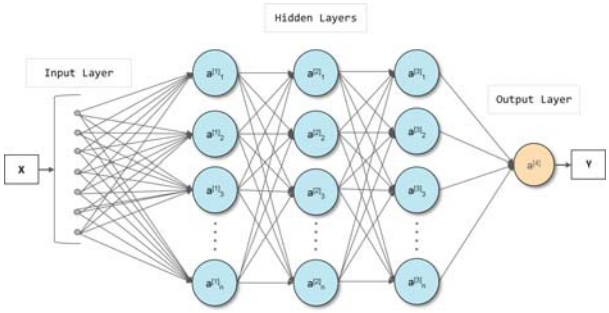


Fig. 2. A feedforward deep neural network or multilayer perceptron consists of an input layer, an output layer, and N hidden layers. Each layer contains several neurons and there are connections between neurons in each layer.

Since the goal of an agent is to maximize cumulative reward, one approach is to learn the state value function that can predict the reward for a given state, $V^\pi(s)$, and then take the action which will bring the agent into a state that obtains the highest reward. However, in recent years, a more common approach is to use policy gradient methods, which seek to directly learn the policy $\pi(s)$ that predicts the optimal action given the current state. Popular policy gradient methods include A3C [17], DPG [18], and PPO [19].

Deterministic policy gradients (DPG) [18] algorithm extends the policy gradients algorithm to deterministic policies $\pi_\theta : \mathcal{S} \mapsto \mathcal{A}$. In particular, under certain conditions we can write the gradient of the objective $J(\theta) = \mathbb{E}_{s\sim\rho^\pi}[r(s,a)]$ as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\pi}[\nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)|_{a=\pi_\theta(s)}], \quad (1)$$

ρ^π is the state distribution under π. Since this theorem relies on $\nabla_a Q^\pi(s,a)$, it requires the action space (and thus the policy) be continuous.

We obtain deep reinforcement learning methods when we use deep neural networks to approximate any of the following components of reinforcement learning: value function, $V(s,\theta)$ or $Q(s,a;\theta)$, policy $\pi(a|s;\theta)$, and model (state transition function and reward function). Here, the parameters are the weights and biases in deep neural networks. A feedforward deep neural network (NN) or multilayer perception (MLP) maps a set of input values to output values with a mathematical function formed by composing many simpler functions at each layer. After computations flow forward from input to output, at the output layer and each hidden layer, we can compute error derivatives backward and backpropagation gradients towards the input layer so that weights can be updated to optimize some loss function.

Deep deterministic policy gradients (DDPG) [20] is a variant of DPG where the policy π and critic Q^π are approximated with deep neural networks. DDPG is an off-policy algorithm, it samples trajectories from a replay buffer of experiences that are stored throughout training.

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\mathcal{D}}[\nabla_\theta \pi_\theta(a|s) \nabla_a Q^\pi(s,a)|_{a=\pi_\theta(s)}], \quad (2)$$

where $\mathcal{D}$ is the replay buffer. The objective of critic is:

$$\mathcal{L}(\theta^Q) = \mathbb{E}_{s,a,r,s'\sim\mathcal{D}}[(y - Q(s,a|\theta^Q))^2]$$
$$where\ y = r + \gamma Q'(s', \pi'(s'|\theta^{\pi'})|\theta^{Q'}) \quad (3)$$

DDPG also makes use of the target network, in which Q' is the target Q function and π' is the target policy. The parameters of target networks are periodically updated with the most recent θ, which helps stabilize learning.

There are some important applications that involve interaction between multiple agents, where emergent behavior and complexity arise from agents co-evolving together. Markov game (MG) is a multi-agent extension of the MDP [21]. A Markov game for N agents is defined by a set of states $\mathcal{S}$ describing the possible configurations of all agents, a set of actions $\mathcal{A}^1$, ..., $\mathcal{A}^N$ and a set of observations $\mathcal{O}^1$, ..., $\mathcal{O}^N$ for each agent. Each agent i uses a policy π_{θ_i} to select action, which produces the next state according to the state transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A}^1 \times ... \times \mathcal{A}^N \mapsto \mathcal{S}$. Each agent i receives a private reward r^i as well as a observation correlated with the state $O^i : \mathcal{S} \mapsto O^i$ [22]. In some environments, there are only a global reward for all agents.

Directly applying single-agent reinforcement learning algorithms to the multi-agent setting by treating other agents as part of the environment is problematic as the environment appears non-stationary from a particular agent's view, violating Markov assumptions required for convergence. Particularly, this non-stationary issue is more severe in the case of deep reinforcement learning with neural networks as function approximators [23]. The extension of DDPG in the multi-agent setting is Multi-agent DDPG (MADDPG) [24]. The core idea of MADDPG is to train a centralized Q function for each agent which conditions on global state and actions of all agents, to alleviate the non-stationary problem and stabilize training. The gradient for agent i is:

$$\nabla_{\theta_i} J(\theta_i) = \mathbb{E}_{s,a\sim\mathcal{D}}[\nabla_{\theta_i} \pi_i \nabla_{a^i} Q_i^\pi(s, a^1, ..., a^N)|_{a^i=\pi_{\theta_i}(o^i)}], \quad (4)$$

Here $Q_i^\pi(s, a^1, ..., a^N)$ is a centralized action-value function that takes as input the actions of all agents $a^1, ..., a^N$, in addition to some state information s (e.g, $s = (o^1, ..., o^N)$). Let s' denote the next state from s after taking actions $a^1, ..., a^N$. The centralized action-value function Q_i^π is updated by:

$$\mathcal{L}(\theta_i) = \mathbb{E}_{s,a,r,s'}[(Q_i^\pi(s, a^1, ..., a^N) - y)^2],$$
$$y = r_i + \gamma Q_i^{\pi'}(s', a^{1\prime}, ..., a^{N\prime})|_{a^{j\prime}=\pi^{j\prime}(o^j)}, \quad (5)$$

where $\pi' = \{\pi_{\theta'_1}, ..., \pi_{\theta'_N}\}$ is the set of target policies.

