LiteFlow: Toward High-Performance Adaptive Neural Networks for Kernel Datapath

Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen, *Senior Member, IEEE*

***Abstract*— Adaptive neural networks (NN) have been used to optimize OS kernel datapath functions because they can achieve superior performance under changing environments. However, how to deploy these NNs remains a challenge. One approach is to deploy these adaptive NNs in the userspace. However, such userspace deployments suffer from either high cross-space communication overhead or low responsiveness, significantly compromising the function performance. On the other hand, pure kernel-space deployments also incur a large performance degradation because the computation logic of model tuning algorithm is typically complex, interfering with the performance of normal datapath execution. This paper presents LiteFlow, a *hybrid* solution to build high-performance adaptive NNs for kernel datapath. At its core, LiteFlow decouples the control path of adaptive NNs into: 1) a kernel-space fast path for efficient model inference; and 2) a userspace slow path for effective model tuning. We have implemented LiteFlow with Linux kernel datapath and evaluated it with three popular datapath functions including congestion control, flow scheduling, and load balancing. Compared to prior works, LiteFlow achieves 44.4% better goodput for congestion control, and improves the completion time for long flows by 33.7% and 56.7% for flow scheduling and load balancing, respectively.**

***Index Terms*— Kernel datapath, adaptive neural network, deployment.**

I. Introduction

OS KERNEL datapath, a *data path* between higher-layer applications and lower-layer network hardware implemented in OS kernel, has provided a variety of important networking functions, including congestion control (CC), packet filtering, scheduling and queueing, *etc*. Recently, we have seen a rising trend of adopting adaptive neural networks (NNs) to optimize them because adaptive NNs can continuously learn and adapt to the varying network environments and outperform handcrafted optimization algorithms due to their superb fitting capabilities. The adaptive NNs can simultaneously perform model inference to give prediction results based on the input data and conduct model tuning to improve the inference accuracy by retraining the NNs through these collected data. So far, adaptive NNs have been used in CC [1], [2], packet forwarding & routing [3], scheduling [4], *etc*, to optimize the function performance, *e.g.*, to achieve better goodput for CC, or better flow completion time (FCT) for scheduling, *etc*. Taking CC for an example, Aurora [1], a 3-layer NN, can achieve 38.5% better latency than BBR [5] while quickly adapting to different network environments.

Despite being promising, current deployment mechanisms for adaptive NNs largely compromise the above advantages. One approach is to deploy the NNs in userspace [1], [2], [3], [4]. For example, Aurora uses TensorFlow [6] and GYM [7] to deploy the NN with pure userspace transport implementation: UDT [8]; MOCC extends Aurora's design and further uses CCP [9] to integrate the userspace-deployed NN with the kernel-space networking stack. Userspace deployment is easy with these existing mature tools but requires the NNs to communicate with the kernel-space networking datapath functions frequently. In this paper, we discover that no matter how we choose the communication interval, the cross-space communication hurts the datapath function performance, compromising the benefit brought by the adaptive NNs. For example, our experiments show that, with a large interval, *e.g.*, 100ms, the goodput of a single flow is 14.9% lower than a small interval, *e.g.*, 1ms, due to reduced responsiveness of the NN. In contrast, with a small interval, the throughput of the datapath degrades by 40.4% when handling many concurrent flows because of the non-negligible overhead (§II-B). Orca has also observed this problem, but it takes a two-level control design which mitigates but not completely solves the performance issue [10].

