DeepScheduler: Enabling Flow-Aware Scheduling in Time-Sensitive Networking

Xiaowu He*, Xiangwen Zhuge*, Fan Dang†, Wang Xu† and Zheng Yang*✉
* School of Software and BNRist, Tsinghua University † Global Innovation Exchange, Tsinghua University
horacehxw@gmail.com, zgxw18@mails.tsinghua.edu.cn, dangfan@tsinghua.edu.cn,
wangxu.93@hotmail.com, hmilyyz@gmail.com

IEEE INFOCOM 2023 - IEEE Conference on Computer Communications | 979-8-3503-3414-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/INFOCOM53939.2023.10228875

Abstract*—Time-Sensitive Networking (TSN) has been considered the most promising network paradigm for time-critical applications (*e.g.*, industrial control) and traffic scheduling is the core of TSN to ensure low latency and determinism. With the demand for flexible production increases, industrial network topologies and settings change frequently due to pipeline switches. As a result, there is a pressing need for a more efficient TSN scheduling algorithm. In this paper, we propose DeepScheduler, a fast and scalable flow-aware TSN scheduler based on deep reinforcement learning. In contrast to prior work that heavily relies on expert knowledge or problem-specific assumptions, DeepScheduler automatically learns effective scheduling policies from the complex dependency among data flows. We design a scalable neural network architecture that can process arbitrary network topologies with informative representations of the problem, and decompose the problem decision space for efficient model training. In addition, we develop a suite of TSN-compatible testbeds with hardware-software co-design and DeepScheduler integration. Extensive experiments on both simulation and physical testbeds show that DeepScheduler runs >150/5 times faster and improves the schedulability by 36%/39% compared to state-of-the-art heuristic/expert-based methods. With both efficiency and effectiveness, DeepScheduler makes scheduling no longer an obstacle towards flexible manufacturing.

***Index Terms*—Time-Sensitive Networking, Traffic Scheduling, Cyber-physical System, Deep Reinforcement Learning.**

I. Introduction

Time-Sensitive Networking (TSN) has been considered the most promising network paradigm for time-critical applications like industrial control and automotive [1]. It successfully bridges the gap between Information Technology (IT) and Operational Technology (OT) by including real-time capabilities in the Ethernet standard [2], making it a one-size-fits-all solution. The real-time capabilities come from the IEEE 802.1Qbv standard [3]. It introduces a time-aware traffic shaper and enables the network switches to send each packet following a predetermined cyclic timetable.

Traffic scheduling is critical, especially for TSN. All switches in a TSN network must synchronize their clocks and forward data frames cooperatively according to a well-designed global schedule to ensure low latency and determinism. However, deriving such a schedule is not trivial. It can be equivalent to a constrained time slot assignment process on each physical link (detail in Sec. II-B) and is an NP-hard combinatorial optimization problem. Typical solutions like integer linear programming (ILP)-based or satisfiability modulus theories (SMT)-based solvers take several hours or even days to yield a valid result, which is tolerable for traditional factories whose network topology is stable.

Unfortunately, as the demand for flexible manufacturing grows, traffic scheduling in industrial networks becomes a formidable obstacle. According to our investigation of a top-tier auto glass manufacturer, a single production line can produce up to 10 distinct types of auto glass per day. Different types of auto glass necessitate distinct pipelines, such as those with or without coatings, heaters, *etc.*, and they require different inspection procedures. Currently, there are approximately 80 to 120 interconnecting devices per pipeline. Switching a pipeline takes 10 minutes to change the mold and about 40 minutes to reconfigure the network schedule and manufacturing settings like the heating curve. As automotive vendors place higher demands on workflow and inspection procedures, more equipment is being installed in production lines. Therefore, scheduling has become a growing burden on production flexibility and efficiency.

To accelerate scheduling, some researchers have adopted heuristic searches like Tabu-search [4] or genetic algorithm [5] in TSN scheduling. Others utilize domain-specific knowledge (DSK) to prune the algorithm's search space [6] or simplify the problem settings [7]. Most of them sacrifice the schedulability of algorithms for a faster run time by only exploring a subset of the whole solution space. Despite meaningful and enlightening attempts, these methods heavily rely on expert experience and problem-specific assumptions, necessitating a redesign if the assumptions or data distribution change slightly.

We observe that though the scheduling problem is difficult, it is significantly simpler to confirm the correctness, or satisfiability, of a potential schedule. This leaves an opportunity to conduct a series of trial-and-error processes and derive better search policies from implicit data distributions with the reinforcement learning (RL) framework. Recently, a number of studies have successfully applied deep learning and RL to tackle NP-hard combinatorial optimization problems (*e.g.*, traveling salesperson problem [8], [9], bin packing [10], [11]).

Nevertheless, due to the TSN scheduling problem's unique characteristics, applying DRL to it brings about the following challenges. **(1) Dynamic network topologies.** Traditional deep learning modules require fixed-size feature vectors as input, while the size and topology of the network vary case

✉ Zheng Yang is the corresponding author.

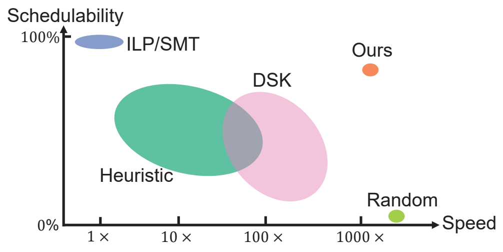


Fig. 1: A comparison of different types of schedulers.

by case. It is difficult to encode the node-link relationships with these modules due to limited scalability. Thus, we need to design a scalable neural network architecture capable of adapting to any network topology. **(2) Complex dependencies among data flows.** Each data flow in the network is shaped by user-specific requirements like source/destination nodes, maximum delay, *etc.*, and also other flows (approximated by the current network status). It is difficult to obtain a meaningful hidden representation capturing the dependency between the vector-based requirements and graph-based network status. Consequently, it is necessary to propose an effective fusion framework between the flow requirements and network status. **(3) Exponentially large decision space.** The TSN scheduling problem requires processing hundreds of data flows simultaneously, and each of them has tens of links and thousands of time slots to select. Its decision space is significantly larger than that of conventional RL applications (*e.g.* robot arm control, game agent control). Therefore, a rational RL framework needs to be designed to help the model learn an effective policy.

In this paper, we present DeepScheduler, a fast and scalable TSN scheduler based on DRL. To deal with the above challenges: (1) We propose a scalable state information encoder based on the graph neural network (GNN), utilizing the graph structure to model the complex relationship between network topology and link states adaptively. (2) We propose a path-based flow-aware encoder that combines the current network status and the flow requirements from the perspective of their possible routes. (3) We decompose the decision space of scheduling into two interdependent sub-tasks, reducing the problem size for each RL agent and allowing the model to efficiently explore the entire search space. Moreover, we propose hard sample mining and incremental training techniques to stabilize the model training process.

The main contributions are as follows:

- DeepScheduler is, as far as we are aware, the first DRL framework that globally solves the TSN scheduling problem. Compared with the SOTA methods (as qualitatively shown in Fig. 1), it runs at the second level ($>$1000/150/5 times faster than ILP/Heuristic/DSK) and adapts to various scenarios (36%/39% schedulability improvement over Heuristic/DSK), making traffic scheduling no longer an obstacle towards flexible manufacturing.
- We present a scalable neural network architecture with a path-based flow-aware encoder and a GNN-based network encoder. It can process arbitrary sizes and topologies of networks efficiently and effectively, and learns informative representations from the complex dependencies between heterogeneous flow requirements and network status.
- We have conducted experiments on hundreds of topologies and different problem sizes to validate DeepScheduler's adaptiveness and reliability. In addition, we develop a suite of TSN-compatible testbeds with hardware-software co-design and integrate DeepScheduler in the network controller. The successful operation on a real-world topology demonstrates DeepScheduler's practicality.