III. Reinforcement Learning for Real-time Scheduling

To show how RL can be applied to solve the real-time scheduling problem, we first start with the policy architecture

design. Then, we formulate the real-time scheduling problem as MG. At last, we present how to learn real-time scheduling policy in detail. Specifically, we propose multi-agent self-cooperative learning to solve the credit assignment problem.

A. Policy Architecture Design

The basic purpose of a real-time scheduler is to find an optimal order to execute the active jobs/tasks. Inspiring by most heuristic schedulers, we intuitively design the RL policy's input as a sequence of job parameters and output as a sequence of corresponding job priorities $p_1, p_2, ..., p_n$. In this setup, we use RL to find a mapping

$$(\tau_1, \tau_2, ..., \tau_n) \longrightarrow (p_1, p_2, ..., p_n) \tag{6}$$

in each time step to maximize the success ratio of the scheduling process.

In order to deal with the variable-length sequence, the most straightforward way is using the recurrent neural network (RNN) [25] and encoder-decoder architectures [26] as policy. The encoder-decoder has been widely used in the neural language process [27], which maps a variable-length sequence input with output. Since most heuristic schedulers, such as LSF and EDF, derive the priority from the corresponding job's local parameters, another solution is using a NN to represent the policy and derive the priority for each job separately. Besides, some global information, such as the minimum deadline of all active jobs, can be appended to the input to explore more efficient policies.

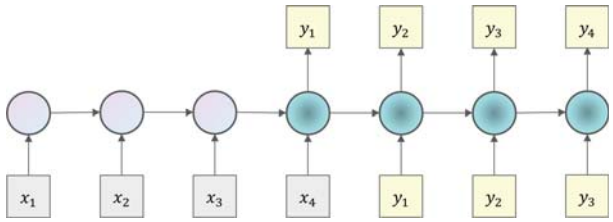


Fig. 3. The recurrent neural network and encoder-decoder architecture with one hidden layer. The circles represent neurons and the squares represent input and output. Encoder folds the data to retain information and decoder does the final task.

For real-time systems, the scheduling overhead is critical and should be much less than the tasks' execution times. Policies with lower overhead tend to be more practical. Thus, we first compare the floating-point operations (FLOPs) between the encoder-decoder and NN policy, assuming that the input size and output size are I and O, respectively. For simplicity and fairness, we suppose that the number of the hidden layer is one, and the neurons of the hidden layers H are identical. The FLOPs of the encoder-decoder for inferring priorities of J jobs are:

$$\begin{aligned} &FLOPs(encoder-decoder) \\ &= (I*H + H*H + H)*J*2 + (H*O)*J \\ &= (2*I*H + 2*H*H + H*O + 2*H)*J \end{aligned} \tag{7}$$

Moreover, the FLOPs of the NN for inferring priorities of J jobs are:

$$FLOPs(NN) = (I*H + H*O + H)*J \tag{8}$$

As we can see, the encoder-decoder policy has nearly three times the FLOPs of NN. Moreover, as the number of layers increases, the ratio grows faster. Meanwhile, NN policies have higher parallelism because they compute the priority for each job separately, and encoder-decoder policies must compute each neuron's output in sequence, as shown in Fig.3. If multiple processors can be used to compute priorities, the overhead will be greatly decreased. Hence, the overhead of encoder-decoder policies is much higher than NN policies.

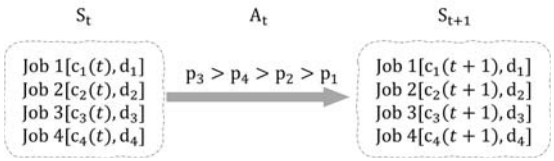
Since the MLP policy only uses the local parameter of one active job to determine the priority, one key drawback is that it cannot automatically extract the interrelation between all active jobs to help make decisions. The best way to make up is using some global information of all active jobs as input. Moreover, although RNN has relatively good robustness to input orders for sequences with small lengths, e.g., dozens, it is also hard to scale to hundreds of inputs, which is a regular size for job sets in real-time systems. Considering all these factors, we select the second solution, i.e., using the NN to represent the policy. The policy's input consists of the parameters of a particular job τ_i as well as some global information that can represent the job set. Empirically, minimum deadline, mean deadline, mean execution time, the maximum execution time of all active jobs, and the number of active jobs and processors are included. The RL policy is only invoked when a new job arrives to ensure the priority inference does not take up too much processor time.

For real-time scheduling, a deterministic policy is more suitable for priority inference than stochastic policy because it is the order of jobs' priorities that affects which jobs will be executed. Employing a stochastic policy will break the optimal order of priorities and make it unpredictable. Thus, the output of the policy NN is set to the priority of a corresponding job.

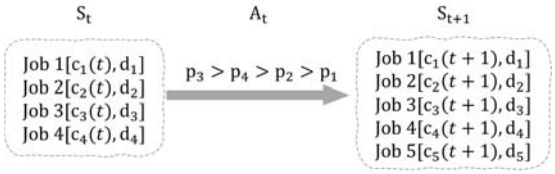
Since there are no explicit criteria for what NN structure should be used for what kind of application, the number of layers and the number of neurons in each layer of the policy NN for RL are often determined empirically. Recent work has proposed the neural architecture search (NAS) [28], which makes use of RL to automatically search the best NN structure that maximizes the accuracy for supervised learning. However, due to the high cost of online learning, NAS is difficult to use in RL tasks. The policy structure mainly affects the overhead of executing the policy, so lightweight NNs with shallow layers and few neurons are more suitable for real-time systems. On the other hand, lightweight NNs tend to be easier to train due to their fewer parameters. In this paper, we empirically design a lightweight NN structure with one hidden layer and eight neurons as the policy. Furthermore, the rectified linear unit (ReLU) [29] is applied for each hidden layer as the nonlinear activation function. The ReLU is expressed as $ReLU(x) = max(0, x)$, which has the lowest computational complexity among all activation functions.

