Efficient DRL-Based Congestion Control With Ultra-Low Overhead

Han Tian, Xudong Liao, Chaoliang Zeng, Decang Sun, Junxue Zhang, and Kai Chen, *Senior Member, IEEE*

***Abstract*— Previous congestion control (CC) algorithms based on deep reinforcement learning (DRL) directly adjust flow sending rate to respond to dynamic bandwidth change, resulting in high inference overhead. Such overhead may consume considerable CPU resources and hurt the datapath performance. In this paper, we present SPINE, a hierarchical congestion control algorithm that fully utilizes the performance gain from deep reinforcement learning but with ultra-low overhead. At its heart, SPINE decouples the congestion control task into two subtasks in different timescales and handles them with different components: 1) lightweight CC executor that performs fine-grained control responding to dynamic bandwidth changes; and 2) RL agent that works at a coarse-grained level that generates control sub-policies for the CC executor. Such two-level control architecture can provide fine-grained DRL-based control with a low model inference overhead. Real-world experiments and emulations show that SPINE achieves consistent high performance across various network conditions with an ultra-low control overhead reduced by at least 80% compared to its DRL-based counterparts, similar to classic CC schemes such as Cubic.**

***Index Terms*— Congestion control, deep reinforcement learning, transport layer protocols.**

I. INTRODUCTION

DRIVEN by the tremendous successes achieved by deep reinforcement learning (DRL) models in wide areas, e.g., games [1], [2], [3], computer systems, and networking [4], [5], [6], [7], the community is exerting efforts to gain the same success on the network transport by incorporating DRL into congestion control (CC) [8], [9], [10]. As one of the major advantages, DRL-based CC schemes have the capability to adapt to variant network conditions with one single control policy. Therefore, network engineers can be free from the operational challenge of manually tuning CC hyperparameters for unseen network conditions.

Manuscript received 20 April 2023; revised 18 July 2023 and 10 September 2023; accepted 24 October 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Caesar. This work was supported in part by the Hong Kong RGC TRS under Grant T41-603/20-R and Grant GRF-16213621, in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2021B0101400001, and in part by the NSFC under Grant 62062005. *(Han Tian and Xudong Liao contributed equally to this work.) (Corresponding author: Kai Chen.)*

Han Tian, Xudong Liao, Chaoliang Zeng, Decang Sun, and Kai Chen are with the iSing Laboratory, The Hong Kong University of Science and Technology, Hong Kong, SAR, China (e-mail: htianab@connect.ust.hk; xliaoaf@cse.ust.hk; czengaf@connect.ust.hk; kaichen@cse.ust.hk).

Junxue Zhang is with the iSing Laboratory, The Hong Kong University of Science and Technology, Hong Kong, SAR, China, and also with Clustar Company, Shenzhen 518071, China (e-mail: jzhangcs@connect.ust.hk).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNET.2023.3330737, provided by the authors.

Digital Object Identifier 10.1109/TNET.2023.3330737

Despite being promising, previous DRL-based CC schemes suffer from high computation overhead incurred by complex model inference for sending rate adjustment (§II-B). Such high inference overhead consumes non-negligible CPU resources [9] and interferes with the datapath performance [11]. Existing solutions handle the overhead issue by lowering the inference frequency to extend the response interval [8], [9], [10]. During the interval, the CC is out of DRL control, either making no rate adjustment [8], [10] or relying on a classic scheme, e.g., Cubic in Orca [9], as a remedy. Therefore, they fail to fully exploit the performance superiority provided by DRL models and is vulnerable to network congestions due to the coarse-grained control [11].

Given the above dilemma, we ponder a question: can a DRL-based CC provide fine-grained control for every ACK while preserving a low computation overhead? In this paper, we present SPINE to answer this question affirmatively, which is a fully DRL-based congestion control algorithm that can perform intelligently fine-grained packet-level control guided by deep neural networks with low overhead. SPINE adopts a more flexible paradigm for the RL agent to properly distribute its learning capability and computation across different granularity levels of the control policy.

At its heart, SPINE adopts a hierarchical control architecture consisting of a lightweight CC executor that reacts to every ACK and loss event as well as a DRL-based policy generator that periodically generates control sub-policies for the CC executor to adapt to the change of network condition, e.g., bandwidth capacity variation or flow arrivals and departures. Specifically, a sub-policy in SPINE is a lightweight parameterized control logic based on additive-increase/multiplicative-decrease (AIMD) and can be defined by a set of parameters outputted by the policy generator (§III-C). Compared to the network event reaction (e.g., ACK and packet loss), the sub-policy adaption for a new network condition is infrequent, resulting in low-frequency DRL model inference. Therefore, SPINE can perform a fine-grained DRL-based CC with a low computation overhead.

To support a timely network adaption while preserving a lower model inference overhead, SPINE leverages a flexible sub-policy update strategy by further introducing a *watcher* module, a smaller model compared to the policy generator (§III-D). The watcher judges whether the current sub-policy still works well and triggers the policy generator to update it if necessary. As a result, this flexible update strategy significantly reduces the execution frequency of the policy generator and potential cross-space (kernel and userspace) communications involved by sub-policy update operations,[1] especially in a stable network condition.

[1]The CC executor can be enforced in kernel while the DRL model is usually executed in userspace.

We have implemented a fully functional SPINE prototype in Linux. We have integrated the CC executor with Linux kernel TCP and implemented cross-space communication functionalities for the RL model to update its sub-policy. Based on this prototype, we performed efficient distributed training in various emulated network conditions and evaluated it extensively. Experimental results show that SPINE achieves consistent high performance across emulated networks and real-world testbeds with ultra-low control overhead. For example, when using SPINE with a large monitor interval of 300ms, SPINE stills achieves higher throughput and lower latency inflation compared to previous DRL-based CC schemes using a monitor interval of 30ms (§ VII-B). Meanwhile, it takes a much lower CPU utilization (2.6% for a single flow) reduced by at least 80% compared to its DRL-based counterparts (131.5% for Aurora and 14.3% for Orca) and comparable to classic CC Cubic (1.1%) (§ VII-A). In addition, SPINE also achieves good fairness property across homogeneous flows and friendliness to Cubic flows due to its dedicated training process (VII-D). Finally, SPINE has better interpretability than previous RL methods. By exploring the sub-policies decided by SPINE under different traffic patterns, we observe several insights that may help in the designing of future congestion control schemes (§ VII-C and § VII-E).

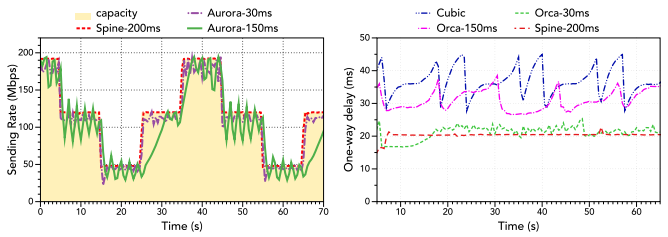
II. MOTIVATIONS

In this section, we explain the trend and difficulty of current learning-based congestion control algorithms and how the characteristics of CC motivate our hierarchical control logic.

A. DRL-Based Congestion Control

Unlike supervised learning algorithms that focus on prediction and classification tasks, reinforcement learning tackles sequential decision-making processes by maximizing the cumulative reward in the long run during the interactions with the environment. Armed with deep neural networks, deep reinforcement learning plays the core role in achieving super-human performance in many games and real-world decision-making tasks [1], [3]. Therefore, researchers have recently incorporated DRL into congestion control to seek to improve control policy and generalize to various unseen network conditions [8], [9], [10]. DRL-based congestion control schemes generate control actions using a well-trained RL agent. During the training, the RL agent receives packet statistics from the network environment and responds with action (e.g. adjusting $cwnd$ and sending rate). For every interaction step, the environment generates a reward regarding the performance metrics (e.g. throughput, latency, and loss in congestion control). With enough empirical experience, the agent learns to adjust its signal-response mapping by updating the deep neural network weights, so that the cumulative reward during the interaction can be maximized. While heuristic-based CC algorithms hand-craft signal-response mapping based on some specific assumptions, DRL-based methods learn the mapping automatically from the empirical data, thus showing better generalization and outperforming specially engineered methods across various network conditions [8], [9], [10].

DRL-based CC schemes also differ from online learning schemes such as Allegro [12] and Vivace [13], which leverage online learning techniques to adjust the sending rate. These methods make micro-experiments with the network by both increasing and decreasing the sending rates and obverse their performance results. Via comparing the difference, they decide the next sending rate towards higher performance. On the other hand, DRL-based CC focuses more on offline learning of the control policy offline that can be directly deployed to map network signals to sending rate adjustment, which is often more efficient and effective as the model will automatically learn the trade-off between exploration of the network and exploitation of performance instead of adopting heuristic-based online learning methods.



(a) Sending rates (b) Delay

Fig. 1. The performance of DRL-based CC schemes with different control intervals.

B. Overhead vs. Performance

Despite being promising, DRL-based models spend several orders of magnitude more time and consume much more computation resources than those simple ACK response functions in classic CC schemes. Empirical experiments in §VII-A have shown that previous DRL-based CC schemes consume much more CPU resources than classic ones such as Cubic, and the overhead increases with the control frequency. As a result, there may not be sufficient CPU resources to fully support the kernel datapath processing pipelines, resulting in degraded throughput, especially with multiple concurrent flows [11]. Furthermore, the intrinsic inference latency of the RL model defines a lower bound over the feasible control interval, and setting the control interval smaller than it will result in infinitely accumulated requests and system failure.

On the other hand, adopting DRL-based CC schemes with a coarse-grained control fails to fully unleash the potential of deep reinforcement learning. We perform experiments to demonstrate the performance degradation when the control frequency decreases. To demonstrate how quickly these schemes respond to network changes, we emulate a network with a minimum RTT of $30ms$ and buffer size of $375KB$, where the bottleneck link bandwidth changes every 10 seconds. A portion of a 10-minute trace is shown in Figure 1(a). We use a clean-slate DRL-based scheme Aurora [8] with different control intervals to send traffic over the network and record the sending rate. The results show that Aurora with a large control interval ($150ms$) can hardly respond to bandwidth changes quickly due to a less frequent $cwnd$ adjustment, compared to Aurora with a small control interval ($30ms$). Also, it tends to be more unstable under static bandwidth due to the response lag.[2]

One way to improve the trade-off between overhead and performance is to incorporate classic schemes back for fine-grained control. Orca [9] realizes it by building a two-level

[2]We also tested Aurora and Orca re-trained with a larger interval (150ms) and action range (5 * original action). However, due to the slow response, their model performance degrades severely with dramatic oscillation compared to their counterparts trained with an interval of 30ms.