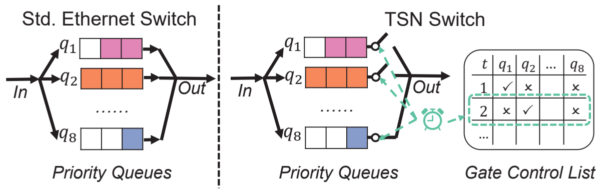
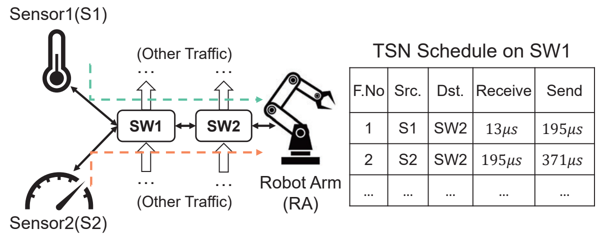


Fig. 2: Standard Ethernet switch vs. TSN switch.



F.No	Src.	Dst.	Receive	Send
1	S1	SW2	$13\mu s$	$195\mu s$
2	S2	SW2	$195\mu s$	$371\mu s$
...	...	...	...	...

Fig. 3: TSN Scheduling example.

The rest of the paper is organized as follows. First, the preliminaries are discussed in Sec. II. Then, we introduce the overview of DeepScheduler in Sec. III and elaborate its design in Sec. IV. We present the implementation in Sec. V, and evaluate proposed method in Sec. VI. We discuss the related work in Sec. VII and conclude the paper in Sec. VIII.

II. Preliminary

A. Switch Model of TSN

Fig. 2 illustrates the key differences between the traditional Ethernet switch and the TSN switch. Firstly, all TSN switches and compatible devices in the same network share a globally synchronized clock with nanosecond precision, as defined in IEEE 802.1AS standard [12]. Secondly, TSN switches can additionally reserve specific time slots for potentially critical traffic, such as sensor signals or industrial control data. This is accomplished by a Gate Control List (GCL) that specifies the exact time that each queue is allowed to transmit data packets [3]. As the GCL in Fig. 2 shows, only q_1 can be selected for transmission in time slot 1, and q_2 in time slot 2 accordingly.

Beyond that, researchers have recently tried to enhance TSN's switching capability from various perspectives. For example, Zhao *et al.* [13] and Yin *et al.* [14] empowers TSN to deal with event-trigger critical traffic, Tan *et al.* [15] combines TSN with IP layer deterministic technologies. These efforts are orthogonal to and can be integrated in DeepScheduler.

B. Scheduling Problem and Formulation

Fig. 3 illustrates an example of TSN scheduling. There are two sensors, two switches, a robot arm, and some background

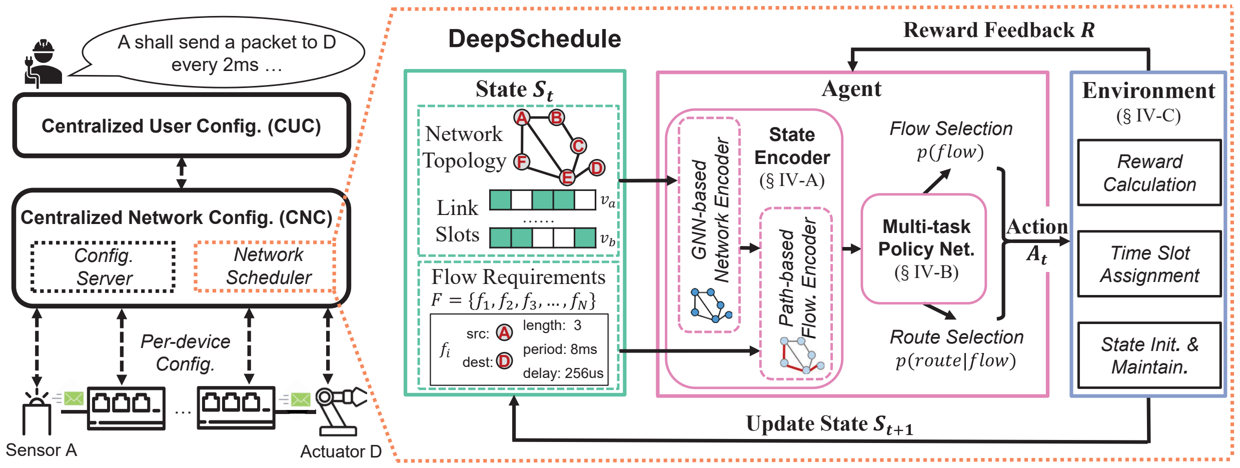


Fig. 4: Overview of DeepScheduler.

traffic in the simple TSN network. The sensors periodically send critical data packets to control the movement of the robot arm. As the physical link can only transmit a packet at one time, it is necessary to predetermine the sending and receiving time of the flows on each switch to ensure determinism.

We model the network topology as a directed graph $\mathcal{G}(V, L)$, where the switches are graph vertices (V) and each physical connection between nodes $v_a, v_b \in V$ adds two links $l_i = (v_a, v_b), l_j = (v_b, v_a) \in L$. In addition, all the critical flows' requirements form a set $F = \{f_1, f_2, ..., f_N\}$. And f_i contains the i-th flow's source ($f_i.src$), destination ($f_i.dest$), packet length ($f_i.len$), period ($f_i.prd$), and maximum transmission delay (f_i.MD). Following previous work's convention [16], we divide the continuous time interval into discrete time slots of $1/64\,\text{ms}$ (enough for a switch to process and transmit one maximum transmission unit (MTU) packet at 1000Mbps), and set the global scheduling period as 1024 time slots ($16\,\text{ms}$).

TSN scheduling is a time slot assignment process with the following constraints: (1) Timing constraints. Each flow should send its packets following f_i.prd, and the end-to-end delay should be smaller than f_i.MD. (2) Slot constraint. Each slot can only be assigned to at most one data frame. (3) Adjacent link constraint. A frame can only be scheduled on a subsequent link after the reception on the previous link. (4) Flow isolation constraint. The packets for different data flows shall not be interleaved over the same link. The above constraints form an NP-hard combinatorial optimization problem and can be modeled as ILP [16] or SMT [17] problems.

III. Overview of DeepScheduler

A. TSN Configuration Model

As shown in the left part of Fig. 4, IEEE 802.1Qcc [18] defines four primary components of a TSN network, *i.e.*, end devices, switches, centralized user configuration (CUC), and centralized network configuration (CNC). The CUC discovers the end devices and retrieves the user's flow requirements. The CNC receives flow requirements from the CUC and collects the network's physical topology. Then, the network scheduler like DeepScheduler in CNC calculates a global schedule that satisfies all TSN network requirements based on these inputs. Following the global schedule, CNC's configuration server configures each switch or end device's TSN features, such as gate control lists, packet sending time, *etc.*

B. DeepScheduler Workflow

As shown in the right part of Fig. 4, DeepScheduler adopts a general model-free reinforcement learning workflow. It solves the TSN scheduling as a multi-step decision-making problem, where a neural network-based agent takes the current *state* of the problem as the input and outputs a scheduling *action*. An environment maintains the *state* based on the agent's output and feeds back *reward* to the agent when necessary.

Specifically, during each step t, DeepScheduler observes the *state* S_t, containing physical network topology, links' slot assignment status, and the set of all flow requirements F. Afterwards, it applies a scalable two-stage state information encoder (Sec. IV-A) to process this observation, followed by a multi-task policy neural network (Sec. IV-B) to make decision of the *action* A_t, *i.e.*, selecting next flow to schedule as well as its routing. The environment executes A_t by allocating time slots for the selected flow, and updates the link status in *state* S_{t+1} accordingly. After the termination of entire process, the environment calculates a *reward* signal R for the RL algorithm (Sec. IV-C) to gradually improve the DeepScheduler agent.