B. MG Formulation

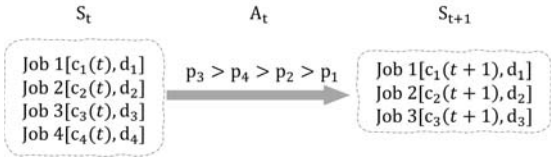
We intuitively regard each job as an agent because they follow a decentralized NN policy to determine the priority, respectively. Then we reformulate the real-time scheduling decision process as an MG. Each system state consists of n vectors, $s_t = (s_t^1, ..., s_t^n)$ where each component s_t^i is the state of the corresponding agent, i.e., active job τ_i. An agent's state is parameters of τ_i, such as remaining execution time and absolute deadline. The action of each agent is its priority. For each step, the transition system determines which jobs will be executed by their priorities and the number of processors. The remaining execution time of job i will be decreased one time unit if executed: $c_i(t+1) = c_i(t) - 1$. The job i will be removed from the job set if $c_i(t) = 0$ or $d_i = t$.



(a) Neither completed jobs nor released jobs at time $t+1$. $c_1(t+1) = c_1(t), c_2(t+1) = c_2(t), c_3(t+1) = c_3(t) - 1, c_4(t+1) = c_4(t) - 1$.$c_i(t+1) > 0$ and $d_i > t+1$ for each job i in the job set at time $t+1$.



(b) A new job will arrive if the arrival condition is met. $c_1(t+1) = c_1(t), c_2(t+1) = c_2(t), c_3(t+1) = c_3(t) - 1, c_4(t+1) = c_4(t) - 1$. $c_i(t+1) > 0$ and $d_i > t+1$ for each job i in the job set at time $t+1$.



(c) A current job misses or meets its deadline. It will be removed from the job set. If $job4$ misses its deadline, $d_4 = t+1$ and $c_4(t) > 1$. If $job4$ meets its deadline, $c_4(t) = 1$. $c_1(t+1) = c_1(t), c_2(t+1) = c_2(t), c_3(t+1) = c_3(t) - 1$. $c_i(t+1) > 0$ and $d_i > t+1$ for each job i in the job set at time $t+1$.

Fig. 4. MG transition of the real-time scheduling decision process.

We demonstrate the transitions of this MG by the examples given in Fig.4. The scheduling behavior of a multiprocessor system having two processors is shown in this figure. Three cases are depending on whether jobs are completed or new jobs arrive at time $t+1$. If there are no jobs completed or released at $t+1$, the two jobs having the highest priorities will be executed, as shown in Fig.4 (a). Meanwhile, the remaining execution time of job three and job four is reduced by a time step. Fig.4(b) shows a special case that a new job instance is released. In the above cases, all jobs' deadlines are later than $t+1$, and the remaining execution times of job three and job four are larger than 1. In the third case, there is a certain job completed. The job i meets the deadline if $c_i(t+1) = 0$, or it misses the deadline if $c_i(t+1) > 0$ and $d_i = t+1$. The completed jobs will be removed from the job set at time $t+1$, as Fig.4(c) shown.

The goal of reinforcement learning is to maximize the expectation of cumulative rewards $\mathbb{E}(R_0)$. Therefore, the performance of learned policy depends largely on the reward function. Instead of explicitly designing a reward function by domain knowledge, we derive the reward function by objective function transformation from real-time scheduling to reinforcement learning to ensure the performance of learned policies. The goal of real-time scheduling is to schedule jobs to make as many jobs as possible meet their deadlines. Therefore, an ideal learning algorithm should learn a policy π that can maximize the expectation of the success ratio. Suppose that SJ is the set of jobs that meet their deadline, and FJ is the set of jobs that miss their deadlines. The total number of jobs that meet their deadlines is $|SJ|$, and the total number of jobs that miss their deadlines is $|FJ|$. TS is all possible initial task sets for a specific real-time system. The objective of real-time scheduling is to find a scheduling policy π that

$$\max_{\pi} \mathbb{E}_{TS}\left(\frac{|SJ|}{|SJ| + |FJ|}\right) \tag{9}$$

The above objective is equivalent to maximize $|SJ|$ and minimize $|FJ|$ simultaneously, which can be rewritten as:

$$\max_{\pi} \mathbb{E}_{TS}(|SJ| - |FJ|) \tag{10}$$

We discretize the scheduling process by processor time unit, i.e., a discrete and indivisible unit of time determined for a specific processor. For any time t, let $|SJ_t|$ denotes the number of jobs whose completion times equal to t that meet their deadlines, and $|FJ_t|$ denotes the number of jobs whose absolute deadlines are t that miss their deadlines. The objective can be rewritten by $|SJ_t|$ and $|FJ_t|$:

$$\max_{\pi} \mathbb{E}_{TS}\left(\sum_t |SJ_t| - |FJ_t|\right) \tag{11}$$

To ensure the RL policy can maximize the success ratio, we derive a reward function of each time instant as below:

$$r_t = |SJ_t| - |FJ_t| \tag{12}$$

Then, the objective function of real-time scheduling is transformed to the objective function of reinforcement learning, as shown below:

$$\max_{\pi} \mathbb{E}_{TS}\left(\sum_t r_t\right) \tag{13}$$

Hence, the objective function of real-time scheduling is transformed to the objective function of reinforcement learning through the reward function. With this reward function, reinforcement learning can generate high-performance scheduling policies that maximize the success ratio among the possible initial task sets for a specific real-time system.

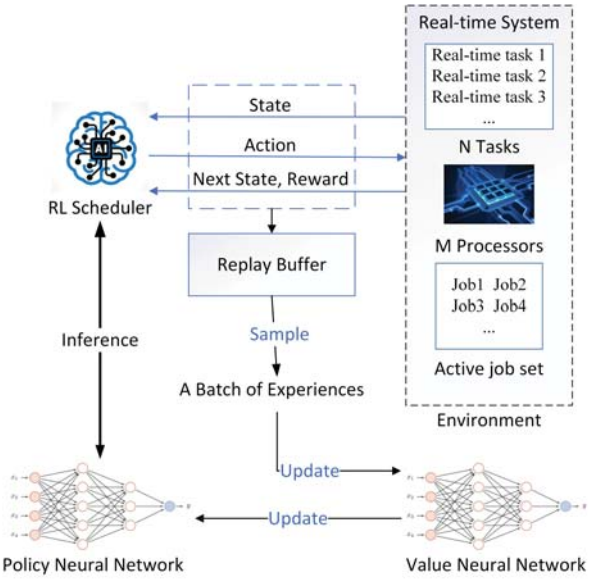


Fig. 5. The framework of our approach. The RL scheduler obtains the experiences by interacting with a real-time system. It stores the experience in a replay buffer and learns from the sampled data. Scheduling policy will be improved through continuous interaction and learning.