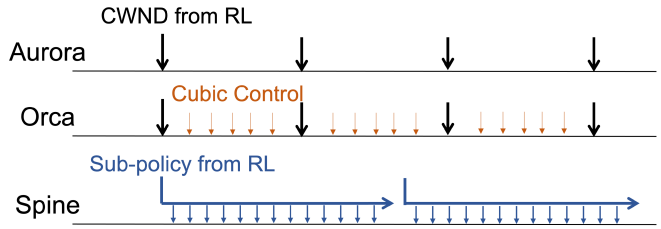


Fig. 2. The control logics of SPINE and previous DRL-based CC algorithms. By generating sub-policies instead of sending rates, SPINE is able to apply the DRL model to react to every ACK received with low overhead.

control framework where the RL model and the underlying scheme (Cubic) control the current $cwnd$ simultaneously at different frequencies. By restraining the sending rate according to delay increase, Orca avoids the bufferbloat issue caused by the loss-based CC Cubic. To show how Orca works with different control intervals, we emulate the same network with a static bottleneck link bandwidth ($100Mbps$) and record the delay. The result is shown in Figure 1(b). It is obvious that introducing Cubic will not eliminate the requirement of fine-grained control of the DRL model: when Orca interacts less frequently with the $cwnd$ (150ms), it can hardly restrain the sending rate increased by Cubic after every ACK, leading to a higher packet delay.

Is it possible to get rid of the trade-off between control granularity and model performance of DRL-based CC? Our observation is that all the previous DRL-based schemes focus on *directly adjusting the current sending rate*. As a result, the RL agent needs to perform two subtasks: i) to consistently update sending rate in high frequency to timely respond to dynamic bandwidth changes; ii) to adapt its current event-action mapping to the change of network condition when the link changes or flows arrive and depart. While DRL-based schemes are good at subtask (ii) for their capabilities to generalize to various network conditions, the overhead issue keeps them from responding to dynamic bandwidth changes quickly, resulting in sub-optimal performance. On the other hand, classic schemes are good at subtask (i) due to their simple hardwired signal-action mapping but can hardly automatically adapt to the network conditions where their assumptions no longer hold. Thus, we argue that in order to achieve high model performance with low overhead, we need to decouple the CC task and adopt a hierarchical architecture to handle the two subtasks in different timescales.

C. Key Design Decisions

Inspired by §II-B, our key design decision is to detach the time-consuming DRL processing from quick sending rate adjustment. To achieve this, SPINE builds a hierarchical policy structure powered by DRL, which learns a DRL model that generates sub-policies instead of sending rates. A sub-policy can be regarded as a parameterized mapping from packet-level events to sending rate adjustment, which is simple and fast compared to the DRL model, thus enabling instantaneously responding to network signals at fine-grained level. As shown in Figure 2, the DRL agent periodically observes the current network condition and generates a sub-policy. As the network changes, the DRL agent keeps updating the current sub-policy at a coarse-grained level. This design brings several benefits to solve the dilemma between performance and overhead as follows:

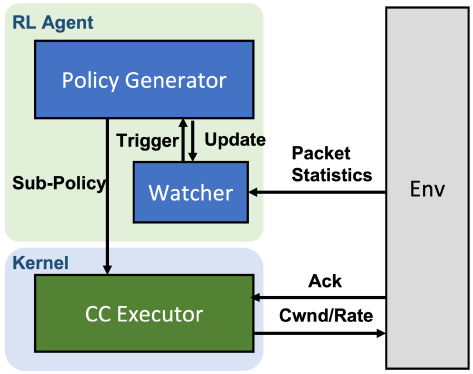


Fig. 3. The high-level architecture of SPINE.

- By generating sub-policy instead of volatile $cwnd$ and sending rate, we greatly reduce the required work frequency for DRL model, resulting in a much lower overhead than previous DRL-based CC algorithms. As a result, the overhead issue is no longer a severe concern when designing DRL-based CC algorithms. (§ VII-A)
- By learning to control every single response at fine-grained level through sub-policy, SPINE achieves consistent high performance across various network conditions, even in dynamic ones where the link capacity varies drastically. (§VII-B)
- The hierarchical policy architecture enables a more flexible policy update strategy. After inspecting the network condition, the DRL model can judge whether the current sub-policy still works properly. If so, the DRL model has no need to update the sub-policy, saving the cost of model inference and cross-space communication. This adaptive strategy further lowers the framework overhead significantly under stable network conditions (§ VII-E1).

III. DESIGN

A. Overview

Figure 3 overviews SPINE. It consists of three blocks: a *policy generator*, a *watcher*, and a *CC executor*. The policy generator and the watcher together form the RL agent that traces the traffic pattern and updates the current sub-policy. The CC executor is implemented in the kernel to enforce the control sub-policy generated by the RL agent, adjusting sending rate responding to ACK and packet loss. Observing the packet statistics collected, the watcher checks whether the deployed sub-policy is still working well under the current network condition. If so, the RL agent takes no action. Otherwise, it will trigger the policy generator module and submit a *report* encoding the network condition information. Once activated by the watcher and receiving a new report, the policy generator will output a new sub-policy and update it in the CC executor. In addition, it will update the watcher so as to continue supervising the updated new sub-policy.

As a result, SPINE utilizes a hierarchical control logic. As shown in Figure 4, the policy generator, watcher and CC executor run in different timescales. The CC executor provides fine-grained control to respond to every acknowledgement. For every monitor interval (MI), the watcher observes the packet statistics as the current state input of the RL agent and triggers the policy generator once in a while. Finally, the policy generator works in a flexible signal-driven style: it only updates the watcher and the executor when triggered.

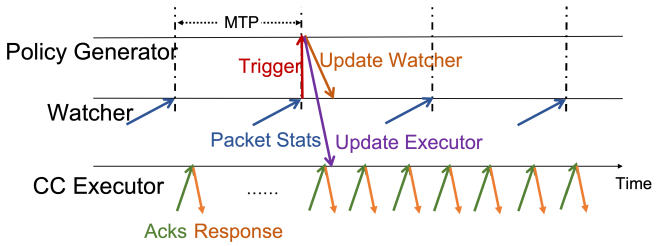


Fig. 4. The time diagram of SPINE's hierarchical control logic.

As a result, the policy generator will not be triggered when the current sub-policy works well and thus have a much lower average working frequency. In addition, the watcher is generally smaller compared to the policy generator due to its simple learning target and thus has a smaller regular computation cost than previous DRL-based schemes.

We will introduce each block in detail in the following subsections. §III-B introduces the basic components of the RL agent, including input state and reward definition. We leave the action definition to §III-C as it relates to the sub-policy used in the CC executor. §III-D introduces the hierarchical LSTM model architecture that combines policy generator and watcher modules in the RL agent, and its training algorithm is given in §V.

B. RL Agent

Both the policy generator and the watcher are integrated into the RL agent. By training the RL agent to maximize the reward accumulated over a flow's lifetime, we achieve two goals: i) we refine the policy generator to produce the optimal sub-policy capable of collecting the maximum reward under the current network conditions, and ii) we train the watcher to monitor network signal feedback and trigger the policy generator when the existing sub-policy no longer aligns with the network conditions. Specifically, the RL agent works as follows: in each monitor interval (e.g. t-th MI), it perceives the current network condition by gathering packet statistics, which are regarded as the current state s_t of the agent. That state input is then fed into the model based on deep neural networks, which will decide whether to update the underlying sub-policy by setting a flag $trigger$. If $trigger$ is $True$, the model will generate a new parameterized sub-policy by outputting its parameter setting a_t and update it to the CC executor by sending the parameter setting into the kernel. Otherwise, the model output is ignored and the sub-policy in the CC executor keeps unchanged. Once a flow is initialized, the RL agent consistently observes the dynamic network condition and updates sub-policy to maximize target performance goal.

State The input state of the RL agent contains collected packet statistics of the flow during the last MI. Here we consider the features that are closely related to the characteristic of the current network condition and flow status, which are shown in Table I. The throughput and delay are normalized with maximum observed throughput and the minimum observed one-way delay respectively. Beyond that, we also include the currently deployed sub-policy into the state, so SPINE can assess the performance of the current sub-policy for more intelligent update decision-making.

Some previous DRL CC schemes [8], [9], [10] stack a fixed-length of history features so that the agent can infer the current network condition more precisely by extracting information from history packet statistics. The history length is often small because the model size as well as its training difficulty will increase when the input feature dimension is augmented. However, as the policy generator of SPINE works in hundreds of milliseconds or even seconds, much longer than MI, we need a model architecture that is able to capture patterns and dependencies from long-term history features. Therefore, we adopt the recurrent neural networks (RNN) as the building block of the hierarchical policy structure (§III-D). RNN is able to capture patterns and dependencies from long-term history features without the need of stacking features, which is important for the policy generator and the watcher to memorize their state histories across sparse triggering events. As a result, we can directly feed the current state feature into the model without state stacking.

TABLE I
THE PACKET STATISTICS AS SPINE'S RL AGENT INPUT

thr	The average throughput.
lat	The average packet delay.
thr_{max}	The maximum throughput of the flow so far.
lat_{min}	The minimum packet delay of the flow so far.
$cwnd$	The current congestion control window.
$loss$	The average packet loss rate.
$flight$	The number of packets in flight.
p_{rate}	The average pacing rate.

Reward An RL agent needs to define a reward function to quantify the performance criterion of the task, which guides the agent to improve the generated sequence of sub-policies in the training phase. Updated with a new sub-policy from the policy generator, the CC executor interacts with the network environment by adjusting the flow sending rate and collecting rewards for each MI. Inspired by the Power-based reward in Orca [9], we define the reward function as follows:

$$R = \left(\frac{thr - \zeta \times loss}{lat'}\right) / \left(\frac{thr_{\max}}{lat_{\min}}\right) - \alpha_{psp} \times trigger, \quad (1)$$

where

$$lat' = \begin{cases} lat_{min} & (lat_{min} \le lat \le \beta \times lat_{min}) \\ lat & otherwise. \end{cases} \quad (2)$$

The first term in Equation 1 is based on the well-studied metric Power $\frac{thr}{lat}$ [14] and has been used in Orca. Generally, we can regard it as the ratio of normalized throughput to normalized latency, with a penalty on lost packets (weighted by ζ). References [15], [16] shows that by maximizing Power, both the network and individual flows achieve the optimal point. However, as the Power cannot be fully optimized in a decentralized way [17], a small queuing delay is allowed to achieve the maximum bandwidth. As shown in Equation 2, the parameter β controls the tolerance: when the latency is smaller than $\beta * lat_{min}$, no penalty of latency is incurred.

The second term defines the penalty of triggering the policy generator to update a new sub-policy, as it will cause further inference overhead of the policy generator and the cross-space communication for sub-policy update. We call the penalty term the *pit stop penalty*, as changing sub-policies takes overhead, and we want to avoid it if not necessary, which is similar to that of changing tyres in motorsports. Without the pit stop penalty, the RL agent cannot resist the temptation to switch sub-policies for every MI, even with negligible changes. The

indicator $trigger$ is equal to 1 when the watcher triggers the policy generator, otherwise 0. The coefficient α_{psp} defines the significance of the penalty. Empirical experiments in §VII-E1 show how this penalty controls the behavior of the watcher and thus the actual working frequency of the policy generator.