IV. Design of DeepScheduler

A. State Information Encoder

We adopt a two-stage encoding pipeline to process the network status and critical traffic flows progressively. Firstly, a graph neural network (GNN) [19]–[21] is incorporated to handle arbitrary network sizes and topologies. Afterwards, we fuse the data flows' requirements and the network status by aggregating link-level features along their possible routes.

1) GNN-based Network Encoder: The GNN-based network encoder is proposed to scale DeepScheduler with arbitrary network topologies. It takes the physical topology and link

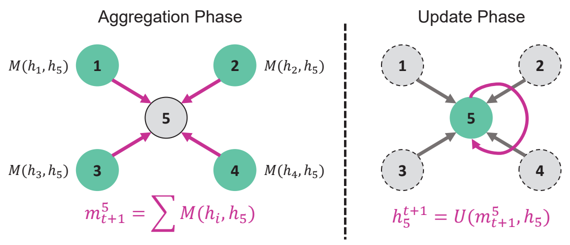


Fig. 5: Message passing framework overview.

status as input and outputs an embedding e_l for each link l, and an embedding s for the TSN network globally.

Network State Representation. Nowadays, most GNNs are designed for node features and performing node tasks such as node classification or prediction [22], while the TSN scheduling problems care more about the link status. Consequently, we transform the original topology $\mathcal{G}(V,L)$ into a virtual graph $\mathcal{L}(V',L')$ to model the physical links. In particular, each node in $\mathcal{L}$ represents a directed link in $\mathcal{G}$, and an edge in $\mathcal{L}$ is generated if two links l_a, l_b in $\mathcal{G}$ are end-to-end connected:

$$V' = L; L' = \{(l_a,l_b) : l_a, l_b \in G, l_a \cap l_b \neq \emptyset\}. \quad (1)$$

As aforementioned, each link l in the TSN network has 1024 time slots as a global scheduling period. Thus, we present raw feature $x_l \in \{0,1\}^{1024}$ as the input of link l, where $x_l[i] = 1$ means the i-th slot in link l is occupied by TSN flows.

Message Passing Neural Network. Message-passing neural network (MPNN) [21] is a general framework that formulates the majority of existing GNN models. As Fig. 5 illustrates, the MPNN contains an aggregation phase and an update phase in each step. During the aggregation, every node calculates *messages* for its neighbors, combining their current hidden states by a function $M(\cdot)$. Then, *messages* for the same node are aggregated through an element-wise sum operation. During the update, the hidden state of each node is updated with the aggregated *messages* by another function $U(\cdot)$. After repeating the above two phases for T times, information of T-hop neighbors is propagated into each node's hidden state.

In DeepScheduler, the GNN-based network encoder adopts two separate multi-layer perceptron (MLP) modules as message function $M(\cdot)$ and update function $U(\cdot)$. Formally, the GNN modules can be denoted as:

$$\begin{aligned} m_{t+1} &= \sum_{w \in \mathcal{N}(l)} M(h^t_w || h^t_l) \\ h^{t+1}_l &= U(m_{t+1} || h^t_l) \end{aligned}, t = 0,1,...,T-1, \quad (2)$$

where $||$ denotes the concatenation operator, and the neighborhood $\mathcal{N}(l)$ of each link l is defined as links that have a directed edge in L' pointing to l. We use x_l as input h^0_l, and final hidden state h^T_l as link embedding e_l.

Graph-level Summarization. Besides link-level encoding, DeepScheduler summarizes the entire network status from a global perspective. We add a super node S into the virtual graph $\mathcal{L}$ and connect every existing node to it. Its initial hidden state h^0_S is the average of all raw input vectors, and the last hidden state h^T_S is used as the graph summary embedding s. Fig. 6 demonstrates the architecture of GNN-based network

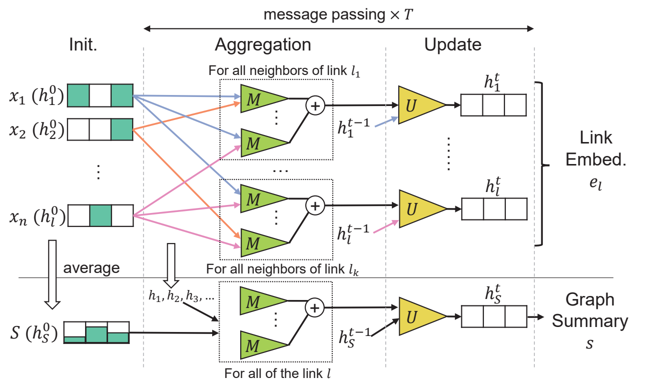


Fig. 6: Architecture of GNN-based network encoder.

encoder. Along with the message passing for ordinary links, node S receives messages from all the other nodes during each step and updates its hidden state following Eq.(2).

2) Path-based Flow-Aware Encoder: The path-based flow-aware encoder aims for an informative representation of each flow that captures the dependencies between the flow and network status. It encodes every flow $f_i \in F$ into an embedding z_i, by combining the GNN's output and the raw requirements.

Path-based Feature Aggregation. To tackle the complex dependencies, we obtain the perspective of each flow's possible routing. Specifically, we extract the link embeddings along k of the shortest paths for each flow and aggregate them. As the order of the links through which the packets may pass also contributes to the scheduling decision, a simple summation of link-level features will deteriorate the model's expressiveness. Consequently, we adopt the Gated Recurrent Unit (GRU) [23] to model the paths in an order-preserving manner.

Fig. 7 demonstrates an example of a proposed path-based feature aggregation. There are 3 simple shortest paths from source node A to destination node D. For each path, we sequentially feed its link embeddings into the GRU module and take the final hidden state as the embedding p^j_i for the j-th path of flow f_i. The hidden state carrying the previous steps' information is a part of the GRU's input in the next step. In this way, we obtain 3 path embedding vectors with the same hidden state dimensions, thereby limiting the dimensionality of the subsequent scheduling decision problem.

Flow Embedding Integration. Apart from the source, destination nodes, we treat flow f_i's remaining entities $(f_i.prd, f_i.\text{MD}, f_i.len)$ as a vector $y_i \in \mathbb{R}^3$, independent of the GNN outputs. To avoid numeric overflow, the raw vector y_i is standardized in advance and then linearly projected into a requirement embedding r_i.

For any flow $f_i \in F$, we integrate the aforementioned graph summary embedding s, path embeddings $p^1_i, p^2_i, ...,$ and flow requirement embedding r_i together as follows:

$$\begin{aligned} h_i &= \text{MLP}(s||p^1_i||p^2_i||...||p^k_i||r_i); \\ z_i &= h_i + \text{MHA}(h_i). \end{aligned} \quad (3)$$

The MHA here denotes a Multi-Head self-Attention module with 4 heads. It is similar to the Transformer's encoder layer [24] without positional encoding. Briefly speaking, the

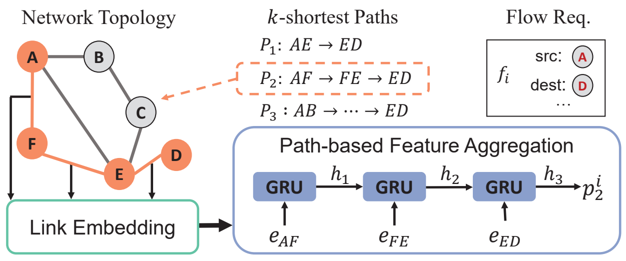


Fig. 7: Example of path-based feature aggregation.