C. The framework of Learning Scheduling Policy

The framework shows the online learning process for scheduling policy. As shown in Fig.5, the environment is a real-time system with M processors and N tasks. There is also an active job set with n jobs. Similar to global schedulers, the RL scheduler/policy generates the priority for all active jobs and selects M jobs with higher priorities to execute. The scheduling experience is stored in the form of 4 tuples (s_t, a_t, r_t, s_{t+1}) in the replay buffer. We extend the actor-critic architecture to learn policy and train double NNs to approximate policy and Q function. The learning process can be divided into two steps: collecting samples and updating the neural networks' parameters.

(1) Collect samples: During system runtime, the RL scheduler observes state information from the environment at each time. And then, the policy neural network is invoked for priority inference. As mentioned before, the action of the RL scheduler is priorities of current jobs. After receiving an action from the RL scheduler, the environment selects the top M jobs with the highest priority to execute. After executing, the RL scheduler receives the next state and reward from the environment. The above process is called a complete interaction between the agents and the environment. Experiences from each interaction will be stored in a replay buffer. There is usually a maximum size of replay buffer. When the buffer is full, old experiences will be discarded.

(2) Update network: Agents learn from the experiences stored in the replay buffer. Each time, a batch of experiences is sampled from the replay buffer and used to train policy neural networks and value neural networks. The value NN is often refined by mean square error (MSE) as the loss function, firstly.

$$MSE = \frac{1}{M}\sum_{m=1}^{M}(y_m - \hat{y}_m)^2 \tag{14}$$

And then, the parameters of the policy neural network is updated in order to maximize the objective $J(\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[\sum r(s,a)]$ by taking steps in the direction of $\nabla_\theta J(\theta)$. We will discuss how to learn the policy in detail in the following subsection.

The above two steps are independent and can be executed in parallel.

Since periodic tasks can be regarded as aperiodic tasks with a fixed interval, the framework is also suitable for periodic tasks or uniprocessor systems. Furthermore, it is noted that our reinforcement learning framework has good potential to be extended to real-time systems with various characteristics. For example, if the systems that execute real-time tasks having weights or values, a weighted reward function could be used to learn scheduling policy. For real-time tasks with precedence constraint, usually represented by a directed acyclic graph, the graph neural network could be employed to compute priorities for such task sets and train it by reinforcement learning approach.

D. Self-cooperative Learning

In our multi-agent setting, there are n homogeneous agents cooperating to get the job done in the environment. At each step t, they obtain a global reward r_t as a function of the state and agents' action $r_t = \mathbb{E}[r(s_t, a_t^1, ..., a_t^n)]$. These n agents cooperate with each other to maximize the joint cumulative expected return $\mathbb{E}[R_0]$. So this is a cooperative game for the job agents [30].

In most multi-agent environments, the number of agents is often fixed. However, the number of jobs keeps changing in the real-time system. The input of centralized Q function is a set of jobs' states and actions $(s^1, a^1, s^2, a^2, ..., s^n, a^n)$ which is a variable-length sequence. Besides, the agents are homogeneous, so that the centralized Q-value should be the same no matter what input order is. In other words, the centralized Q function must be invariant to input permutation, as shown in formula (15):

$$Q(..., s^i, a^i, ..., s^k, a^k, ...) = Q(..., s^k, a^k, ..., s^i, a^i, ...) \tag{15}$$

In order to meet both of the above requirements, we use the symmetric function to aggregate the information from each point. The symmetric function takes n vectors as input and outputs a new vector invariant to the input order. For example, $+$ and $*$ operators are symmetric binary functions [31]. We propose Self-Cooperative Learning (SCL) in this paper. The main idea of SCL is to approximate a centralized Q function by applying a symmetric function on decentralized Q function, $\dot{Q}$ for each agent, and the centralized Q function is defined as:

$$Q(s^1, a^1, ..., s^n, a^n) = g(\dot{Q}(s^1, a^1), ..., \dot{Q}(s^n, a^n)) \tag{16}$$

Algorithm 1 Learning scheduling policy by self-cooperative reinforcement learning

Randomly initialize critic networks $Q(s,a|\theta^Q)$ and actor network $\pi(s|\theta^\pi)$ with weight θ^Q and θ^π as well as corresponding target networks.
Initialize replay buffer B
Initialize a Gaussian random process $\mathcal{N}$ and a exploration episode EP for action exploration
for episode = 1, MAX_EPISODE **do**
    Reset environment
    Randomly generate a task set
    Receive initial observation state s_0
    for t = 1 ,MAX_STEP **do**
        if episode $<$ EXPLORE_EPISODE **then**
            Sample a random action $a_t \sim \mathcal{N}$ for the job set
        else
            Select an action $a_t = \pi(s_t)$ from current policy for the job set
        end if
        Execute action a_t and observe reward r_t and new state s_{t+1}
        Store experience (s_t, a_t, r_t, s_{t+1}) in B
        Sample a batch of experiences $\mathcal{D}$ from B
        Set $y = r + \gamma \frac{1}{N} \sum_i \dot{Q}'(s^{i\prime}, \pi'(s^{i\prime}|\theta^{\pi'})|\theta^{Q'})$
        Update critic by minimizing the loss:$\mathcal{L}(\theta^Q) = \mathbb{E}_{s,a,r,s'\sim\mathcal{D}}[(y - \frac{1}{N}\sum_i \dot{Q}(s^i, a^i|\theta^Q))^2]$
        Update the actor policy using the sampled gradient:
$$\nabla_{\theta^\pi} J(\theta^\pi) = \mathbb{E}_{s\sim\mathcal{D}}[\frac{1}{N}\sum_i \nabla_{\theta^\pi}\pi(s^i|\theta^\pi)\nabla_{\pi(s^i|\theta^\pi)}\dot{Q}(s^i, \pi(s^i|\theta^\pi)|\theta^Q)]$$
        Update the target networks:
$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\pi'} \leftarrow \tau\theta^\pi + (1-\tau)\theta^{\pi'}$$
    end for
end for