C. CC Executor

The Congestion Control (CC) executor in our system, SPINE, functions as one of the pluggable congestion control modules, comparable to Cubic and BBR. However, it uniquely executes sub-policies generated by the policy generator. To be more precise, the CC executor has already embodied the execution logic of the parameterized sub-policy. The policy generator's responsibility is solely to generate and update the hyperparameters of the sub-policy, which in turn, modulate the behavior of the CC executor. To meet the design goal of SPINE, we expect our parameterized sub-policy structure to have the following features:

- **Simple.** The sub-policy should be simple enough so that the CC executor can execute it in the kernel with very low computational overhead.
- **Fine-grained control.** The sub-policy should employ fine-grained control over the sending rate or $cwnd$ to quickly respond to dynamic bandwidth changes.
- **Flexible.** The sub-policy is able to approximate various control mappings from signals to sending rates, enabling the learning of arbitrary optimal policy.

Based on these feature requirements, we design a simple yet efficient sub-policy based on the idea of AIMD. It adopts a combination of the three most commonly used indicators in congestion control: received ACK, packet delay, and loss. While executing the sub-policy, the CC executor also performs an extra slow start function at the beginning in which the sender multiplies its sending rate by 1.1 for every RTT until packet loss occurs, similar with Orca. For the sub-policy execution part, when receiving an ACK packet, the CC executor updates $cwnd$ with the following equation:

$$\Delta cwnd = \begin{cases} -\alpha_{lat} & \frac{RTT}{RTT_{min}} \geq \alpha_{tol} + 1 \\ \alpha_{thr} & otherwise, \end{cases} \tag{3}$$

where $0 \leq \alpha_{thr}, \alpha_{lat} \leq 0.5$ and $0 \leq \alpha_{tol} \leq 2$ are hyperparameters. The behavior of Equation 3 can be described as this: given a new ACK, the CC executor will inspect the current RTT by calculating the ratio $\frac{RTT}{RTT_{min}}$. If the ratio is lower than $\alpha_{tol} + 1$, it judges the link is not congested and increases the $cwnd$ by α_{thr}. Otherwise, it decreases the $cwnd$ by α_{lat}. As a result, the sub-policy defines a target packet delay point and the sending rate adjustment aggressiveness towards it in two directions: α_{thr} controls the aggressiveness in increasing $cwnd$, α_{lat} controls the sensitivity to queuing delay and α_{tol} determines the target delay point that indicates the degree of tolerance for queueing. We note that in order to reflect the transit delay changes in fine-grained control for fast response, we estimate RTT with smoothed round-trip time $srtt$, which is different from the average packet delay lat of one MI in the reward function in §III-B.

When packet loss happens, the CC executor performs a multiplicative decrease of $cwnd$ by a factor of α_{loss}, similar to Cubic:

$$cwnd_{new} = \alpha_{loss} \times cwnd \quad 0 \leq \alpha_{loss} \leq 1, \tag{4}$$

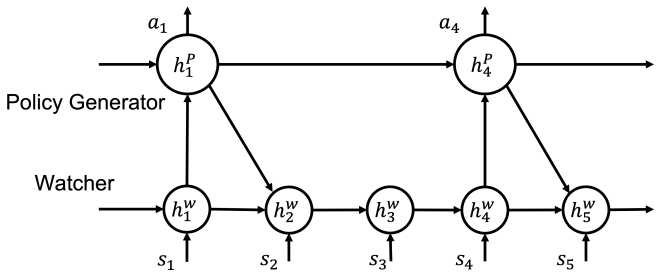


Fig. 5. The hierarchical recurrent neural network architecture with different timescales.

where α_{loss} indicates the sensitivity to packet loss event. After the $cwnd$ is updated, the CC executor calculates the new pacing rate as follows:

$$p_{rate} = \frac{cwnd}{RTT}. \tag{5}$$

With the above sub-policy structure, the parameter setting $(\alpha_{thr}, \alpha_{lat}, \alpha_{tol}, \alpha_{loss})$ determines the behavior of the sub-policy. We define the above parameter set as the action output of the policy generator a_t. When triggered, SPINE will update the sub-policy with a_t for the underlying CC executor that performs lower-level control. Through controlling the parameter setting, SPINE customizes the response of sub-policy to different signals to best suit the current network condition, as illustrated in §VII-C.

One criticism often levelled at DRL-based algorithms concerns their questionable ability to generalize to unseen environments. SPINE counters this critique by defining a sub-policy control mechanism and employing an RL model solely for hyperparameter tuning. This ensures that, during the training process, the search space for the policy of our system is restricted to a dependable policy pool, thereby considerably enhancing the system's ability to generalize to unobserved network conditions. For instance, the sub-policy is designed to increment the CWND until the experienced RTT ratio exceeds $\alpha_{tol}+1$, where $\alpha_{tol} \geq 0$. Consequently, our system is assured to fully utilize the link until RTT inflation occurs, provided there is only one flow present in the bottleneck.

D. Hierarchical Recurrent Architecture

In this section, we introduce the model architecture used in the RL agent. Inspired by the hierarchical multiscale recurrent neural network architecture (HM-RNN) proposed in [18], we design a hierarchical recurrent model, where both the watcher and the policy generator adopt recurrent neural networks as the basic building block and are connected with the report and update communications. We depict the two-layer model in Figure 5. The first layer represents the watcher, which is fed with input state s_t and adaptively triggers the upper layer. The second layer represents the policy generator that outputs the sub-policy parameter setting $a_t = (\alpha_{thr}, \alpha_{lat}, \alpha_{tol}, \alpha_{loss})$ mentioned in §III-C. Once triggered, it will receive the submitted report from the watcher layer and then i) generate a new parameter set a_t to update the sub-policy, and ii) update the watcher. By integrating the policy generator and the watcher together, we can perform gradient descent to collaboratively learn the model weights in both modules.

We use $h_t, o_t = f_{RNN}(h_{t-1}, x_t)$ to denote how an RNN block receives input and hidden state to update its

internal state and output new hidden state, which, in fact, can be instantiated with any popular recurrent neural network architectures. At time step t, the watcher receives the current state s_t and the hidden states from both itself and the upper layer (h^w_{t-1}, h^p_{t-1}) generated in the last time step. It then outputs i) z'_t that decides whether to trigger the upper layer, and ii) the new hidden state h^w_t.

$$h^w_t, z'_t = f^w_{RNN}\left(concat\left(h^w_{t-1}, h^p_{t-1}\right), s_t\right). \tag{6}$$

The binary trigger value z_t is then obtained by:

$$z_t = \begin{cases} 1 & z'_t \geq 0 \\ 0 & otherwise. \end{cases} \tag{7}$$

The policy generator works based on the value of the trigger z_t:

$$h^p_t, a_t = \begin{cases} h^p_{t-1}, a_{t-1} & z_t = 0 \\ f^p_{RNN}(h^p_{t-1}, h^w_t) & z_t = 1, \end{cases} \tag{8}$$

where a_t is the output sub-policy parameter setting. When triggered ($z_t = 1$), the policy generator takes the hidden state of the watcher as the input report to generate new sub-policy output. Otherwise, it just reuses the previous hidden state and output results, and the watcher reuses the old h^p_{t-1} for Equation 6.

The hierarchical policy model in SPINE is different from HM-RNN in several aspects: i) HM-RNN adopts the hierarchical structure to extract high-level representations for natural language modeling, where the higher layers model the long-term dependencies (e.g. sentences or paragraphs). In SPINE, we adopt the hierarchical architecture as the policy model for reinforcement learning, where the policy generator layer focuses on outputting long-term policies; ii) HM-RNN sets complex constraints on the communications between layers to automatically detect the boundaries between sentences and paragraphs. In SPINE, we simplify it to let the policy generator obtain all the available history information for policy decision-making.

IV. ANALYSIS

In this section, we perform a theoretical staleness analysis on SPINE with assumptions to show that controlling sub-policies makes the model less sensitive to control interval. We then present how the watcher module brings overhead reduction to the control system. Due to space limitations, the full proofs of the theorems in this section can be found in Supplementary Materials.

Modeling the trade-off between performance and control interval. For our CC algorithm working with fixed monitor interval T (i.e., the working frequency of the watcher/agent is $1/T$), it detects the network condition for each interval and updates a new sub-policy that fits best. To simplify the analysis, we assume the sub-policy works in binary mode: it either functions "well" or "badly" with respect to the current network condition,[3] and the RL agent has learned to always generate a "well" sub-policy for the current network condition. We also assume that the sub-policy will become stale as the network environment changes, which we define as a *policy drift event*. After the event, it will perform "badly" due to outdated sub-policy until SPINE updates it in the next MI. Thus, all time periods during which the sub-policy is stale (or fresh) are treated equally in terms of performance. We refer to the ratio of the effective time when the sub-policy functions well to the total time as the *effective time ratio* (ETR) and use it to measure the effectiveness of CC algorithms in this section. The following theorem shows how the effective time ratio relates to the dynamic of the network condition and the control interval.

[3]In our context, a good sub-policy denotes a policy that can collects more reward during its operative period.

TABLE II
THE EFFECTIVE TIME RATIOS (ETR) WITH DIFFERENT MIS AND POLICY DRIFT EVENT FREQUENCIES

T(sec)	$\lambda = 1$		$\lambda = 1000$	
	k	ETR	k	ETR
0.005	0.005	99.7%	5	19.9%
0.05	0.05	97.5%	50	2%
0.2	0.2	90.6%	200	0.5%
0.5	0.5	78.7%	500	0.2%

Theorem 1: Suppose the time between two policy drift events complies with an exponential distribution with parameter λ. Then, the effective time ratio of SPINE *with fixed MI length T is $\frac{1}{k}(1-e^{-k})$, where $k = \lambda T$ is the expected number of policy drift events happening during one MI.*

The distribution parameter λ defines the policy drift event frequency and thus how fast a sub-policy will become stale. It is decided by both the dynamics of the network environment and the sub-policy structure. A sub-policy that works well for a longer period should have a lower λ value. Based on the theorem, we present how the effective time ratio changes according to the MI length T and λ in Table II. The result intuitively shows that:

- Because the effective time ratio is defined by $k = \lambda T$, when adopting a control policy with a higher policy drift event frequency (large λ), the CC scheme needs a lower control interval (T) to stay performant. For example, when the policy drift event frequency is low (e.g., $\lambda = 1$, which means the sub-policy will become outdated after one second on average), we only need a course-grained control interval of 200ms to achieve effective control ($ETR = 90.6\%$). On the other hand, when the poilcy drift event frequency is high (e.g., $\lambda = 1000$), even with a 5ms control interval, the control policy can only achieve a ETR of 19.9%. If we regard adjusting $cwnd$ as the control policy, it will incur a much larger policy drift event frequency, as the sending rate needs to be updated quickly to respond to available dynamic bandwidth. Therefore, we have to define a much lower control interval to guarantee a performant policy, which will incur higher overhead, as discussed in the motivation section.
- The effective time ratio is much more sensitive to the value of k when it is larger. For example, when k changes from 0.005 to 0.5, the ETR of the policy only decreases moderately from 99.7% to 78.7%. However, when k further increases from 5 to 500, its ETR decrease drastically from 19.9% to 0.2%. Therefore, as the policy drift events that make our sub-policy outdated (e.g., link capacity changes or flow arrivals and departures) tend to happen less, SPINE will result in a lower policy drift frequency (lower λ) and thus achieve more robust performant congestion control.