An alternative approach is to implement adaptive NNs directly in the kernel-space. Existing works have explored two directions, but they both suffer from performance issues. One direction is to implement complete adaptive NNs, including both model tuning and inference, in the kernel-space [11]. However, such implementation suffers from inevitable datapath function performance degradation due to 2 reasons: (1) The model tuning algorithms required by adaptive NNs consume significant computation resources, interfering with the processing logics of datapath functions. (2) While using advanced CPU instructions such as SIMD/FP instructions can achieve high precision, it introduces overhead [11], [12], [13],

Manuscript received 19 February 2023; revised 1 July 2023; accepted 4 July 2023; approved by IEEE/ACM Transactions on Networking Editor M. Caesar. Date of publication 17 July 2023; date of current version 16 February 2024. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2021B0101400001; in part by the Hong Kong Research Grants Council (RGC) Theme-based Research Scheme (TRS) under Grant T41-603/20-R, Grant GRF-16215119, and Grant GRF-16213621; and in part by the NSFC under Grant 62062005. *(Corresponding author: Kai Chen.)*

Junxue Zhang is with the Intelligent System and Networking Laboratory (iSING Lab), The Hong Kong University of Science and Technology, Hong Kong, and also with Clustar Technology Company Ltd., Shenzhen 518000, China (e-mail: jzhangcs@connect.ust.hk).

Chaoliang Zeng and Kai Chen are with the Intelligent System and Networking Laboratory (iSING Lab), The Hong Kong University of Science and Technology, Hong Kong (e-mail: czengaf@connect.ust.hk; kaichen@cse.ust.hk).

Hong Zhang is with the David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada (e-mail: hongzhangblaze@gmail.com).

Shuihai Hu was with Clustar Technology Company Ltd., Shenzhen 518000, China. He is now with Huawei, Beijing 100015, China (e-mail: hushuihai1@huawei.com).

Digital Object Identifier 10.1109/TNET.2023.3293152

1558-2566 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

further deducting the datapath function performance. In our experiment, we observe that adaptive NNs in the kernel-space degrade the network throughput by up to 90% even with mini-batch. The other direction is to abandon the model tuning and convert the NNs into lightweight NNs for inference only, *e.g.*, the NN is optimized via integer quantization [14], [15], [16], [17] or converted into a decision tree [18]. However, these lightweight NNs lack an important property—learn and adapt to the environmental dynamics. Our experiment results show that such a lack of adaptation capability leads to $> 30\%$ performance degradation.

In this paper, we ask: *Can we design high-performance adaptive NNs for kernel datapath to optimize function performance?* To answer the question, we realize that the crux lies in the fact that existing adaptive NNs perform model inference and model tuning as a whole [1], [2], [3], [4]. However, model inference requires fast execution, which is better suited in the kernel-space; whereas model tuning requires high-precision and intensive calculation, which is better suited in the userspace.

Based on this observation, we present LiteFlow, a hybrid solution that decouples the control path of adaptive NNs into a kernel-space fast path for model inference and a userspace slow path for model tuning, so that both model inference and model tuning can be executed in the right place respectively. We note that the idea of decoupling has been briefly mentioned in a recent work KMLib [11]. However, it does not identify the challenges behind such decoupling, nor does it provide any design or implementation to realize the idea. In contrast, by LiteFlow, this paper takes the first initiative to fully explore the challenges and present comprehensive design and implementation to address these challenges.

Specifically, we identify the following four challenges: (1) The decoupling method requires two NNs of different design targets: one is compatible with kernel-space development environment (*e.g.*, integer only, implemented in C) and can be efficiently executed in kernel-space, the other is compatible with userspace machine learning frameworks, *e.g.*, TensorFlow, *etc* (*e.g.*, use `FP32`, implemented in Python). Thus, it requires non-trivial development and debugging efforts. (2) As only the userspace-deployed NN is further tuned, the kernel-space-deployed NN cannot timely react to the changing network environment, affecting the function performance. (3) Tuning the userspace-deployed NN needs data from kernel-space, leading to performance degradation caused by frequent cross-space communication. (4) A userspace-trained NN may be over-complicated, causing a large inference overhead, thus degrading the datapath performance.

To solve these challenges, we design LiteFlow to (1) *automatically* optimize the NN via high-precision integer quantization (*i.e.*, performing integer quantization that can preserve high precision) and leverages code transformation technology (*i.e.*, translating code written in one programming language into another) to generate a kernel-space-compatible snapshot (it is named as snapshot because it will not be further tuned); (2) *conservatively* update the snapshot with the userspace-tuned NN (by considering both necessity and correctness, see §III-C for details) to make sure that it is accurate under the changing environments; (3) perform online adaption with *batched* data based on the observation that network characteristics do not change at sub-second scale [19] to achieve high accuracy and low overhead simultaneously; (4) *accurately* conduct model optimization to guarantee the datapath performance.