The decentralized Q function, $\dot{Q}$ is state-action value function for a single agent in the environment. Although we regard the scheduling process as a multi-agent setting, agents essentially take the same policy π. In our setting, both the centralized Q function and decentralized Q function are only w.r.t. π. Therefore, it is reasonable to represent the centralized Q function by decentralized Q function in our setting.

Since the agents only receive a global reward from the environment and the global reward is incurred by all agents' actions, we cannot directly train the decentralized Q function by global reward. In multi-agent cooperative tasks, it is hard to assign the global reward to each agent impartially. That is the credit assignment problem.

To overcome the credit assignment problem, we try to express the centralized and decentralized Q function by the global reward rather than the local reward. Suppose r_t^i is the local reward of agent i at time t, we have

$$r_t = \mathbb{E}_i[r_t^i] \tag{17}$$

The decentralized Q function for agent i decomposes into the Bellman equation (t is omitted):

$$\dot{Q}(s^i, a^i) = r^i + \gamma\mathbb{E}_{s^{i\prime}\sim P}[\dot{Q}'(s^{i\prime}, \pi'(s^{i\prime}))] \tag{18}$$

Our approach also makes use of a target network, as DDPG and DQN. Sum over all the decentralized Q functions at the time t and then divide by the number of jobs, we have:

$$\begin{aligned} Q(s,a) &= \frac{1}{N}\sum_i \dot{Q}(s^i, a^i) \\ &= r + \gamma\frac{1}{N}\mathbb{E}_{s^{i\prime}\sim P}[\sum_i \dot{Q}'(s^{i\prime}, \pi'(s^{i\prime}))] \end{aligned} \tag{19}$$

Thus, decentralized Q function can be represented by global reward r. And we approximate $\dot{Q}(s,a)$ by a NN and g by a mean pooling function which is invariant to input permutation. The parameter θ^Q for value neural network is updated through sampling a batch of experiences $\mathcal{D}$ from the replay buffer. The loss function is:

$$\begin{aligned} &\mathcal{L}(\theta^Q) = \mathbb{E}_{s,a,r,s'\sim\mathcal{D}}[(y - \frac{1}{N}\sum_i \dot{Q}(s^i, a^i|\theta^Q))^2], \\ &where\ y = r + \gamma\frac{1}{N}\sum_i \dot{Q}'(s^{i\prime}, \pi'(s^{i\prime}|\theta^{\pi'})|\theta^{Q'}), \end{aligned} \tag{20}$$

The gradient of loss function is:

$$\nabla_{\theta^Q}\mathcal{L}(\theta^Q) = \mathbb{E}_{s,a,r,s'\sim\mathcal{D}}[(y - \frac{1}{N}\sum_i \nabla_{\theta^Q}\dot{Q}(s^i, a^i|\theta^Q))^2], \tag{21}$$

The objective of policy neural network with parameter θ^π is:

$$J(\theta^\pi) = \mathbb{E}_{s\sim\mathcal{D}}[\frac{1}{N}\sum_i \dot{Q}(s^i, \pi(s^i|\theta^\pi)|\theta^Q)], \tag{22}$$

Gradient of objective can be written as:

$$\nabla_{\theta^\pi} J(\theta^\pi) = \mathbb{E}_{s\sim\mathcal{D}}[\frac{1}{N}\sum_i \nabla_{\theta^\pi}\pi(s^i|\theta^\pi)\nabla_{\pi(s^i|\theta^\pi)} \dot{Q}(s^i, \pi(s^i|\theta^\pi)|\theta^Q)], \quad (23)$$

Moreover, instead of directly copying weights, target networks use "soft" updates [32]. A copy of the networks for both the actor and critic are created, denoted by $Q'(s,a)$ and $\pi'(s,a)$ respectively, but their weights are updated by slowly tracking the learned network, as shown in (24). The use of target networks helps improve the stability of the learning by constraining the weights to change slowly.

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \\ \theta^{\pi'} &\leftarrow \tau\theta^\pi + (1-\tau)\theta^{\pi'} \end{aligned} \quad (24)$$

The complete learning algorithm is shown in algorithm 1. In order to accelerate training, We randomly perform actions to collect samples in the first few episodes.

IV. Evaluation

In this section, we evaluate the performance of the proposed RL approach using randomly generated task instance sets. Specifically, we first train RL policies using self-cooperative learning under different settings and then evaluate RL policies and baselines with different metrics.

A. Simulation Setup and Metrics

Compared with CPU, GPU is more suitable for neural networks in training and inference. However, most real-time systems do not have any GPU resources. Hence, all schedulers only use the CPU to make scheduling decisions in the experiments. The platform used to collect the experimental data is a Windows desktop with Intel Core i7-6700 3.40GHz CPU with eight cores, 16GB of RAM.

More than 10 thousand randomly generated task sets are used for training and evaluating the scheduling policy. Specifically, we generate aperiodic task loads with exponentially distributed arrivals, exponentially distributed execution times, and uniformly distributed deadlines. The rate parameter for the exponential arrival interval varies from 12 to 25 to help generate different workloads. This rate parameter is the expected value of the job arrival interval. The relative deadlines follow uniform distribution $U[5, 50]$. For each randomly generated task T_i, its execution time, C_i follows exponential distributions whose rate parameter is set according to task T_i's granularity. Here T_i's granularity is defined as the ratio of its execution time and relative deadline, i.e., C_i/D_i, which vary from 0.1 to 0.6. The number of tasks N is five times the processor number M, and each task has at least 20 job instances to make sure that each experiment is carried for a sufficiently long time. The preemption and migration overhead is ignored if not explicitly declared.