MHA module calculates each flow's embedding by a weighted combination of all the other flows, based on the relevance score calculated between each pair of embeddings h_i, h_j.

To this end, we come up with an informative representation z_i for flow f_i that combines not only its semantic characteristics but also information from the entire network (with the summary s) and related flows (with the attention mechanism).

B. Multi-task Policy Network

The multi-task policy network is designed to capture the essential characteristics for a global scheduling decision with simplified multi-step sub-tasks. It obtains all the flow representations $z_1, z_2, ..., z_N$ from the state information encoder, and outputs stochastic decisions for each sub-task at each step.

1) Action Space Decomposition: In RL, both large action space and long action episodes have negative effects on sample efficiency and model expressiveness. To deal with exponentially large combinatorial action spaces, we decompose the entire scheduling policy π into a sequencing policy π^s that selects the next flow to schedule $f_t \in F/F_{\text{scheduled}}$, and a routing policy π^r that picks a route $r_t \in \{1, 2, ..., k\}$ for the selected flow. As DNN is more suitable for high-level decision-making rather than yielding the exact solutions [25], we leave the final time slot assignment process out of the agent's design.

Such a decomposition makes the whole learning framework a sequential Markov game with two interactive agents [26], where both agents share the same high-level objective. Consequently, the multi-task policy π with the initial state s_0 can be factorized as:

$$\pi(f_1, r_1, f_2, r_2, ..., f_N, r_N | s_0) = \prod_{t=1}^{N} \pi^s(f_t | s_{t-1}) \pi^r(r_t | f_t, s_{t-1}), \tag{4}$$

where π^s and π^r interleavingly generate the full solution $\mathcal{C} = \{f_1, r_1, f_2, r_2, ..., f_N, r_N\}$ for the scheduling problem.

2) Policy Network Design: Given the decomposed action space and carefully designed state encoder, the DNN design of the policy network is rather straightforward. For each flow embedding input z_i, we apply two MLP modules $Q(\cdot) : \mathbb{R}^{d_h} \to \mathbb{R}$ and $W(\cdot) : \mathbb{R}^{d_h} \to \mathbb{R}^k$, where k is the number of shortest paths considered. We treat $Q(z_i)$ as the priority score of the TSN flows to schedule and adopt a softmax [27] to output the sequencing policy:

$$\pi^s(f_i | s_t) = \frac{\exp(Q(z_i))}{\sum_j \exp(Q(z_j))}. \tag{5}$$

Notice that the scheduled flows $f_j \in F_{\text{scheduled}}$ shall not be sampled again, thus we set $Q(f_j) = -\infty$ manually. Similarly, the routing policy adopts another softmax operation to select across k possible routes.

C. Environment & RL Training

In DeepScheduler, the environment and RL training algorithms cooperatively provide the agent with trial-and-error opportunities and guide it to learn better scheduling policies over time from the problem's implicit data distributions.

1) Environment: The environment is mainly responsible for state maintenance and reward calculation. Given the next flow f_{t+1} to schedule and its route r_{t+1}, it assigns the currently available time slots and updates the link status accordingly. After all the flows are scheduled properly or the scheduling fails, it calculates the reward signal R for the RL training.

Time Slot Assignment. The time slot assignment algorithm will be called at every step of the RL, *i.e.*, N times in a single training iteration of DeepScheduler. In practice, we select the simple but widely adopted *earliest-valid-slot-first* algorithm [4], [5]. For each link in the predetermined route, it iterates all the empty slots after the flow's departure time from the previous link, and selects the first slot that satisfies the constraints mentioned in Sec. II-B. If there is no valid slot at the current link, it will terminate the scheduling process.

Reward Calculation. DeepScheduler aims to satisfy all the flow requirements globally, and the reward design is closely related to this overall objective. In particular, it is defined as:

$$R = \mathbf{1}_{\text{success}} + \alpha \times P_{\text{success}}, \tag{6}$$

where $\mathbf{1}_{\text{success}}$ equals 1 if the scheduling succeed and 0 otherwise, and P_{success} is the percentage of successfully scheduled flows before termination. The higher reward R implies a better performance of the scheduling algorithm. Conceptually, the first term $\mathbf{1}_{\text{success}}$ represents the scheduling problem's ultimate objective. The second term P_{success} smooths the reward and helps DeepScheduler improve its schedulability even if it fails in some cases. We set $\alpha = 0.1$ after fine-tuning.

2) RL Training: DeepScheduler adopts an enhanced policy gradient algorithm for RL training. Its main idea is to directly perform gradient descent on the DNN parameters with the observed rewards. For simplicity, we define all the parameters in DeepScheduler's neural network as θ, and the parameters related to sequencing policy and routing policy as θ^s and θ^r, respectively. The gradient is formulated according to the well-known REINFORCE algorithm [28] as follows:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} = \mathbb{E}_{\pi}\Big[\Big(R(\mathcal{C}_\pi) - b\Big) \cdot \sum_{t=1}^{N} \Big(\nabla_{\theta^s} \log \pi^s(f_t | s_{t-1}; \theta^s) + \nabla_{\theta^r} \log \pi^r(r_t | f_t, s_{t-1}; \theta^r)\Big)\Big], \tag{7}$$

where $\mathcal{C}_\pi$ denotes the full solution trajectory sampled according to policy π and the baseline value b is introduced to reduce gradient variance and increase the training speed. It is set to the exponential running average of previous rewards R, representing the improvement of policy π over time. Notice that θ^s and θ^r share the same parameters from the encoder.

Algorithm 1: RL w/ Enhanced Policy Gradient.

```
Input: dataset $D$, training steps $K$, batch size $B$
Output: optimized parameter $\theta$ of DeepScheduler
$D' \leftarrow$ empty FIFO queue;
for $step = 1$ to $K$ do
    $s_0^i \sim \textsc{SampleInput}(D + D')$ for $i \in \{1, ..., B\}$;
    $\mathcal{C}^i \leftarrow \{\}$ for $i \in \{1, ..., B\}$;
    for $t = 1$ to $N$ do
        for $i = 1$ to $B$ do
            $f_t^i \sim \textsc{SampleFlow}(\pi^s(\cdot | s_{t-1}^i))$;
            $r_t^i \sim \textsc{SampleRoute}(\pi^r(\cdot | f_t^i, s_{t-1}^i))$;
            $s_t^i \leftarrow \textsc{UpdateState}(s_{t-1}^i, f_t^i, r_t^i)$;
            $\mathcal{C}^i \leftarrow \mathcal{C}^i$.append$(f_t^i, r_t^i, s_t^i)$;
    $\boldsymbol{\nabla_\theta} \leftarrow \frac{1}{B}\sum_{i=1}^{B}(R(\mathcal{C}^i) - b)\nabla_\theta \log \pi_\theta(\mathcal{C}^i)$;
    $\theta \leftarrow \theta + \alpha \boldsymbol{\nabla_\theta}$;
    $b \leftarrow \beta \cdot \text{average}_i R(\mathcal{C}^i) + (1 - \beta) \cdot b$;
    $D'$.insert$(s_0^{i^*})$, where $i^* = \arg\min_i R(\mathcal{C}^i)$;
if $step \geq threshold$ then
    Update $D$ with flow number $N' > N$;
return $\theta$;
```

Alg. 1 demonstrates the RL training process enhanced with the proposed hard sample mining and incremental learning techniques. Inspired by the prioritized replay for Q-learning [29], we propose a hard sample mining method to help the model learns faster. Specifically, we maintain a fixed-size first-in-first-out buffer D', collected by inserting the already trained sample with the lowest advantage $(R(\mathcal{C}_\pi) - b)$ in each training batch. During training, the data points are uniformly sampled from $D + D'$, thus the more informative samples in D' may be re-utilized in the subsequent training. Moreover, it is difficult for the model to improve its policy if the initial RL episode length (*i.e.*, the number of flows) is too long. Thus, we propose an incremental training technique that gradually adds the number of flows during training. Without it, the training process may sometimes fail to converge and have to be restarted with a different random seed.