We implement LiteFlow by realizing both the userspace and kernel-space designs described above. In userspace, LiteFlow offers standard interfaces for users to provide their own customized implementation of online adaptation functions. Thus, it can be integrated with any learning frameworks and utilities, *e.g.*, TensorFlow [6], PyTorch [20], MXNet [21], GYM [7], *etc*. In the kernel-space, LiteFlow is implemented with Linux kernel v4.15.0 and follows the modularization principle to separate the whole function into different kernel modules. Therefore, it is general and can support different NNs for different datapath functions.

To showcase LiteFlow can enable high-performance adaptive NNs in kernel datapath, we use it to optimize 3 popular kernel datapath functions with 4 different NNs. For CC, we evaluate LiteFlow with Aurora [1] and MOCC [2]. Experiment results show that for flow goodput, LiteFlow with these NNs can outperform userspace-deployed NNs by up to 44.4% while suffering no more overhead than kernel-space CC algorithms such as BBR and CUBIC. For flow scheduling, we evaluate LiteFlow with FFNN [19]. Experiment results show that LiteFlow with FFNN can outperform userspace-deployed FFNN by 33.7% for long flows. For load balancing [22], we design an MLP model and use LiteFlow to enable it in kernel datapath. Compared to userspace-deployed MLP, LiteFlow with MLP can achieve 56.7% lower FCT for long flows.

On a more general note, we have seen an emerging trend to move the networking functionalities into the userspace [8], [23] or offload them to the hardware, *e.g.*, SmartNICs [24], [25]. Nevertheless, there is still a large body of works [1], [9], [26], [27], [28] remain on the kernel network datapath which LiteFlow can directly apply to improve their performance. Meanwhile, for SmartNIC-offloaded adaptive NNs, we believe that LiteFlow's idea, albeit not directly applicable, can provide certain insight. For example, they can offload a snapshot NN on the hardware for efficient inference while leaving the model tuning part in the software stack for easy and flexible implementations.

This work does not raise any ethical issues.

II. Background & Motivation

A. Adaptive Neural Networks for Kernel Datapath Functions

Unlike traditional NN deployments which separate training from inference, adaptive NNs combine them as a whole and can continuously learn and adapt to the varying network environment while delivering superb performance. Therefore, there has been an increasing trend to adopt them to optimize datapath function performance. Applications of adaptive NNs for networking datapath include congestion control [1], [2], flow scheduling [4], network routing/forwarding [3], *etc*.

Unlike traditional optimization solutions, which heavily rely on operators' expertise to achieve ideal performance, NNs use a data-driven approach to automatically and continuously learn the optimization strategies without any human involvement, which can quickly adapt to the dynamics. Furthermore, due to the tremendous non-linear fitting capabilities of NNs, they have achieved better performance than hand-crafted ones.

Despite being promising, how to deploy these NNs remains a challenge. One approach is to deploy the NNs in the userspace, *e.g.*, with TensorFlow [6], PyTorch [20], GYM [7],

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

etc. When the kernel datapath function needs to make some updates, *e.g.*, changing the flow sending rate in response to network congestion, it will send the required input data to the userspace-deployed NNs to calculate the corresponding inference results. These results will then be sent back to the kernel-space for execution.

While the aforementioned userspace deployment simplifies the implementation, it introduces a performance penalty. In order to quickly react to network variations, the userspace-deployed NN needs to perform frequent communication with the corresponding kernel-space datapath functions. However, frequent communication between the two spaces will consume a lot of CPU resources, reducing the amount of CPU resources that can be allocated to the kernel datapath functions for executing their packet/traffic processing logic. As a result, existing adaptive NNs cannot deliver high performance when supporting a large number of concurrent networking processing pipelines.