Though SPINE's sub-policy can achieve a better trade-off between performance and control granularity than directly controlling the sending rate based on the above analysis. How to deliberately design a parametrized sub-policy with a minimum λ (i.e., being effective for the longest period) is an interesting topic that we hope to study in the future.

Watcher analysis. The following theorem shows when to add a watcher to lower the computation overhead of SPINE.

Theorem 2: With the assumption in Theorem 1, the costs of SPINE *with or without a watcher are equal if* $\frac{cost_w}{cost_p} = e^{-k}$, *where* $cost_w$ *and* $cost_p$ *are the costs of the watcher and the policy generator, and* $k = \lambda T$.

Thus, when k is small with our sub-policy strategy, a watcher model slightly smaller than the policy generator ($e^{-k}cost_p$) is enough to gain overhead benefit. For example, when $T = 200ms$ and $\lambda = 1$, the watcher design will lower the computation overhead of SPINE as long as its model overhead is less than $e^{-k} \approx 81.9\%$ of the policy generator.

V. TRAINING ALGORITHM

To design a DRL-based CC algorithm, we first formulate the CC problem as a reinforcement learning problem. At each t-th MI, the flow/agent sequentially interacts with the network environment in the following way: it observes packet statistics as the state $s_t \in \mathcal{S}$, and generates new sub-policy $a_t \in \mathcal{A}$ based on the agent policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$. The sub-policy responds to packet-level signals by adjusting sending rate during the next MI, and the flow will receive a reward r_t based on the reward function and newly collected statistics as the next state s_{t+1}. Though the formulation assumes that the agent outputs sub-policy for every MI, we will directly reuse the old sub-policy in the intervals when the watcher is not triggered according to our adaptively updated strategy. The goal of the agent is to maximize the expected cumulative reward during the sequence of interactions $\mathcal{J} = \mathbb{E}(\sum_{t=0}^{T} \gamma^t r_t)$, where γ is a discount factor to help the agent focus more on collected reward in the near future.

We adopt deep deterministic policy gradient (DDPG) [19], a famous model-free off-policy RL algorithm to learn sub-policies. During the training, the RL agent updates the model parameters of our hierarchical recurrent neural networks to adjust the mapping from packet statistics to sub-policy so as to maximize collected rewards. The key features of the training algorithm of SPINE are as follows. See Supplementary Materials for the complete training algorithm.

Stored hidden state and burn-in steps. In our system, SPINE, we leverage a recurrent model as the policy model. This model is designed to receive and generate hidden states, thereby encapsulating historical information efficiently. Nevertheless, traditional Reinforcement Learning (RL) training methods predominantly store interaction data, neglecting the hidden states generated throughout the process. This can potentially result in a loss of historical data and cause training instability. To circumvent these problems, we incorporate strategies proposed in [20]. Primarily, we preserve the recurrent hidden state within the accumulated trajectories. This preserved state is then used to initialize the policy model during training, ensuring a comprehensive inclusion of historical information. Further, during the training phase, while sampling sequences of interactions, we extract an additional segment from the start of the sequence, referred to as 'burn-in' steps. This additional segment solely participates in the forwarding phase, aiding in establishing a stable hidden state at the onset of the sample sequence. These strategies enable us to mitigate the training difficulties inherent to RNN models and enhance the effectiveness of our approach.

TABLE III
TRAINING ENVIRONMENT PARAMETERS

Bandwidth	One-way delay	Buffer size	Loss rate
40-200 Mbps	10-100ms	0.5-2 BDP	0-1%

Probabilistic trigger Two problems exist in training the trigger unit of the watcher: i) It is non-differentiable. The derivatives of Equation 7 are zero almost everywhere, so gradient back-propagation cannot proceed; ii) As exploration plays an important part in reinforcement learning to collect rich experience, when untriggered, the deterministic trigger unit prevents the agent from exploring more diversified sub-policy decisions (e.g., "Can I challenge the status quo with a better sub-policy?"). Therefore, during the training, we inject noise into the unit by replacing the deterministic trigger unit in Equation 7 with a probabilistic one during training:

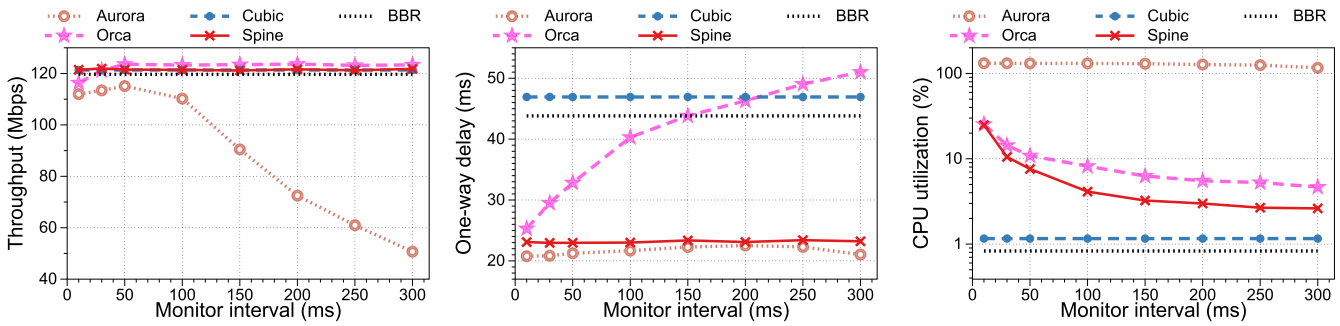
$$z_t \sim Binomial(z'_t). \tag{9}$$

It is plain to see that $\mathbb{E}(z_t) = z'_t$ and $\frac{d\mathbb{E}(z_t)}{dz'_t} = 1$. Thus for back-propagation, we let the gradient simply go through the unit without change as if it is an identity function so that the watcher can learn to update its triggering strategy. As a result, SPINE is able to explore the possibility of improving the current sub-policy generated by the policy generator, even if it performs poorly for the time being and the watcher does not recommend it through a low z'_t value. Based on the probabilistic unit, we will run the policy generator layer for every time interval during the training phase to update a better sub-policy to replace the current one. While in the inference phase, the policy generator only works when $z_t = 1$, thus saving most of incurred inference overhead.

VI. IMPLEMENTATION

Model Architecture We use Pytorch [21] to build the hierarchical policy model in §III-D, where LSTM is used as the building block for the recurrent model. The LSTM layers in the watcher and policy generator consist of 64 and 128-dimensional hidden state vectors, respectively. In the policy generator, we feed the LSTM layer hidden state h_t^p into an MLP layer and a tanh layer to get the output action a_t. The critic model used during the training (see the training details in Supplementary Materials) also adopts a single LSTM layer with a 128-dimensional hidden state vector. For the CC executor, we implement a congestion control module inside the Linux kernel TCP stack which receives control parameters from the RL agent and performs ACK-level congestion control. Inspired by CCP [22], the CC executor and userspace RL model are communicated via netlink [23].

Training Our training implementation is based on a generalized RL training framework DI-engine [24], which supports various DRL algorithms and customized environments and policies. We build our emulated congestion control training environments based on Pantheon [25], where Mahimahi [26] is used to emulate various network conditions. The range of settings of the training environments is shown in Table III.



(a) Throughput (b) Latency (c) Overhead

Fig. 6. The performances and overheads of SPINE and previous DRL-based algorithms.

TABLE IV
TRAINING HYPERPARAMETERS IN SPINE

Name	Value
learning rate (α)	0.005
gamma (γ)	0.98
batch size	64
model update interval (second)	5
model update step	20
action control coefficient (α)	0.05
monitoring time inverval (ms)	30
pit stop penalty (α_{psp})	0.03

We also add a random number of Cubic flows as the background traffic. In order to stabilize the training process, SPINE adopts episode-based training. Each episode consists of two phases: the collecting phase and the learning phase. In the collecting phase, several distributed collectors perceive states and enforce actions in various training environments to collect experiences for a period of time, which are stored in a data structure called replay memory [1]. In the learning phase, the centralized learner samples experiences from the replay memory for training. As a result, the learner updates the model only after the collectors have collected enough new experiences with sufficient length, and the experiences collected in one episode are from the static RL agent with no update. Therefore, the learning process will be less stochastic and the model will converge faster. We use 8 actors to collect the training experience in parallel. The entire training hyperparameter set is given in Table IV. We train and evaluate SPINE on a Linux server with 80 CPU cores, 256GB RAM, and equipped with NVIDIA GeForce RTX 3090 GPU.

VII. EVALUATION

In this section, we evaluate the performance of SPINE with emulated and real testbed experiments. In §VII-A, we show how SPINE preserves high performance with low DRL model inference frequency due to its insensitivity to monitor interval. In §VII-B, we demonstrate that SPINE achieves consistent high performance across a wide range of network conditions, including dynamically changing ones. For a better understanding of the control logic of SPINE, we inspect how SPINE updates its sub-policy in §VII-C. We evaluate the convergence properties of SPINE in §VII-D. Finally, we inspect the improvement brought by the watcher module and explore more possibilities of SPINE in §VII-E.