V. Implementation

We implement DeepScheduler using PyTorch [30] and PyTorch Geometric [31]. The GNN encoder consists of 3 message passing steps, where the functions $M(\cdot)$ and $U(\cdot)$ are both two-layer MLPs. The dimensions of link/path/flow embedding (e_v, p_i^k, f_i) and requirement embedding (r_i) are 128 and 32, respectively. Leaky_ReLU [32] is used as activation function for all the modules. We train the model for 10 epochs with the Adam optimizer [33], taking about . The learning rate starts at 10^{-4} and decays by 0.99 every epoch. Training data is generated on the fly, with 28 data points per batch and 4,000 steps per epoch. The size of the hard sample buffer is set to 100. We increase the number of flows in training data by 10 every epoch from 150 to 200. During the evaluation, DeepScheduler samples 10 candidate solutions from the stochastic output of the multi-task policy network and selects a valid one. In terms of the hardware environment,

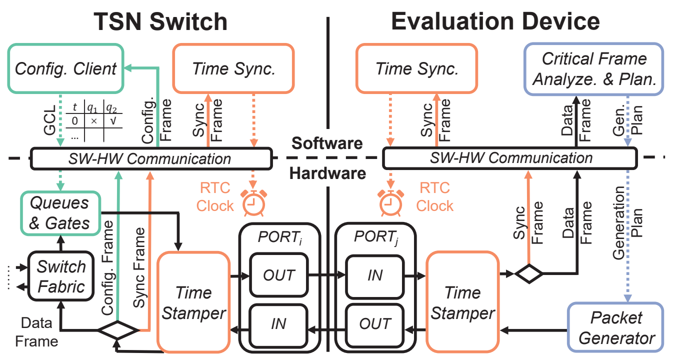


Fig. 8: Architecture of TSN testbeds.

experiments for scheduling methods are all conducted on a server with two Intel Xeon E5-2560 CPUs, two NVIDIA 2080Ti GPUs, and 256 GB memory.

To verify the schedule calculated by DeepScheduler, we further develop a suite of TSN switches and evaluation devices on the Xilinx-Zynq7000 FPGA board [34] with hardware-software co-design. As Fig. 8 shows, its hardware part implements time-critical functionalities like switch fabric and transmission selection, while the software part contains modules with sophisticated processing logic, including time synchronization, CNC configuration client, *etc.* Both the TSN switch and evaluation device comply with the IEEE TSN standards [3], [12], [18], and can measure end-to-end packet latency with nanosecond-level precision. We integrate DeepScheduler into the testbed as a part of CNC.

VI. Evaluation

A. Experiment Setup

1) Network Topology: We select the following three typical random topologies to represent various application scenarios:

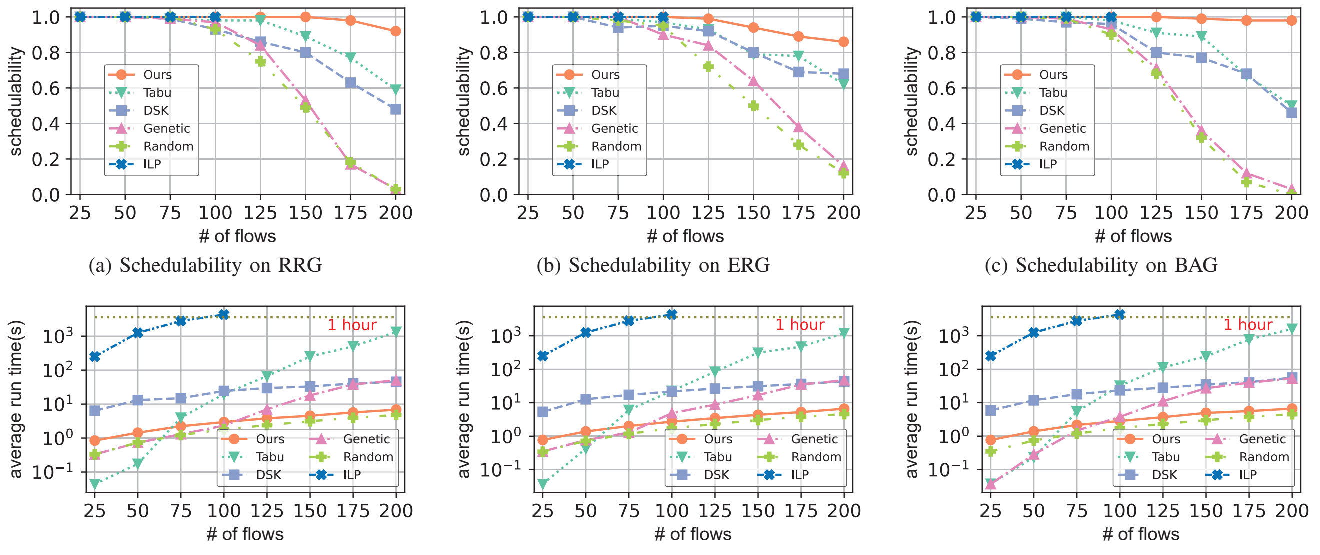
Random Regular Graph (RRG) has each node randomly connected to four other nodes. It resembles the case where each switch has a fixed number of ports.

Erdős-Rényi Graph (ERG) is generated by connecting each pair of nodes with a probability of 0.25, which shares universal properties across all graph types [35].

Barabási-Albert Graph (BAG) is grown by attaching each new node with three edges preferentially to existing nodes with a high degree. It represents the situation where several central switches dominate the network evolution [36].

We only pick the connected graphs generated during experiments. Each node represents a TSN switch in an industrial network. Unless otherwise mentioned, each topology has 20 nodes, and we train DeepScheduler against them separately.

2) Flow Requirement: The TSN flow requirements are randomly generated in accordance with IEC/IEEE 60802 [37], the TSN profile for industrial automation. We randomly select two switches for each flow as the source and destination. The period and maximum end-to-end delay are randomly chosen from $\{0.5\,\text{ms}, 1\,\text{ms}, 2\,\text{ms}, 4\,\text{ms}, 8\,\text{ms}, 16\,\text{ms}\}$ and $\{2\,\text{ms}, 4\,\text{ms}, 8\,\text{ms}, 16\,\text{ms}\}$, respectively. The payload length is uniformly distributed between 1 and 8 packets.



(a) Schedulability on RRG (b) Schedulability on ERG (c) Schedulability on BAG

(d) Average run time on RRG (e) Average run time on ERG (f) Average run time on BAG

Fig. 9: Overall performance of DeepScheduler on different network topologies.

3) Comparative Methods: We compare DeepScheduler's performance to that of the following methods:

Random samples 10 candidate orders and routes randomly and then assigns time slots by the default greedy algorithm.

ILP [16] formulates the scheduling constraints into integer linear programming and solves them with Z3-solver [38].

Tabu [4] and **Genetic** [5] adapt general-purpose metaheuristic search algorithms, *i.e.*, tabu search and genetic search, into TSN scheduling. They iteratively generate many flow orders and search for a better one until the solution cannot be improved anymore.

DSK utilizes domain-specific expert knowledge in TSN scheduling. It first sorts all flows according to predefined multi-level priority [7], then assigns the time slots by a customized *lowest-degree-first* heuristic algorithm [39].

Tabu and DSK are state-of-the-art methods.

B. Overall Performance

Fig. 9 demonstrates the overall performance of DeepScheduler. For each experiment, we randomly generate 100 scheduling problems, then record the schedulability, *i.e.*, successful scheduling rate, and time consumption of different methods. To sum up, DeepScheduler achieves the highest schedulability among the counterparts and only takes several seconds to finish all test cases.