B. Performance Penalty

In this section, we perform testbed experiments to demonstrate the performance degradation mentioned above. We choose congestion control (CC), one of the most important network functions, as the evaluated function. The algorithm we choose is Aurora [1]. We deploy Aurora with TensorFlow [6] and GYM [7] to enable online adaptation.[1] To interact with the kernel-space CC function, we further use CCP [9] to invoke the userspace-deployed Aurora model (CCP-Aurora). CCP-Aurora requires cross-space communication to perform congestion control. We use TCP BBR [5], a traditional CC algorithm completely implemented in the kernel-space, as the baseline. We build a testbed with 2 servers connected to a Mellanox SN2100 [29] switch via 100Gbps Ethernet links. We set the RTT to be 10ms via `netem` [30] to match the design of these NN-based algorithms. Each server is equipped with one 4-core 2.60GHz CPU and is installed with Ubuntu (kernel version 4.15.0). Note that one single TCP flow can reach around 1.6Gbps maximum throughput with our testbed settings.

Fine-grained cross-space communication is necessary to achieve good performance: In this experiment, we launch one single flow controlled by CCP-Aurora. We also set the receiver link to be 1Gbps (via switch configuration) and generate background UDP traffic (constant rate at 0.1Gbps) to emulate network congestion. The buffer size is 150KB. The network characteristics are stable and match the training environment of Aurora. We consider three communication intervals, *i.e.*, 100ms, 10ms, and 1ms. For each communication interval, we run the experiment for 10 seconds and measure the average goodput of the flow every 0.1 seconds. The CDF of the results is shown in Figure 1a.

As we can see, the achieved goodput decreases from 672.08Mbps to 585.17Mbps on average when we increase the communication interval. With a large communication interval, the datapath function cannot quickly reduce the sending rate at the sender side when congestion arises in the network. As a result, severe packet loss occurs in the network, degrading the goodput of the flow. In Figure 1b, we further measure the queue length of the bottleneck link. We observe that the queue length is small and stable when setting a small communication interval (*e.g.*, 1ms). But when we increase the interval, the queue length increases and oscillates significantly. The results indicate that a fine-grained communication interval is necessary to make the CCP-Aurora responsive enough to achieve good network performance.

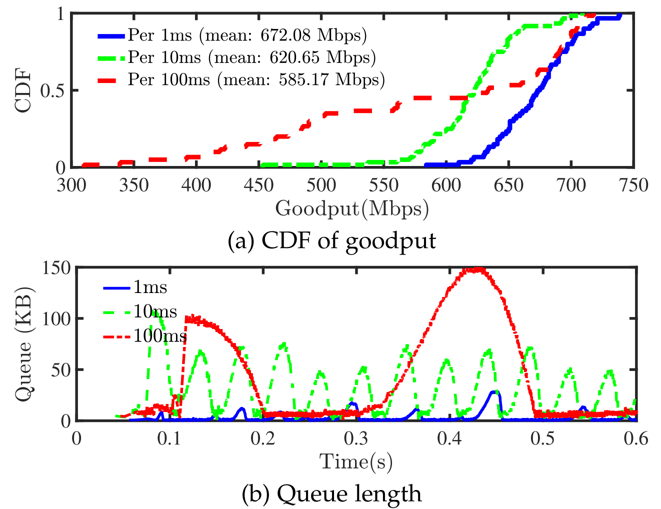


Fig. 1. Fine-grained cross-space communication is necessary to achieve good network performance.