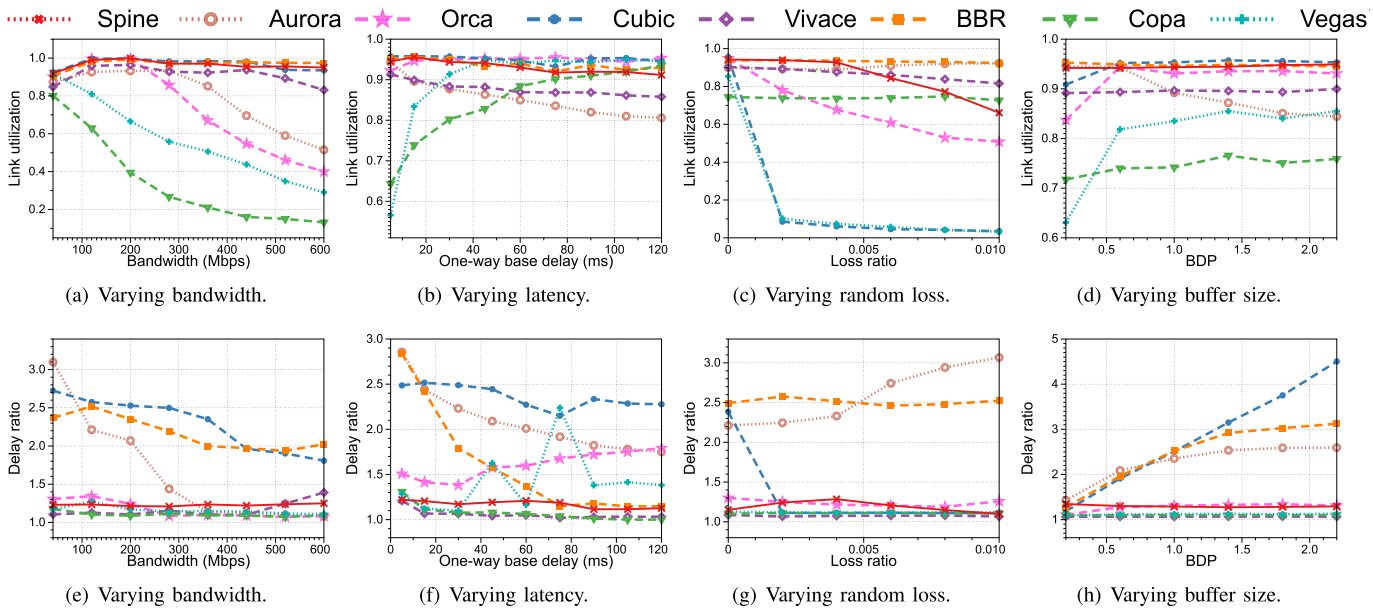
Evaluation setup. In emulated experiments, unless specified otherwise, we establish the network as the dumbbell topology with one single flow by default. The emulated bottleneck is implemented with Mahimahi [26]. In the real-world experiments, we turn to Pantheon [25] framework and deploy the sender and receiver at two AWS nodes.

A. Monitor Interval Insensitivity

To understand how SPINE reduces the overhead without undermining performance by lowering the control interval, we continue the motivation experiment in §II-B and evaluate SPINE and previous DRL-based algorithms with different MIs. We use a larger buffer size (750KB) to allow enough latency inflation to indicate the congestion. We repeat each trial 10 times and report their average performance (throughput and latency) and the corresponding overheads in Figure 6.[4] The performances of Cubic and BBR are also shown as baselines. With the results of the experiment, we identify the following key observations:

- With the increase in monitor interval, the performance of previous DRL-based CC algorithms degrades: Aurora fails to achieve full bandwidth utilization (Figure 6(a)). The reason is that its sending rate is directly controlled by the RL model, which is unable to respond to bandwidth changes when the control frequency is low, and the control action may easily become stale. Orca, on the other hand, though achieving low latency with high frequency, has even higher latency than Cubic with larger MIs (Figure 6(b)). The reason is that, without a fine-grained control granularity, Orca can hardly restrain the increasing $cwnd$ promptly, and thus keep experiencing high queuing delay.
- The performances of previous DRL-based CC algorithms also degrade when the MI decreases to less than 30ms. We inspect the implementations of these algorithms and find that they generally take more than 10ms for the model inference and cross-space communication between the user space and the kernel. Thus, we think the reason for the degradation is that when the MI is small enough to be similar to the running time for enforcing new action, the actual time between when the action is enforced and when the next state is collected is very small. Therefore, there is not enough time left to collect effective feedback, and the agent cannot make timely intelligent decisions.
- SPINE's high performance is insensitive to the changing of MI. As shown in Figure 6(a) and Figure 6(b), SPINE yields consistent high link utilization with small queue in bottleneck across various MIs from 20ms to 300ms. The reason is that SPINE imbues control response to every

[4]The variance of the repeated results are within $\pm 5\%$.



(a) Varying bandwidth. (b) Varying latency. (c) Varying random loss. (d) Varying buffer size.

(e) Varying bandwidth. (f) Varying latency. (g) Varying random loss. (h) Varying buffer size.

Fig. 7. The performance of SPINE and other CC algorithms in terms of link utilization and latency under varying bandwidth, base delay, loss rate and bottleneck buffer size. The link utilization is defined as the ratio of throughput to the link capacity, and the delay ratio as the ratio of packet delay to the base one-way delay.

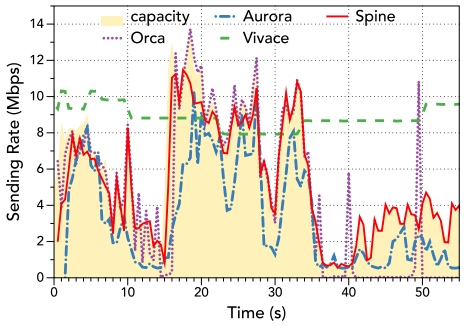


Fig. 8. SPINE demonstrate good reactivity.

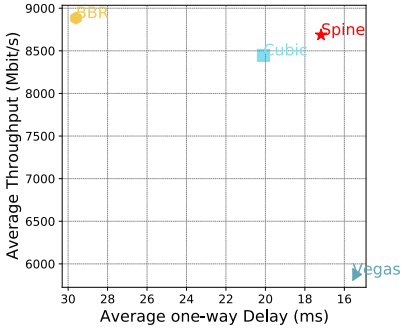


Fig. 9. The throughput vs. latency in 10Gbps high-speed network.

ACK with reinforcement learning intelligence, whose actual control frequency is independent from and much higher than the MI used. The results illustrate the important role of the hierarchical policy structure in SPINE's design. Even with a much lower control frequency (e.g., MI of 300ms), SPINE's sub-policy in the CC executor can still properly adjust the sending rate at a packet-level control level and thus can adapt to bandwidth changes agilely.

- With the increase of monitor interval, the overheads of SPINE and DRL-based CC algorithms decrease due to lower inference frequency, as shown in Figure 6(c). The extent of overhead decrease also depends on their implementations. For example, the major overhead in Aurora may result from its inefficient userspace implementation.

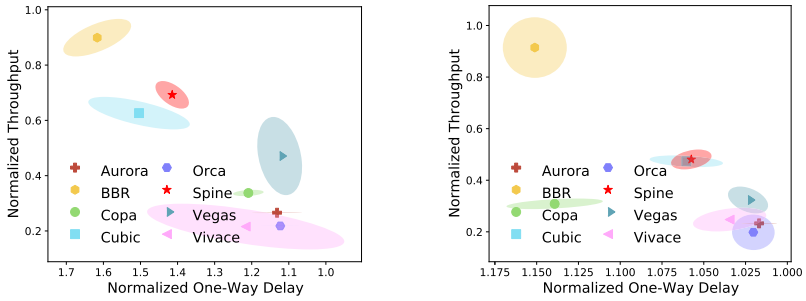
The insensitivity property to control interval enables the deployment of SPINE with an ultra-low working frequency without undermining model performance, which therefore achieves a much lower model overhead compared to other DRL-based algorithms. For example, when using SPINE with a monitor interval of 300ms, it achieves better performance than previous DRL-based solutions (e.g., Orca) using monitor intervals of 30ms but with a much lower CPU utilization (from 10.5% to 2.6%), which is comparable to heuristic-based algorithms such as Cubic (1.1%).

B. Consistent High Performance

Here, we evaluate SPINE with extensive emulations and real-world experiments to show its consistent high performance across various network environments. We repeat each test 10 times and report the average values. We use Orca and Aurora with MI of 30ms as it preserves relatively high performance, as shown in §VII-A. For SPINE, we use MI of 200ms, which has a much lower overhead due to its low inference frequency. We also compare SPINE with other heuristic-based CC schemes including Cubic [27], BBR [28], Copa [29], Vegas [30] and online learning scheme Vivace [13].

1) Diverse Emulated Networks: We first compare SPINE with other CC algorithms across a wide and diverse range of emulated networks by demonstrating the link utilization and latency ratio with varying bandwidth, base delay, random loss rate, and buffer size. Specifically, we alter one link characteristic of them at a time while holding the other three constant, and compare SPINE with other baselines. For constant values, we use the bandwidth of 100Mbps, base RTT of 30ms, buffer size of 1 BDP (Bandwidth-Delay Product), and no random loss rate. The average results of 10 trials are shown in Figure 7.[5]

[5]The variance of the repeated results are within ±5%.



(a) Intra-Continental Experiments (b) Inter-Continental Experiments

Fig. 10. Overall normalized throughput vs one-way delay in real-world. The normalization bandwidths are both 1200Mbps and normalization one-way delays are 35ms and 115ms, respectively.

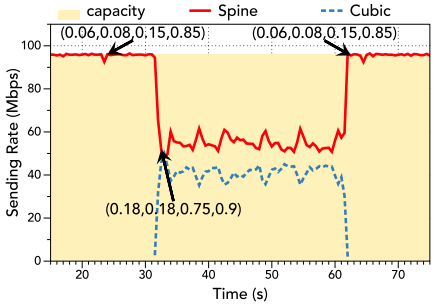


Fig. 11. Details of SPINE's sub-policy when competing with Cubic.

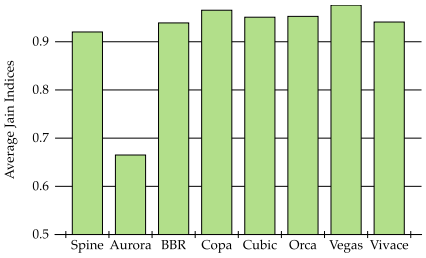


Fig. 12. Jain indices of competing flows.

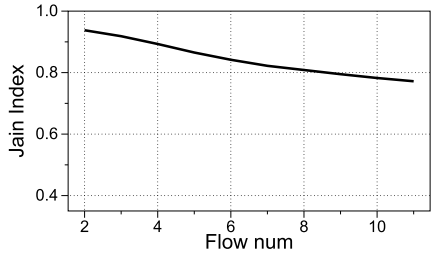


Fig. 13. SPINE's fairness across the different number of flows.

We observe that SPINE achieves consistent good performance across different bandwidths, latencies, random losses, and buffer sizes compared to other CC baselines. For example, when changing bandwidth, SPINE achieves high throughput similar to Cubic and BBR, which, however, both incur much larger queuing latency with a delay ratio from 2 to 3. The performance of previous DRL-based schemes Aurora and Orca degrades when the bandwidth or the base delay is large. This may be due to their limited generalization capability as they have not been trained in links with large BDPs. On the other hand, SPINE exhibits promising generalization ability with a limited range of training environments, which, we think, can further be improved with extensive training data from the wild Internet. We attribute SPINE's good generalizability to the introduction of domain-specific knowledge of congestion control, as discussed in §III-C.

When we vary the random loss rate, schemes that use packet loss (Orca, Vivace, Vegas, Cubic, and SPINE) as congestion signals reduce their bandwidths apparently. Because SPINE learns to use both latency and loss as signals to detect network congestion, non-congestion loss alone has a limited effect on the decision-making of SPINE. For example, when the random loss rate is 1%, the link utilization of Cubic drops dramatically (from 0.95 to 0.03), and that of SPINE decreases much slower (from 0.95 to 0.64). For varying buffer size, SPINE achieves small latency inflation and near full bandwidth utilization with buffer size from 0.2 to 2.2 BDP. We attribute the low requirement of the buffer size of SPINE to its fine-grained control and thus a stable queue. On the other hand, the link utilization of Orca reduces to 80% in shallow buffer (0.2 BDP), as the MI of 30ms for Orca may not be enough to restrain the underlying Cubic without causing latency inflation and loss.