We use the success ratio to evaluate the performance of learned scheduling policies and baseline heuristics. The success ratio is defined as the percentage of jobs that meet their deadlines. We also evaluate the execution overhead of the RL scheduler and baseline scheduler, using the time complexity and time overhead as metrics.

B. Training Policies

To speed up training, we carry our training experiments on a Ubuntu 16.04 server with Intel Xeon E7-4809 2.10GHz CPU, 2 NVIDIA Tesla P100 GPU. The use of GPU is to accelerate training. PyTorch [14], an open-source Python machine learning library, is used for constructing and updating neural networks. The hyper-parameters for training are shown in TABLE I.

TABLE I
HYPER-PARAMETERS

Parameter	Value
Memory capacity of replay buffer	1e6
Batch size	100
γ	0.95
τ	1e-3
Explore episodes	30
Learning rate for policy NN	1e-3
Learning rate for value NN	1e-2
Number of hidden layers of value NN	2
Number of neurons in each layer of the value NN	8, 4

The task instance sets for training are randomly generated, following the pattern described in the previous subsection. We train the scheduling policy based on $\boldsymbol{Algorithm1}$ until convergence. (at least 200 episodes).

C. Success Ratio Evaluation

In this subsection, we investigate the performance of learned scheduling policy in terms of success ratio at first. Specifically, we compare the success ratio between RL policy with three heuristic scheduling policies, G-EDF, EDF-BF, and LSF. For the simulation data to be presented, each data point on each curve in the figures is the average of 50 experiments, each carried out for a sufficiently long time (more than 1000 job arrivals). For fairness, in each experiment, the same random job sets are used to evaluate different scheduling policies.

We first evaluate the success ratio over different loads and task granularities. The load is defined as the sum of actual computation times of all the arriving jobs over the time interval during which they arrive.

As seen from Fig.6, our learned policy has the best overall success ratio comparing with the baselines. These simulation data are collected for different incoming loads. The above result shows that our RL approach makes policies converge to the high success ratio. At low loads, as expected, nearly all jobs meet their deadlines no matter which policy is used. As the load increases, more and more jobs miss their deadlines, the success ratios of all policies decrease as expected. Although heuristics have a high success ratio in low loads, their performance decreases rapidly when the load exceeds 0.9. The success ratio of RL policy decreases much more slowly

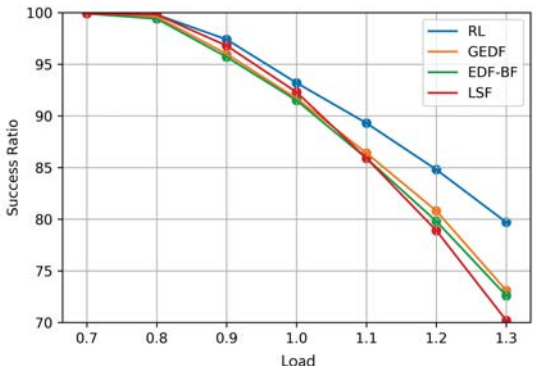


Fig. 6. Success ratio under different system loads for reinforcement learning policy and heuristic policies. The granularity is less than 0.4. N = 40, M = 8

and consistently obtains high success ratios for different loads. Only the success ratio of RL policy remains above 80% when the load increases to 1.3.

Generally, it is hard to schedule task sets with large granularity tasks. To further verify the robustness of the RL method, we test some extraordinary task sets that contain both small granularity tasks and large granularity tasks. In these cases, the granularities of most tasks are more than 0.4. From Fig.7, one can see that RL policy achieves better overall performance than other heuristics. Compared to the performance under low task granularity, the performance gaps between RL policy and other policies grow more rapidly as the load increases. The success ratio of RL policy is $5\% \sim 10\%$ higher than the best heuristic policy when the load exceeds 0.9.

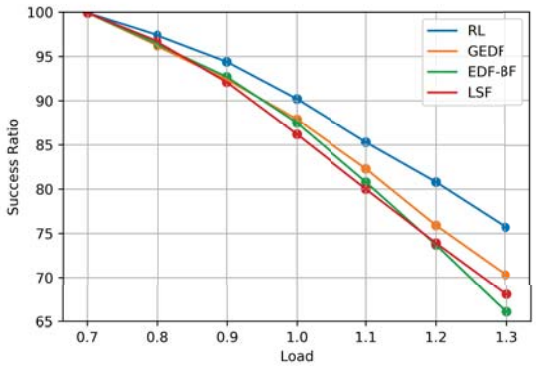


Fig. 7. Success ratio under different system loads for reinforcement learning policy and heuristic policies. The granularity of tasks is large than 0.4. N = 40, M = 8

Then, we evaluate the success ratio over different numbers of processors, as shown in Fig.8. For this set of simulations, the granularity of tasks is less than 0.4, and the system loads are about 1. As the number of processors increases, the success ratio gets closer to 100. These results show that our RL approach can learn sound policies for real-time systems with different numbers of processors, system loads, or task granularity.

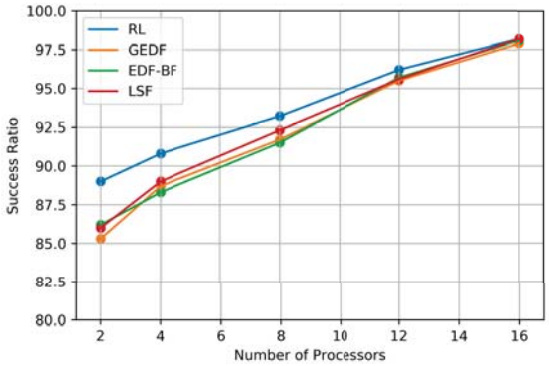


Fig. 8. Success ratio under different number of processors for reinforcement learning policy and heuristic policies. The average granularity of all tasks is 0.3.