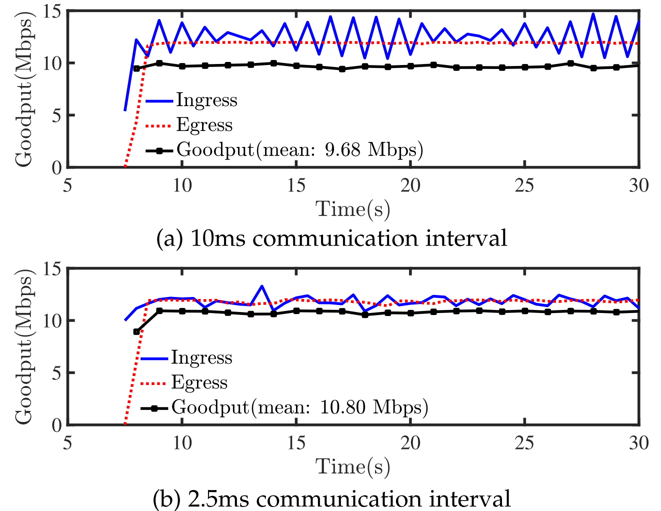


Fig. 2. Toy example of how flow reacts to congestion when controlled using Aurora with different intervals.

To help readers better understand the phenomenon, we will use experiments to visualize it. As we cannot observe the ingress/egress throughput of the bottleneck queue at a very fine-grained view on our testbed, we conduct contrived toy experiments to visualize the problem. We deploy Aurora with UDT [8] and configure UDT to communicate with the Aurora model with an interval. The bottleneck link is an emulated link using Mahimahi [31], with bandwidth and one-way RTT set as 12Mbps and 10ms, respectively. In our emulated experiment, only a single flow is launched and we visualize the ingress/egress speed of the bottleneck link when the flow is controlled by different intervals, *e.g.*, 10ms and 2.5ms.

The results are shown in Figure 2. We observe that, with a 10ms communication interval, the sending rate of a flow cannot even converge to the available bandwidth of the bottleneck link under such a simple experiment setting. Consequently, the flow suffers from degraded goodput. On the contrary, a 2.5ms communication interval mitigates the problem and achieves better flow goodput. The toy experiment further confirms

[1]We use code from Aurora's official code repository: https://github.com/PCCproject/PCC-RL/tree/master/src/gym/online

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

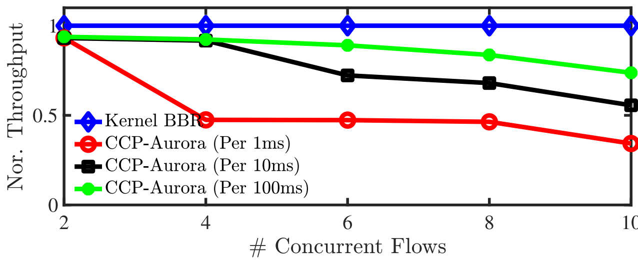


Fig. 3. Fine-grained cross-space communication suffers high overhead.

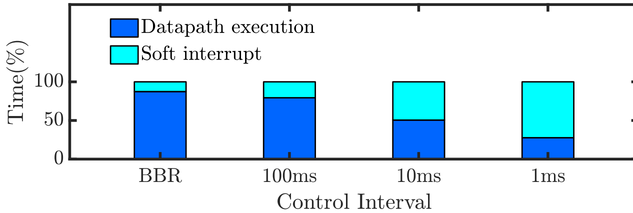


Fig. 4. Software interrupt (softirq) caused by frequent communication leads to the increasing overhead.

that for NN-based CC algorithms, such as Aurora, a fine-grained communication interval is necessary even in very simple network environments.

Fine-grained cross-space communication suffers from high overhead: In this experiment, we launch N flows controlled by CCP-Aurora (N = 2, 4, 6, 8, 10). Similarly, we vary the communication interval from 100ms to 1ms. As a baseline, we also run this experiment with BBR, which does not have any overhead of cross-space communication. For each communication interval, Figure 3 shows the normalized aggregated throughput of N flows (normalized by the aggregated throughput of BBR) as N increases. Here we use normalized values to highlight the performance degradation caused by the cross-space communication.

We mainly have two observations. First, when the number of flows increases, the normalized aggregated throughput of CCP-Aurora decreases. Second, smaller communication interval leads to worse performance. When there are 10 concurrent flows, the normalized aggregated throughput with a 1ms interval is 5.5Gbps, which is less than half of the baseline of 16.1Gbps.