Fig. 9a, 9b, and 9c display the schedulability results. DeepScheduler consistently outperforms other methods, except for ILP, which has exhausted all possible solutions. When the problem size reaches 200 flows, DeepScheduler retains 92%, 88%, 98% schedulability and is 35%, 20%, 48% higher than the best counterpart on RRG, ERG, and BAG, respectively. Moreover, DSK works well on the ERG but fails to outperform Tabu on RRG and BAG, indicating the expert knowledge has poor adaptiveness to the problem settings. By contrast, DeepScheduler can automatically learn the implicit distributions from the perspective of traffic flows.

Fig. 9d, 9e, and 9f illustrate the run time results in logarithmic scale. The average time consumption of DeepScheduler, DSK, and Random increase linearly to the problem size, while others grow exponentially. Since ILP takes up more than 1 hour with 100 flows, it is not reported for greater problem sizes. Furthermore, DeepScheduler takes about 50% more run time than Random, which results from the deep learning agent. Even with 200 flows, DeepScheduler is able to finish scheduling within 7 s, while Tabu and DSK take at least 20 min and 45 s, respectively. This is because DSK's degree calculation is time-consuming, and Tabu goes through thousands of candidates to look for a valid schedule. In contrast, DeepScheduler utilizes relatively light-weight algorithms and can find feasible solutions with a limited number of trials, thus having the lowest run time apart from Random.

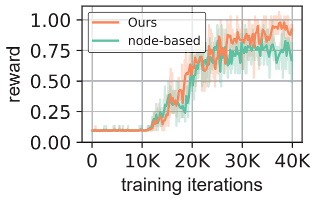
C. Ablation Study

1) Impact of Path-based Flow-aware Encoder: We compare the path-based flow-aware encoder with the node-based encoder, which directly calculates the node embedding for each flow's source s and destination d as follows:

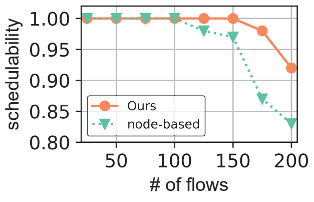
$$x_s = \sum_{v.src=s} e_v, x_d = \sum_{u.dest=d} e_u. \tag{8}$$

The node embedding x_s, x_d is then used to replace the path embedding p_i^j in Eq.(3). We train the model on RRG with the same hyperparameters as Sec. V and then report their training curve and schedulability. The incremental training technique is disabled to better demonstrate their expressiveness.

As is shown in Fig. 10, the node-based method tends to have a higher variance, and its training reward reaches a plateau before 0.8. Meanwhile, the proposed path-based method can learn a better policy with the final training reward of over



(a) Training curve



(b) Schedulability

Fig. 10: Path-based vs. node-based flow encoder.

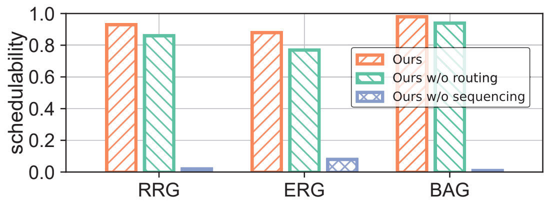
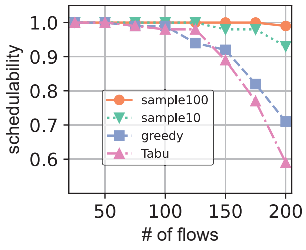
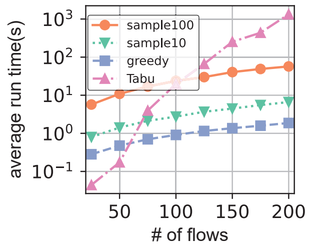


Fig. 11: Breakdown of multi-task policy's contribution.

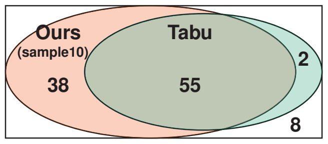


(a) Schedulability

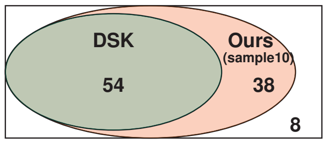


(b) Average run time

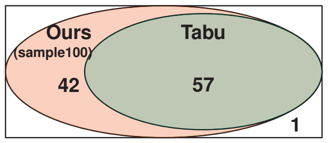
Fig. 12: Performance of different decoding strategies and Tabu.



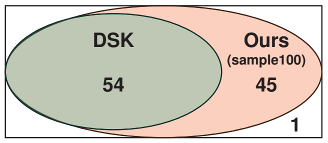
(a) Sample10 vs. Tabu



(b) Sample10 vs. DSK



(c) Sample100 vs. Tabu



(d) Sample100 vs. DSK

Fig. 13: Schedulability on the same set of problems.

0.9 on average. Fig. 10b further demonstrates that the path-based encoding has 9% schedulability improvement compared to its counterpart on the 200-flow problems. This is because the path-level feature better captures the scheduling problem's semantic characteristics from perspective of each flow.

2) Impact of Multi-task Policy: To validate the effectiveness of each part of the proposed multi-task policy, we replace the sequencing part and the routing part with random selection individually. We train the modified models from scratch and evaluate their schedulability on each topology with 200 flows.

As Fig. 11 illustrates, removing any part of DeepScheduler's policy network worsens the schedulability. The sequencing policy contributes to 91% more average schedulability than the Random baseline, while the routing policy further improves it by 7.3%. The sequencing policy has a greater impact than the routing policy in line with previous work's observation [4], [17], [40]. Without it, DeepScheduler has only marginal schedulability improvement compared to the Random baseline.

3) Impact of Sampling Strategy: We compare the sampling strategy with 10 samples, 100 samples, and greedy decoding that selects the output with the largest probability on RRG. Tabu search, the best available heuristic, is also illustrated.

Fig. 12 shows the impact of sampling strategy on the performance of DeepScheduler. As expected, the schedulability becomes higher when the number of samples increases. At 200-flow problem size, the greedy, sample10, and sample100 methods' schedulability are 71%, 92%, and 99%, respectively. When sampling 100 times, DeepScheduler successfully schedules all TSN flow requirements for problem size smaller than 200, while taking an order of magnitude less time than Tabu This suggests that DeepScheduler learns a more efficient policy on TSN scheduling problems.

D. Deep Dive of DeepScheduler

1) Schedulability on TSN Problems: To better understand the schedulability of DeepScheduler, we conduct a more detailed analysis of the scheduling results. For the same set of 100 randomly generated scheduling problems, we record the scheduling results on DeepScheduler, Tabu, and DSK.

Fig. 13 shows the coverage relationship between their schedulable sets. Most problems that cannot be scheduled by DeepScheduler, are also unsolvable for comparative methods. Specifically, DeepScheduler with 10 samples covers 96.5% and 100% of Tabu and DSK's schedulable problems. Fig. 13c, 13d further demonstrate that DeepScheduler with 100 samples can schedule all of the exceptional cases. This implies that DeepScheduler effectively covers the scheduling capability of SOTA methods. Therefore, industrial network operators can safely replace traditional heuristic-based or expert knowledge-based methods with it in most situations.

2) Generalizability on Different Network Size: The aforementioned experiments are all conducted with 20 nodes in each network topology. To evaluate DeepScheduler's model generalizability, we use the same pre-trained model to schedule RRG networks of different sizes. To maintain a similar network load, we generate 100, 150, 200, and 250 flows for the network with 10, 15, 20, and 25 nodes, respectively.