2) Reactivity: To evaluate how SPINE reacts to dynamically changing network conditions, we create an emulated link with a trace from LTE network [31] with dramatically changing capacity. We use base RTT of 30ms, and an adequate buffer to absorb the traffic. As illustrated in Figure 8, SPINE surprisingly achieves good reactivity with a large MI of 200ms: it achieves the highest bandwidth (5.08Mbps). Meanwhile, SPINE can achieve the lowest latency (RTT=78.1 ms) among other learning-based CC schemes, which, however, all use a small MI of 30ms (Vivace defines RTT_{min} as its MI). We attribute the good reactivity of SPINE to its hierarchical policy structure, where the sub-policy performs a fine-grained control with a low DRL model inference frequency.

3) High-Speed Networks: To assess the performance of SPINE in high-speed network environments, we construct a dumbbell topology with high-speed connections in our testbed. We establish a connection between two end-hosts with an NVIDIA ConnectX-5 NIC paired with a Mellanox SN2700 100G switch powered by Intel Xeon(R) Gold 5218R CPU. To emulate a real-world high-speed WAN scenario, we cap the receiver bandwidth at 10Gbps and introduce an additional one-way latency of 15ms (with tc + citation). Due to their implementation overhead, the bandwidths of previous learning-based schemes Aurora, Orca, and Vivace are restricted to a maximum of 1Gbps. The results for throughput and latency of SPINE and other baselines are depicted in Figure 9. We observe that SPINE exhibits comparable performance with BBR and Cubic, closely matching the full capacity of the available link bandwidth while achieving much lower latency. The results underscore that SPINE's

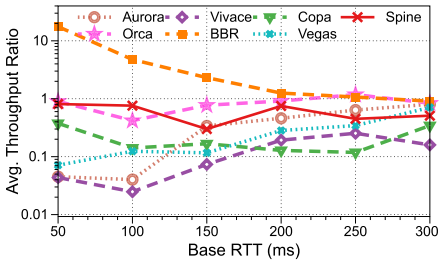


Fig. 14. Throughput ratios to Cubic across RTTs.

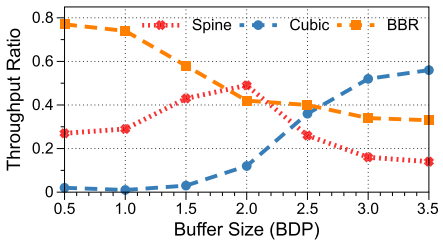


Fig. 15. Throughput ratios to bandwidth of SPINE, Cubic and BBR across buffer sizes.

performance remains robust in high-speed networks, even when operating at bandwidths significantly beyond its training region.

4) Real-World Experiments: For real-world evaluation of SPINE, we follow the experiment settings in Orca [9] and evaluate SPINE in the inter-continent scenario and intra-continent scenarios. We deploy the sender at AWS Seoul and locate the receiver at AWS Singapore and London to vary the experiment environment. The bandwidths of both links are up to 10Gbps. We evaluate each CC scheme by running its one flow for 60 seconds, repeating each trial 10 times, and summarize the overall average normalized throughput and one-way delay in Figure 10.

We observe that SPINE defines one of the frontiers in terms of high throughput and low latency: it achieves better link utilization than most CCs, including Cubic. Meanwhile, it delivers smaller latency than Cubic in both intra-continental (10(a)) and inter-continental (10(a)) scenarios, respectively. The reason is that SPINE adopts the DRL model to adjust the sending rate at a fine-grained level and thus can rapidly adapt to Internet bandwidth fluctuation without incurring bufferbloat. On the other hand, though performing well in the emulated experiments, other learning-based algorithms such as Orca, Aurora, and Vivace all fail to achieve high utilization, which has also been observed in emulated experiments in §VII-B1. The superior performance of SPINE among learning-based schemes validates the advantage of our hierarchical policy structure to imbue ACK-level control with RL intelligence.

C. Under the Hood

In this section, we take a deeper look at the behavior of SPINE to understand how it updates the sub-policy to adapt to various network conditions. One line of criticism of learning-based algorithms is their poor interpretability, which hinders researchers and engineers from inspecting cases with poor performance and improving the algorithm. However, SPINE provides semantically task-related sub-policy parameters for lower-level congestion control, which, as we illustrate in this section, will provide insights into the design of a better CC algorithm.

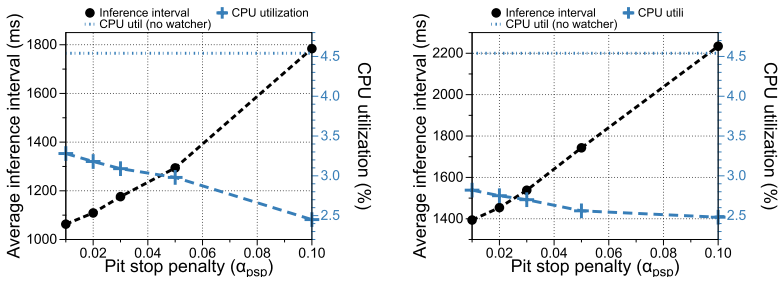
To shed light on how SPINE "thinks" during the control process, we run a SPINE flow on an emulated link of 100Mbps with 30ms base RTT and one BDP buffer. We start a Cubic flow during this process and inspect how SPINE responds by updating its sub-policy. We show its behavior in Figure 11 and mark the updated parameter for the sub-policy on the figure. We observe that SPINE starts with low α_{thr} (0.06) and α_{lat} (0.08) to enforce a moderate sending rate adjustment strategy (see §III-C for meaning of parameters). It also adopts a low α_{tol} (0.15) to maintain low latency inflation ($1.2\times RTT_{min}$). When the Cubic flow goes in, SPINE detects sudden inflation at packet delay. Thus, it updates the sub-policy with more aggressive parameters ($\alpha_{thr} = 0.18, \alpha_{lat} = 0.18$) to quickly adjust its $cwnd$ responding to ACK and delay inflation signals. Meanwhile, it adopts a higher α_{tol} (0.75) and α_{loss} to enable tolerance on higher latency inflation and possible packet loss. When the Cubic flow exits, SPINE then restores its conservative sub-policy. One interesting observation is that SPINE maps its packet statistics, primarily the link rate and the maximum throughput, to a target delay point. When the SPINE flow detects that its throughput decreases and latency increases under a conservative sub-policy, it judges that other aggressive flows come and thus resets its delay equilibrium point to a larger value until a new consensus on the queuing delay is achieved. Therefore, though working as a delay-based scheme, SPINE is able to grab bandwidth from Cubic with its flexible delay target.

D. Fairness and Friendliness

In this section, we seek to understand how SPINE performs when competing with other SPINE flows and with Cubic flows. First, we setup a 100Mbps link with 30ms RTT and 1 BDP buffer, and we start three flows with a running time of 120s, for whom we set the inter-arrival time to be 40 seconds. We repeat each experiment 10 times and calculate the average Jain Index of each CC algorithm in Figure 12.

We observe that by adopting a Power-based reward that is maximized at the fair operating point, SPINE achieves better fairness than the clean-slate DRL-based CC Aurora but is still not good enough compared to other classic schemes and Orca, which incorporates Cubic to provide fairness property. To further explore the generalizability of SPINE's fairness property, we also increase the number of competing flows from 2 to 10 and record their Jain indices in Figure 13. The results show that SPINE's fairness stays decent yet slowly decreases with the increasing of number of competing flows. The reason is that the current DRL-based algorithms have not learned toward a fair scheme and have no provable guarantee of fair convergence. While we focus on reducing overhead and improving the performance of DRL-based scheme in this paper, we believe the fairness issue of DRL-based scheme will soon be solved in the near future, as several works have been devoted to learning fairness for deep reinforcement learning recently [32], [33], [34].

We also study the TCP friendliness of SPINE by competing it with one Cubic flow under different base RTTs. We use the same link setting as we use in the fairness part and tune the buffer size correspondingly with respect to varying RTT. Figure 14 plots the ratio of throughput of evaluated CC to the throughput of the Cubic flow. We observe that SPINE achieves good friendliness to Cubic. The reason is that we have added Cubic as the background traffic during the training, so that



(a) Trace 1 (square wave) (b) Trace 2 (static)

Fig. 16. The actual average inference interval of the policy generator changes with different pit stop penalty values.

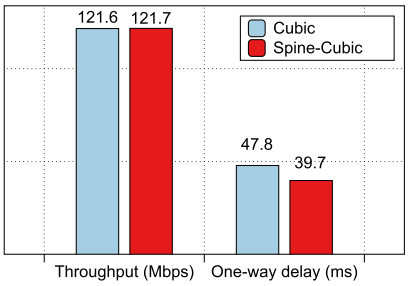
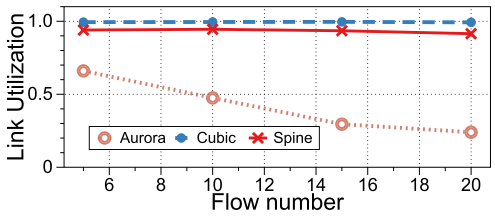
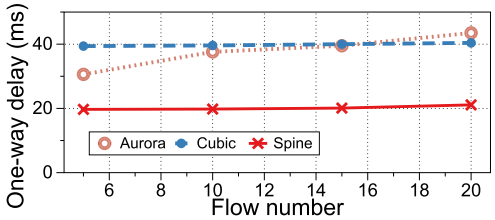


Fig. 17. Spine-Cubic improves Cubic by adaptively tuning its hyperparameters.



(a) Link utilization



(b) One-way delay

Fig. 18. The average link utilization and latency with concurrent flows.

SPINE learns to become more aggressive when competing with Cubic flows, as illustrated in §VII-C.

Considering the increasing prevalence of BBR in real-world Internet scenarios, we broaden our investigation to explore the friendliness of SPINE in a mixed scenario. This experiment involves a SPINE flow, a Cubic flow, and a BBR flow. Keeping the link setting consistent with the TCP friendliness experiment, we fix the RTT at 50ms and run for 5 minutes. Figure 15 presents the ratios of overall throughput to bandwidth across varying buffer sizes. In scenarios where the buffer size was small, BBR predominates over Cubic, with SPINE demonstrating compatibility with BBR. As the buffer size escalated, Cubic, being a loss-based scheme, adopts a more aggressive policy, subsequently claiming more bandwidth. During this transition, SPINE exhibits a comparatively gentle behavior towards the other two flows. The results bolster the demonstration of SPINE's adaptability and friendliness in mixed traffic scenarios.