In the real world, the preemption overhead is ubiquitous. Some real-time systems may have a non-negligible preemption overhead. To verify the applicability of the RL approach for such systems, we test the RL approach on a specific system with a time step preemption overhead. Besides, a new feature that indicates whether the job is running on a processor must be added to NN's input as a crucial characteristic. In Fig.9, we show the learning process of the RL policy and use GEDF to schedule the same task sets in each episode as the baseline. During the learning process, the success ratio of the RL policy increases gradually and exceeds the baseline in about 80 episodes. Experimentally, the proposed RL approach has good convergence and scalability for real-time systems with preemption overhead. For other real-time systems with various characteristics, the RL approach can also learn high-performance policy online.

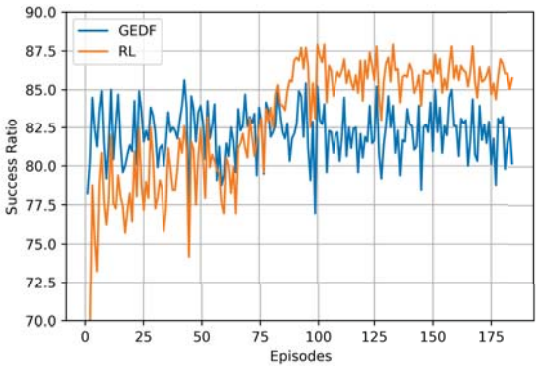


Fig. 9. Learning process on a system with one time step preemption overhead.

D. Overhead Evaluation

Another important consideration in evaluating scheduling policies is their actual runtime overhead. Generally, it is hard

to measure the actual runtime overhead for baseline heuristics because they only use simple formulas for computing priorities. So, we analyzed and compared the time complexity and FLOPs for RL schedulers and baseline heuristics. Simple heuristic schedulers compute the priority for all jobs and do a binary search or sort to prioritize all active jobs. The time complexity could be $O(logN)$ for GEDF and $O(N)$ for LSF, where N is the number of active jobs. In the RL method, the scheduler computes the priorities of all jobs by invoking the policy neural network N times and then sort the priorities. The time complexity of RL schedulers could be $O(N)$, which is not lower than simple heuristics. Moreover, baseline heuristics always compute priorities based on a simple formula, e.g., $D_i - C_i$ in LSF. Their FLOPs for each job are approximately one. However, RL schedulers use policy NN, a relatively complicated method to compute priorities. The FLOPs of RL schedulers shown in section III.A are larger than baselines.

Although baseline schedulers have relatively lower overheads than RL schedulers, RL policies' overheads are also tolerable in many computationally demanding tasks (applications). On the other hand, their overheads could be further reduced by using simpler approximators or parallel computing. To demonstrate the parallelism of RL schedulers, we then test the actual overhead time of RL policies under different numbers of jobs in the multiprocessor system. As shown in Fig.10, the learned scheduling policy's total overhead stays at a relatively low, stable level. Specifically, the average overheads only increase 50 microseconds as the job number increase from 10 to 500. The overheads do not increase linearly because the policy neural network is computed in parallel by a multiprocessor. Furthermore, some applications would prefer high-performance RL schedulers rather than baseline schedulers having lower overheads. For example, in avionics and automotive control applications, jobs usually require hundreds of thousands or millions of cycles to execute. Hence, RL policies are more suitable for many applications.

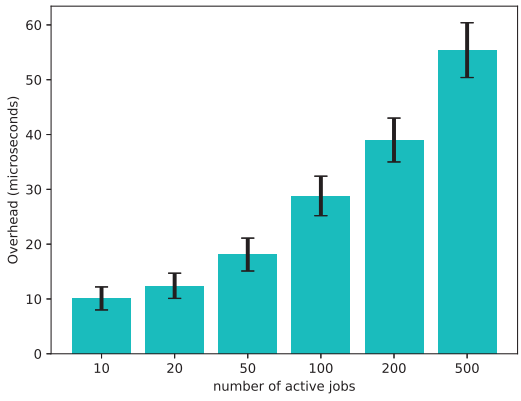


Fig. 10. Minimum, mean and maximum overheads under different number of jobs. We test 10 thousand different data for each number of jobs in a 8-core processor system.

In this section, we have verified the effectiveness of multi-agent self-cooperative learning through simulation results. One can be readily seen that the multi-agent self- cooperative learning can learn an acceptable scheduling policy for various task/system models. The learned scheduling policy has low, stable overhead and high scheduling performance, making it strongly desired by real-time multiprocessor systems that deal with unpredictable frequent task instance arrivals.

V. Use Case

In this section, we demonstrate how the RL approach in this paper might be realistically applied. A prospective application is the Unmanned Vehicle System (UVS), which attracts significant interest in the critical embedded market. Supporting advanced UVS functionalities requires real-time systems capable of performing real-time processing of a massive amount of diverse data, consistently coming from a score of on-board sensors, such as LiDARs and IMU [33]. The real-time tasks in UVS consist of perception, control, localization, planning, prediction. An intuitive example is shown in TABLE II.

TABLE II
An example of real-time tasks in UVS

Task	Execution Time (ms)	Deadline (ms)	Period (ms)
Perception	150	300	500
Control	8	20	Aperiodic
Localization	10	50	Aperiodic
Planning	200	500	500
Prediction	150	400	1000

First, to learn a high-performance scheduling policy, one should select an approximator with appropriate complexity as the scheduling policy based on the system's computational resources. The input to the policy must include all task characteristics that can help prioritize. Then, the RL algorithm runs the scheduler to collect samples during system runtime and updates the policy using the samples until convergence. Since the RL approach is learning from real systems, it must lead to scheduling policies that are stronger than simple heuristics. Users can also try different hyper-parameters and approximators, then select the best policy. If the system changes in the future, users can use RL again to improve the policy.