As Fig. 14 demonstrates, DeepScheduler is capable of solving the TSN scheduling problem on smaller unseen network sizes. It successfully schedules 96%, 99%, and 92% of cases on networks with 10, 15, and 20 nodes, respectively. However, there is a performance decline on it for scheduling problems with 25 nodes and 250 flows, indicating that DeepScheduler is not well generalizable to greater network sizes and unseen flow numbers. This is because increasing the number of flows leads to larger search space and longer action sequences, which are not learned during the training process. Nevertheless,

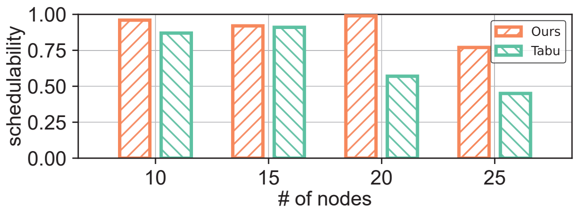


Fig. 14: Generalizability on different network sizes.

DeepScheduler still outperforms the Tabu baseline by 32% and can be improved through fine-tuning.

E. *Testbed Deployment*

To verify the schedule calculated by DeepScheduler, we deploy it into CNC in our TSN testbeds with 9 TSN switches and 8 evaluation devices. The testbeds make up an A380 topology as shown in Fig. 15a, which is a simplified version of the control network on the Airbus A380 airplane [41]. In each experiment, we randomly generate and schedule 20 flows, then record their transmission delay at nanosecond-level precision. The bias (difference between average delay and scheduled delay) and jitter (standard deviation of delay) for every flow in the experiments are measured for 1000 periods.

Fig. 15b and 15c illustrate the cumulative distribution of testbed results for 100 randomly generated experiments. As Fig. 15b demonstrates, the biases lie narrowly between 3.3 µs and 3.5 µs, with an average of 3.38 µs. This is caused by the Ethernet PHY's processing delay (already considered in the time slot length design). Fig. 15c shows that 95% of jitters are less than 100 ns, which is two orders of magnitude lower than a single time slot. Moreover, all testbed experiments show zero packet loss rate, indicating there is no collision between flows. In conclusion, all packets are transmitted deterministically and in consistency with the scheduling results.

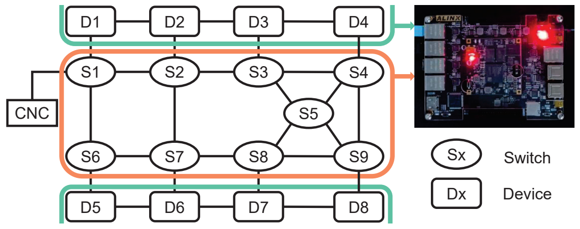
VII. Related Work

A. *TSN Scheduling*

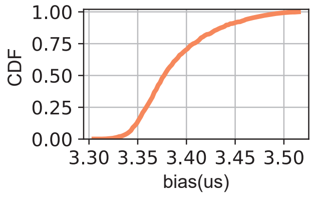
The TSN scheduling problem has been widely studied. Most researchers try to satisfy a complete set of flow requirements globally. Some of them formulate the scheduling constraints into Satisfiability Modulo Theories (SMT) [17], [42] or Integer Linear Programs (ILP) [16]. Others utilize the heuristics search [4], [5], [43] or domain-specific knowledge [6], [7], [44] to speed up scheduling with sacrification on schedulability. Different from them, DeepScheduler automatically learns a global scheduling policy and shows superior performance. Another group of work considers the online scheduling problem. They propose rule-based [45], [46] or learning-based [39], [47] algorithms to schedule incoming flows based on current network status. Albeit inspiring, they are infeasible for the industry due to the restricted problem assumptions.

B. *DRL for Combinatorial Optimization*

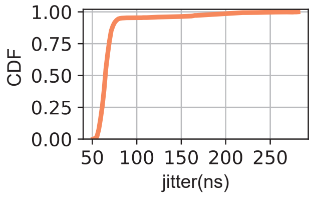
The attempts of adapting DRL into NP-hard combinatorial optimization starts in 2017, when Bello *et al.* [48] combines the policy gradient and pointer network architecture [8] to tackle the famous traveling salesman problem (TSP). Based on their work, Kool *et al.* [9] improves the pointer network with an attention mechanism. Some work tries other DRL frameworks like deep Q-learning and Monte Carlo tree search, and applies them to maximum cut problems [49] and bin packing problems [50], respectively. Compared to the above work, DeepScheduler's unique contributions are the two-stage state information encoder and multi-task policy to tackle the distinctive challenges of the TSN scheduling problem.



(a) Testbed of A380 topology.



(b) CDF of bias.



(c) CDF of jitter.

Fig. 15: Testbed deployment results.

C. *GNN in Computer Networking*

Recently, several pioneering works utilize graph neural networks for computing networking management or optimization. In particular, RouteNet and its predecessor [51], [52] try to model the relationship between network links and routing policies with GNN and then estimate some key performance indicators such as delay and jitter. Zhu *et al.* [53] incorporates GNN-based models to prune search space in the large-scale network planning problem. These studies mainly aim for a scalable representation of the overall network status, whereas DeepScheduler further proposes an information fusion framework between TSN traffic flow and current network status through path-based feature aggregation.

VIII. Conclusion

In this work, we propose DeepScheduler, a novel DRL-based TSN scheduler that automatically learns the implicit data distributions. It effectively processes any network topologies with the two-stage flow-aware information encoder, and efficiently explores the entire solution space with the action space decomposition and enhanced RL training. The evaluations demonstrate that DeepScheduler is fast and scalable, and improves the schedulability significantly. We believe DeepScheduler makes a meaningful step towards flexible manufacturing.

Acknowledgment

This work is supported in part by the National Key Research Plan under grant No. 2021YFB2900100, the NSFC under grant No. 61832010, 62202263, 62202262, and 61972131.

References

[1] H. Lasi, P. Fettke, H.-G. Kemper *et al.*, “Industry 4.0,” *Business & information systems engineering*, vol. 6, no. 4, pp. 239–242, 2014.

[2] D. Bruckner, M.-P. Stănică, R. Blair *et al.*, “An introduction to opc ua tsn for industrial communication systems,” *Proceedings of the IEEE*, vol. 107, no. 6, pp. 1121–1131, 2019.

[3] *Enhancements for Scheduled Traffic*, IEEE Std. 802.1Qbv, 2015.

[4] F. Dürr and N. G. Nayak, “No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN),” in *Proc. of the RTNS*, ser. RTNS ’16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 203–212.

[5] M. Pahlevan and R. Obermaisser, “Genetic Algorithm for Scheduling Time-Triggered Traffic in Time-Sensitive Networks,” in *Proc. of IEEE ETFA*, vol. 1, Sep. 2018, pp. 337–344.

[6] D. Hellmanns, A. Glavackij, J. Falk *et al.*, “Scaling TSN Scheduling for Factory Automation Networks,” in *Proc. of IEEE WFCS*, Apr. 2020, pp. 1–8.

[7] J. Yan, W. Quan, X. Jiang *et al.*, “Injection Time Planning: Making CQF Practical in Time-Sensitive Networking,” in *Proc. of IEEE INFOCOM*, Jul. 2020, pp. 616–625.

[8] O. Vinyals, M. Fortunato, and N. Jaitly, “Pointer Networks,” in *Proc. of the NeurIPS*, vol. 28. Curran Associates, Inc., 2015.

[9] W. Kool, H. van Hoof, and M. Welling, “Attention, learn to solve routing problems!” in *Proc. of the ICLR*, 2019.

[10] H. Hu, X. Zhang, X. Yan *et al.*, “Solving a New 3D Bin Packing Problem with Deep Reinforcement Learning Method,” *arXiv:1708.05930 [cs]*, Aug. 2017.