The reason is that cross-space communication has significant CPU overhead. When the kernel needs to invoke a userspace program (*i.e.*, to request the CCP-Aurora for inference results), it generates a software interrupt to switch the execution flow from the kernel-space to userspace, leading to extra overhead in handling interruptions. When there are multiple concurrent flows, the remaining CPU resource is not sufficient to fully support the kernel processing pipelines of a datapath function.

To confirm this, we use `mpstat` [32] to measure the CPU usage when there are 10 active flows. We measure the CPU usage under CCP-Aurora with different communication intervals. We also measure the CPU usage of kernel BBR as our baseline. Figure 4 shows the portion of CPU cycles spent on different tasks. As we can see, with BBR, the software interrupts only take 15.4ms (mainly for handling packet receiving logic), occupying only ~12.6% of the total CPU execution time. In contrast, with CCP-Aurora, when we decrease the communication interval from 100ms to 1ms, the time spent on handling software interrupts dramatically increases from 30.8ms to 133.9ms. The portion of time handling software interrupts over total execution time increases from 20.6% to 72.3%. It is worthwhile to note that the software interrupts are mainly caused by cross-space switching rather than Aurora's userspace execution. The result indicates that a fine-grained communication interval will consume a significant amount of CPU resources, leaving limited CPU resources for executing normal packet processing logics for datapath functions. As a result, userspace NN deployment suffers from high overhead when supporting many concurrent flows. In some cases where the CPU is not the bottleneck, *e.g.*, the available bandwidth is the bottleneck for Internet settings, we will not see considerable performance degradation. However, we believe they still suffer the cross-space communication overhead to some extent.

Conclusion: When pursuing high-performance datapath functions, the userspace deployment of NNs has an inherent problem of suffering from either high overhead or low responsiveness. No matter how we set the communication interval, we encounter an inevitable performance penalty.

C. What About Adaptive Neural Networks Direct in Kernel-Space Datapath?

To eliminate the performance penalty caused by the cross-space communication, one may deploy the adaptive NNs in the kernel-space. There are two existing directions but they both suffer from function performance degradation.

One approach is to directly implement the adaptive NNs, both the NN optimization and inference, in the kernel-space. This approach introduces dramatic NN development and debugging difficulties since we have to use system programming languages such as C and suffer from various constraints, *e.g.*, limited library support, *etc*, in the kernel-space. Although there are some research works, such as KMLib [11], targeting at lowering the development difficulties for NNs in the kernel-space, they still suffer from inevitable performance degradation issues. To realize NN adaptation, we have to implement the model optimization algorithms, such as Stochastic Gradient Descent (SGD) [33], ADAM [34], *etc*. As these algorithms require over-complicated computations, *e.g.*, gradient calculation, directly implementing adaptive NNs in the kernel-space degrades the performance. Furthermore, in an integer-only development environment, implementing these algorithms either suffers accuracy loss (*e.g.*, approximation using lookup table) or enlarged overhead of using SIMD/FP instructions [11]. With the same testbed mentioned in §II-B, we have implemented an SGD optimizer in the kernel-space to optimize a hand-crafted C version of Aurora. Our experiment results show that the throughput drops by up to 90% even with batched data.

Another approach is to abandon the NN optimization and convert the NNs to one-time lightweight NNs for inference only. These lightweight NNs are potential to be executed efficiently in the kernel-space since we can perform integer quantization [14], [15], [16], [17] to avoid using SIMD/FP instructions or transform the NNs into C/C++-based decision trees [18] which are compatible with the kernel-space. However, these lightweight NNs lose an important property — learning and adapting to the dynamics. Our experiments show that without such property, the datapath functions suffer from dramatic performance loss. In our experiment, we hand-craft a lightweight Aurora model optimized via integer quantization and deploy it in the kernel-space. Initially, we tune the pattern of background traffic as that when we train the Aurora