E. Deep Dive

1) The Watcher and Pit Stop Penalty: We next demonstrate how the watcher affects the working frequency of the policy generator as well as the overall overhead. As mentioned in §III-B, the pit stop penalty α_{psp} controls the trade-off between overhead and the triggering frequency of the policy generator. Therefore, we retrain SPINE with different α_{psp}, and evaluate both their overheads and the actual average inference intervals of the policy generator. These models are evaluated on the two traces used in the motivation experiment (§II-B), where the first trace has a more dynamic link capacity (square wave) than the second one (static). To set a baseline, we also train and evaluate SPINE without a watcher module, which means the policy generator directly receives the input state and outputs sub-policy for every MI. We fix the MI of all models as 200ms.

Figure 16 shows the results. We observe that adding a watcher will consistently and significantly reduce the working frequency of the policy generator. For example, when α_{psp} is set to be 0.03 (that is used in previous evaluation sections), we can decrease the actual inference interval from 200ms to 1.2 seconds and 1.5 seconds in both traces. As a result, the CPU utilization is further reduced by almost 40% compared to the baseline with no watcher. Furthermore, with a larger penalty value, the trained model tends to trigger the policy generator less frequently, which would further lower the CPU overhead. It is because the watcher will learn not to trigger the policy generator unless it can obtain adequate performance gain over the pit stop penalty. Therefore, the penalty allows for small performance variation of the current sub-policy. However, we do not observe obvious performance degradations until using a very large penalty value ($\alpha_{psp} = 0.3$). Finally, we find that the interval reduction is larger on the second trace. For example, when $\alpha_{psp} = 0.03$, the inference interval for the second trace is 1.54 seconds, 30% larger than that in the first one (1.17 seconds). The result validates our assumption that the watcher tends to update sub-policy less frequently under stable network conditions. In practice, the choice of the pit-stop penalty is largely influenced by factors including the policy generator's overhead and network conditions. Specifically, lower generator overhead can accommodate a higher α_{psp}, enabling better control granularity with acceptable overhead increase. Moreover, in stable network conditions, a lower α_{psp} can still maintain a modest policy update frequency, reducing control overhead.

2) SPINE-X: Though in the paper, we mainly evaluate SPINE using a sub-policy based on AIMD, SPINE can also adopt other customized sub-policies and even classic CC schemes by tuning the knobs of the fine-grained control process. Thus, we can generalize our framework to SPINE-X,

TABLE V
THE THROUGHPUT AND DELAY OF FLOWS IN LONG-SHORT EXPERIMENT. THE DELAY RATIO IS DEFINED AS THE RATIO OF THE AVERAGE ONE-WAY PACKET DELAY TO THE BASE ONE-WAY DELAY

Flows	SPINE		Cubic	
	Thr. (Mbps)	Delay ratio	Thr. (Mbps)	Delay ratio
Overall	184.2	1.79	194.7	2.39
Per Long flow	10.5	1.79	6.2	2.39
Per Short flow	10.8	1.87	11.8	2.41

TABLE VI
THE THROUGHPUT AND DELAY OF FLOWS IN HETEROGENEOUS RTT EXPERIMENT

Flows	SPINE		Cubic	
	Thr. (Mbps)	Delay ratio	Thr. (Mbps)	Delay ratio
Overall	193.2	/	197.3	/
Per small-RTT flow	6.6	2.17	13.8	2.61
Per large-RTT flow	15.1	1.40	8.3	1.49

where X can be any parameterized sub-policy. To further explore the potential of our hierarchical policy structure with existing CC schemes, we adopt a parameterized Cubic algorithm as the sub-policy in our framework, where the RL model updates the multiplicative factor β and the cubic function coefficient C of Cubic. We call it SPINE-Cubic, and evaluate it under the motivation trace in §VII-A. The average throughput and one-way delay are shown in Figure 17. As expected, with the help of our hierarchical policy structure, the parameterized Cubic scheme is able to achieve lower delay while preserving full bandwidth utilization. The reason is that SPINE-Cubic adaptively controls the aggressiveness of the underlying Cubic through β and C over changing network environment. Thus, SPINE-Cubic is more flexible than the original algorithm with fixed hyperparameters. This improvement provides a great opportunity to adopt SPINE-X as an auxiliary tool to automatically tune current CC schemes regarding various network conditions.

In addition, we observe that SPINE-Cubic also inherits the limitation of Cubic. Because SPINE-Cubic only responds to packet loss to cut the congestion window, it will still be hard to mitigate bufferbloat and distinguish non-congestion loss from congestion loss. Therefore, though classical heuristic-based CC algorithms can be improved by SPINE framework, SPINE-X may still hold the drawbacks of the underlying sub-policy. It remains an open problem how to design a flexible and simple sub-policy with the minimum assumptions of the network environment.

3) Scalability: In this section, we further explore the benefits brought by SPINE's hierarchical design through the scalability experiments. We start N concurrent flows simultaneously (N = 5,10,15,20) on an emulated link of 200Mbps bandwidth, 30ms base RTT and 1 BDP buffer. 4 CPU cores are dedicated exclusively to this set of experiments to evaluate the scalability. Aurora and Cubic are also tested as baselines. We record the overall link utilization and latency, as shown in Figure 18. We observe that with the increased number of concurrent flows, the aggregated throughput of Aurora, the naïve DRL-based CC scheme, decreases. The reason is that due to the high inference overhead of multiple Aurora flows, the remaining CPU resource is insufficient to support the pipelined transmissions in the datapath. On the other hand, SPINE achieves consistently high performance with up to 20 concurrent flows, similar to Cubic.

Furthermore, we also extend the scalability experiment on heterogeneous flows, with respect to flow running times and per-flow RTTs. First, we inspect the performance of short flows of SPINE co-existing with several long-running flows, which is the common case in the Internet. Starting with the previous scalability experiment setup, we vary the running time of flows so that the concurrent flows consist of 80% short flows and 20% long flows. Specifically, we initialize 4 long flows running throughout the trial and a lot of short flows that arrive and depart in a very frequent manner,[6] so that there are almost 16 short flows co-existing with 4 long flows for a long period (100 seconds). We repeat the experiment 20 times and report the average throughput and latency for both types of flows, and the overall throughput of all flows in Table V. We can see with the slow-start phase, short flows of SPINE can grab the bandwidth quickly and achieves similar throughput to that of long flows. Second, we conduct experiments to investigate the performance of multiple SPINE flows with heterogeneous RTTs. With the same experiment setup, we start 10 large RTT flows (90ms) and 10 small RTT flows (30ms) running simultaneously for 100 seconds. We repeat the experiment 20 times and record their throughputs and delay ratios in Table VI. We observe that SPINE flows with small RTT share lower bandwidth than those of large RTT. In addition to the aforementioned fairness issue, another reason for this result is that SPINE flows with large base RTT, according to Equation 3, tend to have a target delay point tolerating larger queuing delay under the same sub-policy. Differentiating the behaviors of flows with heterogeneous RTTs and improving RTT fairness may be a future direction of DRL-based CC schemes.

VIII. RELATED WORK

The congestion control task has been an enduring hotspot in the networking research field for more than three decades with

[6]Short flows come following the exponential distribution of $\lambda = 4$ and their running times are drawn from the Gaussian distribution $\mathcal{N}(4, 1^2)$.

a plethora of CC algorithms. The classic schemes [27], [29], [30], [35], [36], [37], [38] are generally designed based on heuristics about how should we respond to specific congestion signals in specific cases and thus are often categorized as heuristic-based schemes. For example, loss-based schemes such as Cubic [27], TCP Tahoe, and TCP Reno [35] respond to packet loss events by cutting the congestion window. On the other hand, delay-based schemes such as TCP Vegas [30] and Copa [29] respond to delay changes to keep the queuing delay low. Heuristic-based algorithms require careful hand-crafting of the signal-action mapping and can backfire under network conditions where the heuristics are violated.

There are also CC solutions focusing on specific types of networks. For example, Sprout [31], Verus [39] and ExLL [40] focus on highly variable LTE network with self-inflicted queuing delays and packet losses. On the other hand, works such as Timely [41], HPCC [42], DCTCP [43] and ICTCP [44] focus on data center networks with ultra-high bandwidth and low latency. However, as these methods are deliberately designed for the characteristics of the target network scenario, they can hardly be adapted to other network conditions without a significant amount of time for hand-crafted tuning. For example, BBR [28] was firstly invented by Google for inter-datacenter communications and then extended to other network environments by Google engineers after a few years of manually engineering.

Recent years have seen a plethora of learning-based CC algorithms due to the rising of machine learning [6], [8], [9], [12], [13], [25], [45], [46], [47]. For example, PCC Allegro [12] and Vivace [13] utilize an online learning paradigm. Different from machine learning solutions, they adaptively optimize pre-defined utility functions by exploring various sending rates and observing feedback from the network. However, the online exploration phase of them takes several RTTs to collect empirical performance evidence, preventing them from reacting quickly to signals timely, especially when the RTT is large and the network changes rapidly. DeepCC [45] also adopts two-level logic with DRL agent and Cubic, where the agent learns to enforce the maximum congestion window allowed by the underlying Cubic ($cwnd_{max}$), which can be regarded as an instance of SPINE-X mentioned in §VII-E2 without a hierarchical policy structure.

IX. DISCUSSION

In this section, we give a detailed discussion about the comparison of deep reinforcement learning and other related methods including multi-armed bandit and layering as optimization decomposition.

Backup Mechanisms for DRL-based Solutions: Given the unpredictable nature of network environments, DRL-based Congestion Control (CC) algorithms might occasionally fail or underperform, necessitating the consideration of backup mechanisms. One possible approach is to introduce classic CC schemes as a backup when the DRL model underperforms. For example, Libra [48] implements a control cycle consisting of exploration, evaluation, and exploitation stages to quantify the performance of different CC algorithms and facilitate real-time policy switching. The system can fall back on a classic CC scheme such as Cubic when necessary, especially during unstable network conditions where feedback may be delayed or lost. Exploring these backup mechanisms can pave the way for more resilient and reliable DRL-based CC algorithms. We leave it for the future work.

Multi-armed bandit vs RL Another possible RL-based approach to solve CC is multi-armed bandit [49], where the agent learns to choose the action that maximizes the instantaneous reward in one step. Multi-armed bandit is one of the simplest reinforcement learning algorithms and has been applied in fuzzing [50], wireless network spectrum scheduling [51] and small cell activation in 5G networks [52]. However, as congestion control is a sequential decision making process where the actions (sending rate adjustments) enforced by the end-host have long-term effect on both the involved network elements and other competing flows' behaviors, we adopt RL rather than bandit to optimize the cumulative collected reward in the future of the flow's lifetime in this work.