VI. Related Work

Glaubius et.al. [34] has represented the real-time scheduling decision model on the open real-time system as an MDP, and suitable scheduling policies are learned online using reinforcement learning [35]. In open real-time systems, jobs are released periodically. Whenever a job is granted the resource, it occupies the resource for a finite and bounded subsequent duration. The duration for which a job occupies the resource may vary from run to run of the job released by the same task, but overall obeys a known independent and bounded distribution over any reasonably large sample of runs of that task. The objective of scheduling is to achieve near-optimal schedules to maintain relative utilization of shared

resources among tasks near a user-specified target level. Since their method is based on a traditional non-deep reinforcement learning algorithm, it has poor convergence and only computes static scheduling forms for given tasks. Hence, it is not easy to extend their work to learn a robust dynamic scheduling policy for multiprocessor systems executing aperiodic tasks.

VII. Summary, Discussion and Future Work

The traditional design pattern of real-time scheduling policy is in a dilemma. In this paper, we investigate a new real-time scheduling framework based on reinforcement learning. We formulate the real-time decision model as MG and derive the reward function from the objective function of real-time scheduling. Specifically, we introduce multi-agent self-cooperative learning, a novel reinforcement learning approach that solves the credit assignment problem and mitigates the curse of dimensionality. The main idea of self-cooperative learning is to use the symmetric function to aggregate each job's value function to approximate the centralized value function. We implement a real-time scheduling simulator and the proposed reinforcement learning algorithm. Simulation results show that the proposed RL approach can train a well-performance scheduler. Since the learned RL policy has low, stable overhead and high performance, it would also be strongly desired by the real-time systems that deal with unpredictable frequently-changing job arrivals. The safety of our approach can be guaranteed by equipping an admission controller that avoids online deadline misses.

In the future, we will optimize our method in generalization and convergence by using other learning algorithms or function approximators. As another follow-up work, we intend to extend our work to help the offline design of hard real-time multiprocessor systems through automatically learning scheduling policy and testing system safety (feasibility). We also intend to explore learning scheduling policies for real-time systems with complex characteristics. In this paper, we have only considered using fixed structure models for learning a high-performance policy. We will explore how to learn both optimal and low-overhead policies simultaneously.

Acknowledgment

This research was supported by the National Key R&D Program of China (No. 2018YFB0904500). This research was also supported by the National Natural Science Foundation of China (No. 61902383), and the Youth Innovation Promotion Association CAS (No. 2021103), and Key Technology and Integrated Application Research of Smart Stadiums by National Speed Skating Stadium (Z181100005918007).

References

[1] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations research*, vol. 26, no. 1, pp. 127–140, 1978.

[2] O. U. P. Zapata and P. M. Alvarez, "Edf and rm multiprocessor scheduling algorithms: Survey and performance evaluation," *Seccion de Computacion Av. IPN*, vol. 2508, 2005.

[3] J. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS*, Jun. 2000, pp. 35–43, iSSN: 1068-3070.

[4] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[5] S. Baruah, J. Gehrke, and C. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *Proceedings of 9th International Parallel Processing Symposium*, Apr. 1995, pp. 280–288.

[6] A. Burchard, J. Liebeherr, Y. Oh, and S. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1429–1442, Dec. 1995.

[7] S. Kato and N. Yamasaki, "Semi-partitioning technique for multiprocessor real-time scheduling," *migration*, vol. 1, p. P2, 2008.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016. [Online]. Available: http://www.nature.com/articles/nature16961

[9] Y. Li, "Deep reinforcement learning: An overview," *arXiv:1701.07274 [cs]*, Nov. 2018. [Online]. Available: http://arxiv.org/abs/1701.07274

[10] A. Goldie and A. Mirhoseini, "Placement Optimization with Deep Reinforcement Learning," in *Proceedings of the 2020 International Symposium on Physical Design*. Taipei Taiwan: ACM, Mar. 2020, pp. 3–7. [Online]. Available: https://dl.acm.org/doi/10.1145/3372780.3378174

[11] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3389–3396.

[12] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[13] G. A. Rummery, "Problem solving with reinforcement learning," Ph.D. dissertation, University of Cambridge Ph. D. dissertation, 1995.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[15] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[16] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[17] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *arXiv:1602.01783 [cs]*, Jun. 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[18] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv:1707.06347 [cs]*, Aug. 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[20] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv:1509.02971 [cs, stat]*, Jul. 2019. [Online]. Available: http://arxiv.org/abs/1509.02971

[21] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 157–163. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9781558603356500271

[22] Lucian Buoniu, Robert Babuka, and B. D. Schutter, *Multi-agent Reinforcement Learning: An Overview*. Springer Berlin Heidelberg, 2010.

[23] L. Busoniu, R. Babuska, and B. De Schutter, "A Comprehensive Survey of Multiagent Reinforcement Learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 2, pp. 156–172, Mar. 2008.

[24] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Advances in neural information processing systems*, 2017, pp. 6379–6390.

[25] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[26] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1724–1734.

[27] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *International Conference on Learning Representations*, 2015.

[28] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," *arXiv:1611.01578 [cs]*, Feb. 2017, arXiv: 1611.01578. [Online]. Available: http://arxiv.org/abs/1611.01578

[29] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323.

[30] X. Chu and H. Ye, "Parameter Sharing Deep Deterministic Policy Gradient for Cooperative Multi- agent Reinforcement Learning," *arXiv preprint arXiv:1710.00336*, p. 12, 2017.

[31] R. Q. Charles, H. Su, M. Kaichun, and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, Jul. 2017, pp. 77–85. [Online]. Available: http://ieeexplore.ieee.org/document/8099499/

[32] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[33] J. Van Brummelen, M. O'Brien, D. Gruyer, and H. Najjaran, "Autonomous vehicle perception: The technology of today and tomorrow," *Transportation research part C: emerging technologies*, vol. 89, pp. 384–406, 2018.

[34] R. Glaubius, T. Tidwell, W. D. Smart, and C. Gill, "Scheduling Design and Verification for Open Soft Real-Time Systems," in *2008 Real-Time Systems Symposium*, Nov. 2008, pp. 505–514.

[35] R. Glaubius, T. Tidwell, C. Gill, and W. D. Smart, "Real-time scheduling via reinforcement learning," in *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*. AUAI Press, 2010, pp. 201–209.