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

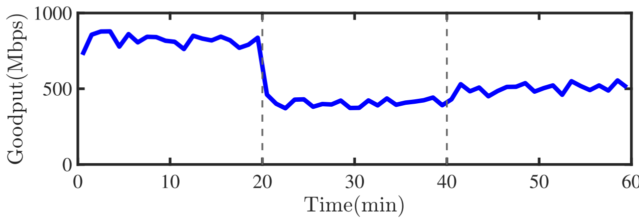


Fig. 5. Lack of online adaptation leads to performance degradation with traffic dynamics.

model, then we randomly change the traffic pattern every 20 minutes. We measure the goodput of a single flow and the results are shown in Figure 5. From the experiments, we observe that when the training environment fits the testbed environment, Aurora can reach ideal performance. Yet when the environment dynamically changes, the performance of Aurora degrades because it cannot learn and adapt to the new environment.

Conclusion: Directly deploying adaptive NNs in the kernel-space leads to suboptimal performance because of either large implementation complexity/overhead or lack of adaptation towards the changing environment.

III. LiteFlow

Given the current adaptive NN deployment solutions, either in userspace or in kernel-space, have an inherent problem of function performance loss, we ask: *Can we build high-performance adaptive neural networks for kernel datapath to optimize datapath functions?* Our answer is LiteFlow, a *hybrid* solution to build high-performance adaptive NNs for kernel datapath.

Instead of adopting one control path for both model inference and model optimization as the state-of-the-art adaptive NNs do, LiteFlow decouples model inference from model optimization into two paths so that each of them can be executed in the *right* place. Specifically, LiteFlow builds a kernel-space path, *i.e.*, fast path, for model inference, and a path from kernel-space to userspace, *i.e.*, slow path, for model tuning.

Why LiteFlow Works?: As discussed above, LiteFlow puts the NN inference in the fast kernel-space path while leaving NN optimization in the slow path. Our insight is that (1) NNs for kernel datapath functions require very frequent inference to be responsive. Thus, putting them in the kernel-space can reduce the overhead. (2) Model tuning, *i.e.*, online adaptation, is suitable in userspace because it can benefit from easy-to-use APIs, advanced features (such as floating point/multi-thread support) brought by mature software and libraries in the userspace.

Design Challenges: Although the decoupling idea has been briefly mentioned in KMLib [11], the challenges behind the idea are left unexplored. To the best of our knowledge, we are among the first to identify these challenges and provide comprehensive designs and implementations to address these challenges by proposing LiteFlow.

Specifically, we identify the following four challenges:

C1. The decoupling method requires two NNs of different design targets: one is compatible with the kernel-space environment and can be efficiently executed there, while the other one is compatible with userspace machine learning frameworks. Thus, users have to develop two NNs, and suffer from restrictions in kernel-space as discussed in §II-C, which introduces non-trivial development and debugging efforts.

C2. Since only the userspace-deployed NN is further tuned, the snapshot in the kernel-space cannot timely react to the changing network environment, degrading the function performance.

C3. Tuning the userspace-deployed NN requires continuously delivering data from kernel-space to userspace. Such frequent data exchange causes massive cross-space communication, yielding a large overhead.

C4. The kernel-space is performance-critical. An arbitrary userspace-trained NN may cause a large inference overhead if the NN is over-complicated, further deducting the datapath performance.

To address these challenges, we design LiteFlow to:

- *automatically* optimize the NN via high-precision integer quantization and leverages code transformation technology to generate a kernel-space-compatible snapshot.
- *conservatively* update the snapshot with the userspace-tuned NN to make it accurate under the changing environment. Specifically, LiteFlow considers both correctness: LiteFlow waits for the online adaption to converge, and necessity: LiteFlow minimizes the number of snapshot updates to avoid the interference of function performance caused by kernel-space locks.
- *batchwisely* perform online adaption based on the network characteristics to simultaneously achieve high accuracy and low overhead.
- *accurately* perform model optimization to reduce inference overhead while keeping high model fidelity.

Existing related works [11], [14], [16], [17], [18], [35], [36], [37], [38], [39], [40] that fail to simultaneously solve the four challenges do not lead to a practical and deployable solution (see more discussions in §VIII).