Layering as optimization decomposition Many layered network architectures have been modeled as a generalized network utility maximization (NUM) problem in an integrated framework named "layering as optimization decomposition" [53], where the original optimization problem is decomposed into subproblems handled by both distributed computation elements and functional modules in different layers. We can formulate SPINE as a hybrid decomposition case of optimization decomposition: the NUM problem is decomposed not only horizontally across distributed end-hosts, but also vertically across the policy generator, the watcher and the CC executor. As the policy generator learns to control the hyperparameters of the CC executor (sub-policy) in a data-driven manner, SPINE can be regarded as an algorithm that adaptively optimizes various parameterized NUM subproblems implicated by the behavior of the CC executor to approximately optimize the RL objective function. An interesting future direction is to demonstrate how the parameterized subproblem of the underlying sub-policy affects/limits the approximation of the objective function theoretically.

X. CONCLUSION

In conclusion, we present SPINE, a DRL-based CC algorithm. With the help of the hierarchical policy structure, SPINE achieves ultra-high control frequency with ultra-low model inference frequency. Therefore, it achieves consistent high performance across various network conditions with low overhead comparable to classic CC scheme Cubic.

SPINE is far from being the end of the story, and there are still many open questions about learning-based congestion control. However, we believe SPINE has made a significant step forward towards a practical fully learning-based congestion control algorithm by providing a new DRL architecture and training paradigm. Also, the hierarchical policy architecture proposed in SPINE will shed light on the adoption of reinforcement learning in various networking and system applications requiring fine-grained control in the future.

REFERENCES

[1] V. Mnih et al., "Playing Atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.
[2] V. Mnih et al., "Asynchronous methods for deep reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937.

[3] D. Silver et al., "A general reinforcement learning algorithm that masters chess, Shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, Dec. 2018.

[4] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 197–210.

[5] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Interest Group Data Commun.* New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 270–288.

[6] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng, "ACC: Automatic ECN tuning for high-speed datacenter networks," in *Proc. ACM SIGCOMM Conf.* New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 384–397, doi: 10.1145/3452296.3472927.

[7] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi, "$LinnOS$: Predictability on unpredictable flash storage with a light neural network," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2020, pp. 173–190.

[8] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2019, pp. 3050–3059.

[9] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the Internet," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 632–647.

[10] Y. Ma et al., "Multi-objective congestion control," in *Proc. 17th Eur. Conf. Comput. Syst.*, Mar. 2022, pp. 218–235.

[11] J. Zhang, C. Zeng, H. Zhang, S. Hu, and K. Chen, "LiteFlow: Towards high-performance adaptive neural networks for kernel datapath," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 414–427.

[12] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting congestion control for consistent high performance," in *Proc. 12th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2015, pp. 395–408.v.

[13] M. Dong and T. Meng, "PCC Vivace: Online-learning congestion control," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Apr. 2018, pp. 343–356.

[14] A. Giessler, J. Hänle, A. König, and E. Pade, "Free buffer allocation—An investigation by simulation," *Comput. Netw.*, vol. 2, no. 3, pp. 191–208, Jul. 1978.

[15] L. Kleinrock, "On flow control in computer networks," in *Proc. Int. Conf. Commun.*, vol. 2, 1978, pp. 2–27.

[16] L. Kleinrock, "Power and deterministic rules of thumb for probabilistic problems in computer communications," in *Proc. Int. Conf. Commun. (ICC)*, vol. 3, 1979, pp. 1–43.

[17] J. Jaffe, "Flow control power is nondecentralizable," *IEEE Trans. Commun.*, vol. COM-29, no. 9, pp. 1301–1306, Sep. 1981.

[18] J. Chung, S. Ahn, and Y. Bengio, "Hierarchical multiscale recurrent neural networks," 2016, *arXiv:1609.01704*.

[19] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," 2015, *arXiv:1509.02971*.

[20] S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney, "Recurrent experience replay in distributed reinforcement learning," in *Proc. Int. Conf. Learn. Represent.*, 2018, pp. 13854–13865.

[21] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[22] A. Narayan et al., "Restructuring endpoint congestion control," in *Proc. Appl. Netw. Res. Workshop*, Jul. 2018, pp. 30–43.

[23] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov, "Linux netlink as an ip services protocol," Tech. Rep. RFC 3549, 2003.

[24] D. Engine Contributors. (2021). *DI-Engine: OpenDILab Decision Intelligence Engine*. [Online]. Available: https://github.com/opendilab/DI-engine

[25] F. Y. Yan et al., "Pantheon: The training ground for internet congestion-control research," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 731–743.

[26] R. Netravali et al., "Mahimahi: Accurate record-and-replay for HTTP," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2015, pp. 417–429.

[27] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008.

[28] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 20–53, Oct. 2016.

[29] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 329–342.

[30] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, *TCP Vegas: New Techniques for Congestion Detection and Avoidance*, no. 4. New York, NY, USA: ACM, 1994.

[31] K. Winstein, A. Sivaraman, and H. Balakrishnan, "Stochastic forecasts achieve high throughput and low delay over cellular networks," in *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 459–471.

[32] J. Jiang and Z. Lu, "Learning fairness in multi-agent systems," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 13854–13865.

[33] U. Siddique, P. Weng, and M. Zimmer, "Learning fair policies in multi-objective (deep) reinforcement learning with average and discounted rewards," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 8905–8915.

[34] M. Zimmer, C. Glanois, U. Siddique, and P. Weng, "Learning fair policies in decentralized cooperative multi-agent reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 12967–12978.

[35] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 18, no. 4, pp. 314–329, 1988.

[36] S. Floyd, T. Henderson, and A. Gurtov, "The newreno modification to TCP's fast recovery algorithm," Tech. Rep. RFC 2582, 1999.

[37] S. Liu, T. Başar, and R. Srikant, "TCP-illinois: A loss- and delay-based congestion control algorithm for high-speed networks," *Perform. Eval.*, vol. 65, nos. 6–7, pp. 417–440, Jun. 2008.

[38] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound TCP approach for high-speed and long distance networks," in *Proc. IEEE INFOCOM. 25TH IEEE Int. Conf. Comput. Commun.*, Apr. 2006, pp. 1–12.

[39] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg, "Adaptive congestion control for unpredictable cellular networks," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 509–522.

[40] S. Park, J. Lee, J. Kim, J. Lee, S. Ha, and K. Lee, "ExLL: An extremely low-latency congestion control for mobile cellular networks," in *Proc. 14th Int. Conf. Emerg. Netw. EXperiments Technol.*, Dec. 2018, pp. 307–319.

[41] R. Mittal et al., "TIMELY: RTT-based congestion control for the datacenter," in *Proc. ACM Conf. Special Interest Group Data Commun.* New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 537–550, doi: 10.1145/2785956.2787510.

[42] Y. Li et al., "HPCC: High precision congestion control," in *Proc. ACM Special Interest Group Data Commun.* New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 44–58, doi: 10.1145/3341302.3342085.

[43] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.* New York, NY, USA: Association for Computing Machinery, Aug. 2010, pp. 63–74.

[44] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for TCP in data-center networks," *IEEE/ACM Trans. Netw.*, vol. 21, no. 2, pp. 345–358, Apr. 2013.

[45] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Wanna make your TCP scheme great for cellular networks? Let machines do it for you!" *IEEE J. Sel. Areas Commun.*, vol. 39, no. 1, pp. 265–279, Jan. 2021.

[46] K. Winstein and H. Balakrishnan, "TCP ex machina: Computer-generated congestion control," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*. New York, NY, USA: Association for Computing Machinery, Aug. 2013, pp. 123–134.

[47] T. Meng, N. R. Schiff, P. B. Godfrey, and M. Schapira, "PCC proteus: Scavenger transport and beyond," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 615–631.

[48] Z. Du, J. Zheng, H. Yu, L. Kong, and G. Chen, "A unified congestion control framework for diverse application preferences and network conditions," in *Proc. 17th Int. Conf. Emerg. Netw. Exp. Technol.* New York, NY, USA: Association for Computing Machinery, Dec. 2021, pp. 282–296, doi: 10.1145/3485983.3494840.

[49] D. Bouneffouf, I. Rish, and C. Aggarwal, "Survey on applications of multi-armed and contextual bandits," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2020, pp. 1–8.

[50] T. Yue et al., "EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, Aug. 2020, pp. 2307–2324. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[51] F. Li, D. Yu, H. Yang, J. Yu, H. Karl, and X. Cheng, "Multi-armed-bandit-based spectrum scheduling algorithms in wireless networks: A survey," *IEEE Wireless Commun.*, vol. 27, no. 1, pp. 24–30, Feb. 2020.

[52] S. Maghsudi and E. Hossain, "Multi-armed bandits with application to 5G small cells," *IEEE Wireless Commun.*, vol. 23, no. 3, pp. 64–73, Jun. 2016.

[53] M. Chiang, S. H. Low, A. R. Calderbank, and J. C. Doyle, "Layering as optimization decomposition: A mathematical theory of network architectures," *Proc. IEEE*, vol. 95, no. 1, pp. 255–312, Jan. 2007.

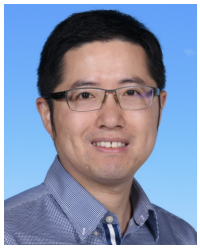
Han Tian received the B.Eng. degree in biology engineering from Beihang University, Beijing, China, in 2012, the M.Eng. degree in computer science from Sun Yat-sen University, China, and the Ph.D. degree from The Hong Kong University of Science and Technology, supervised by Prof. Kai Chen and Prof. Qiang Yang. He has authored and coauthored several refereed journals and proceedings including, IEEE S&P, CoNEXT, and EuroSys. His research interests include machine learning and its applications including networking, systems, and private computing.

Xudong Liao received the bachelor's degree from Wuhan University, Wuhan, China, in 2020. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology. His research interests include congestion control, distributed systems, and datacenter networking, with a particular focus on developing innovative solutions to improve the efficiency and performance of modern networked systems.

Chaoliang Zeng received the B.S. degree in computer science from the University of Science and Technology of China (USTC), China, in 2018, and the Ph.D. degree from the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology (HKUST), in 2023. His research interests include hardware-accelerated datacenter systems, datacenter networking, and machine learning systems.

Decang Sun received the master's degree from Northeastern University in 2020. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology. His research interests include datacenter networking and hardware acceleration.

Junxue Zhang received the B.S. and M.S. degrees from Southeast University, China, and the Ph.D. degree in computer science and engineering from the iSing Laboratory, The Hong Kong University of Science and Technology (HKUST), supervised by Prof. Kai Chen. He is currently a Research Assistant Professor with the Department of Computer Science and Engineering, HKUST. His research interests include data center networking, machine learning systems, and privacy-preserving computation. His research work has been published in many top conferences and journals, such as SIGCOMM, NSDI, CoNEXT, and IEEE/ACM TRANSACTIONS ON NETWORKING.



Kai Chen (Senior Member, IEEE) received the Ph.D. degree in computer science from Northwestern University, Evanston, IL, USA, in 2012. He is currently a Professor with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong. His research interests include data center networking, machine learning systems, and privacy-preserving computing.