[11] J. Zhang, B. Zi, and X. Ge, “Attend2Pack: Bin Packing through Deep Reinforcement Learning with Attention,” *arXiv:2107.04333 [cs]*, Aug. 2021.

[12] *Timing and Synchronization for Time-Sensitive Applications*, IEEE Std. 802.1AS, 2020.

[13] Y. Zhao, Z. Yang, X. He *et al.*, “E-tsn: Enabling event-triggered critical traffic in time-sensitive networking for industrial applications,” in *Proc. of IEEE ICDCS*, Jul. 2022.

[14] S. Yin, S. Wang, Y. Huang *et al.*, “Critical Event-Triggered Flows Tolerance in Time-Sensitive Networks,” in *Proc. of IEEE ICC Workshops*, Jun. 2021, pp. 1–6.

[15] W. Tan and B. Wu, “Long-distance Deterministic Transmission among TSN Networks: Converging CQF and DIP,” in *Proc. of IEEE ICNP*, 2021, pp. 1–6.

[16] E. Schweissguth, P. Danielis, D. Timmermann *et al.*, “Ilp-based joint routing and scheduling for time-triggered networks,” in *Proc. of the RTNS*, 2017, pp. 8–17.

[17] S. S. Craciunas, R. S. Oliver, M. Chmelík *et al.*, “Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks,” in *Proc. of the RTNS*, ser. RTNS ’16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 183–192.

[18] *Stream Reservation Protocols (SRP) Enhancements and Performance Improvements*, IEEE Std. 802.1Qcc, 2018.

[19] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre *et al.*, “Convolutional networks on graphs for learning molecular fingerprints,” in *Proc. of the NeurIPS*, vol. 28, 2015.

[20] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *Proc. of the ICLR*, 2017.

[21] J. Gilmer, S. S. Schoenholz, P. F. Riley *et al.*, “Neural message passing for quantum chemistry,” in *Proc. of the ICML*, ser. ICML’17. JMLR.org, 2017, p. 1263–1272.

[22] J. Zhou, G. Cui, S. Hu *et al.*, “Graph neural networks: A review of methods and applications,” *AI Open*, vol. 1, pp. 57–81, 2020.

[23] K. Cho, B. Van Merriënboer, D. Bahdanau *et al.*, “On the properties of neural machine translation: Encoder-decoder approaches,” *arXiv preprint arXiv:1409.1259*, 2014.

[24] A. Vaswani, N. Shazeer, N. Parmar *et al.*, “Attention is All you Need,” in *Proc. of the NeurIPS*, I. Guyon, U. V. Luxburg, S. Bengio *et al.*, Eds. Curran Associates, Inc., 2017, pp. 5998–6008.

[25] Y. Bengio, A. Lodi, and A. Prouvost, “Machine Learning for Combinatorial Optimization: A Methodological Tour d’Horizon,” *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021.

[26] M. L. Littman, “Markov games as a framework for multi-agent reinforcement learning,” in *Proc. of the ICML*. Morgan Kaufmann, 1994, pp. 157–163.

[27] J. W. Gibbs, *Elementary Principles in Statistical Mechanics: Developed with Especial Reference to the Rational Foundation of Thermodynamics*, ser. Cambridge Library Collection - Mathematics. Cambridge University Press, 2010.

[28] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine Learning*, vol. 8, no. 3, pp. 229–256, May 1992.

[29] T. Schaul, J. Quan, I. Antonoglou *et al.*, “Prioritized experience replay,” in *ICLR (Poster)*, 2016.

[30] A. Paszke, S. Gross, F. Massa *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Proc. of the NeurIPS*. Curran Associates, Inc., 2019, pp. 8024–8035.

[31] M. Fey and J. E. Lenssen, “Fast graph representation learning with PyTorch Geometric,” in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[32] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[33] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[34] Xilinx. (2021, Jul.) Socs with hardware and software programmability. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[35] S. E. Fienberg, “A brief history of statistical models for network analysis and open challenges,” *Journal of Computational and Graphical Statistics*, vol. 21, no. 4, pp. 825–839, 2012.

[36] A.-L. Barabási and R. Albert, “Emergence of scaling in random networks,” *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[37] *TSN Profile for Industrial Automation*, IEC/IEEE Std. 60 802, 2018.

[38] Z3Prover. (2021) Github repository of z3prover. [Online]. Available: https://github.com/Z3Prover/z3

[39] C. Zhong, H. Jia, H. Wan *et al.*, “DRLS: A Deep Reinforcement Learning Based Scheduler for Time-Triggered Ethernet,” in *Proc. of the ICCCN*, Jul. 2021, pp. 1–11.

[40] F. Pozo, W. Steiner, G. Rodriguez-Navas *et al.*, “A decomposition approach for SMT-based schedule synthesis for time-triggered networks,” in *Proc. of IEEE ETFA*, Sep. 2015, pp. 1–8.

[41] F. Boulanger, D. Marcadet, M. Rayrole *et al.*, “A time synchronization protocol for a664-p7,” in *Proc. of IEEE DASC*. IEEE, 2018, pp. 1–9.

[42] F. Pozo, G. Rodriguez-Navas, H. Hansson *et al.*, “Smt-based synthesis of ttethernet schedules: A performance study,” in *Proc. of IEEE SIES*. IEEE, 2015, pp. 1–4.

[43] M. Pahlevan, N. Tabassam, and R. Obermaisser, “Heuristic list scheduler for time triggered traffic in time sensitive networks,” *ACM SIGBED Review*, vol. 16, no. 1, pp. 15–20, Feb. 2019.

[44] Y. Li, J. Jiang, and S. H. Hong, “Joint Traffic Routing and Scheduling Algorithm Eliminating the Nondeterministic Interruption for TSN Networks Used in IIoT,” *IEEE Internet of Things Journal*, pp. 1–1, 2022.

[45] N. Wang, Q. Yu, H. Wan *et al.*, “Adaptive Scheduling for Multicluster Time-Triggered Train Communication Networks,” *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1120–1130, 2019.

[46] Y. Huang, S. Wang, T. Huang *et al.*, “Online Routing and Scheduling for Time-Sensitive Networks,” in *Proc. of IEEE ICDCS*, Jul. 2021, pp. 272–281.

[47] H. Jia, Y. Jiang, C. Zhong *et al.*, “TTDeep: Time-Triggered Scheduling for Real-Time Ethernet via Deep Reinforcement Learning,” in *Proc. of IEEE GLOBECOM*, 2021, pp. 1–6.

[48] I. Bello, H. Pham, Q. V. Le *et al.*, “Neural Combinatorial Optimization with Reinforcement Learning,” *arXiv:1611.09940 [cs, stat]*, Jan. 2017.

[49] T. Barrett, W. Clements, J. Foerster *et al.*, “Exploratory Combinatorial Optimization with Reinforcement Learning,” in *Proc. of the AAAI*, vol. 34, no. 04, Apr. 2020, pp. 3243–3250.

[50] A. Laterre, Y. Fu, M. Jabri *et al.*, “Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization,” in *Proc. of the NeurIPS*, 2018.

[51] P. Almasan, J. Suárez-Varela, A. Badia-Sampera *et al.*, “Deep Reinforcement Learning meets Graph Neural Networks: Exploring a routing optimization use case,” *arXiv:1910.07421 [cs]*, Feb. 2020.

[52] K. Rusek, J. Suárez-Varela, P. Almasan *et al.*, “RouteNet: Leveraging Graph Neural Networks for network modeling and optimization in SDN,” *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2260–2270, Oct. 2020.

[53] H. Zhu, V. Gupta, S. S. Ahuja *et al.*, “Network Planning with Deep Reinforcement Learning,” in *Proc. of ACM SIGCOMM*, 2021, p. 14.