Architecture & Workflow: Figure 6 shows the architecture of LiteFlow. LiteFlow is a hybrid framework, which consists of both userspace and kernel-space components. LiteFlow also provides APIs for users to implement their customized model tuning algorithms (more details in §IV).

The workflow of LiteFlow is as follows: Given a userspace-designed and trained NN, LiteFlow first generates

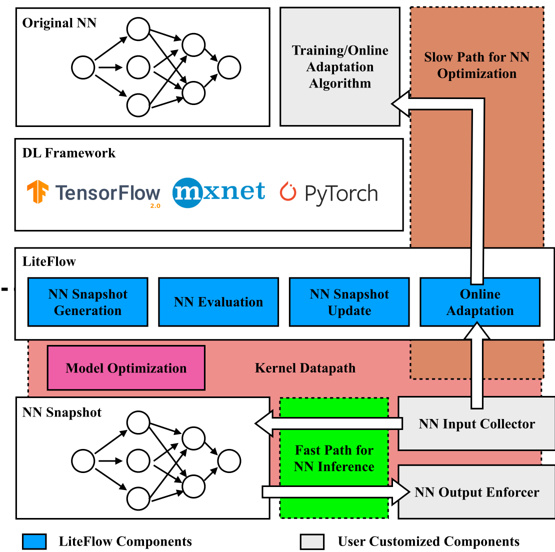


Fig. 6. LiteFlow Architecture.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

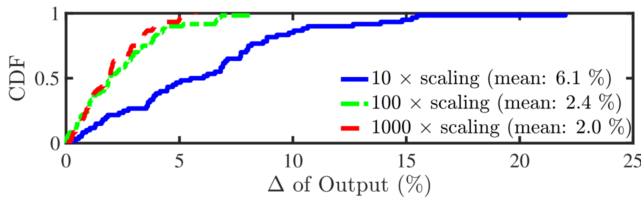


Fig. 7. LiteFlow's integer quantization does not lose much accuracy by adding scaling layers.

the NN snapshot, which will be deployed in the kernel-space fast path for inference (§III-A). Meanwhile, LiteFlow also collects the input and output data of the snapshot to further tune the userspace-deployed NN in the slow path in a batch mode (§III-B). After every training batch, LiteFlow evaluates whether it needs to update the snapshot from both correctness and necessity aspects (§III-C and §III-D). It is worthwhile to note that LiteFlow does not explicitly evaluate the performance of a NN but relies on the common wisdom that online adaptation is likely to lead to a better NN after it converges. Moreover, if the NN to be deployed is over-complicated, we will perform model optimization before deploying the NN (§III-E). We will show how LiteFlow works in detail in the following sections.

A. NN Snapshot Generation

In order not to compromise the performance gain brought by adaptive NNs, LiteFlow has to generate accurate and efficient kernel-compatible snapshots. To achieve it, LiteFlow first performs high-precision integer quantization to (1) avoid the overhead of using SIMD/FP instructions and (2) keep high accuracy. Second, to make the snapshot kernel-compatible, LiteFlow leverages code translation technics to transform the userspace NN into a kernel module.

High-precision Integer Quantization: Directly performing vanilla integer quantization [16], [17] causes dramatic accuracy loss in kernel-space. For example, we have a NN for CC, and its output is the portion α of the line rate as target sending rate, thus, $\alpha \in [0, 1]$. After we perform integer quantization, $\alpha \in \{0, 1\}$, which causes the target sending rate to be either 0 or line rate, leading to dramatic performance degradation. In order to prevent such accuracy loss, LiteFlow performs an input/output scaling before quantization. In the above case, we will add a scale-up layer after the original output layer, thus, the output becomes $\alpha' = \alpha \times C$, where C denotes the scaling factor and is usually a large integer, *e.g.*, 1000. As a result, $\alpha' \in \{0, 1, \ldots, 1000\}$, thus the sending rate is $\lfloor \frac{\alpha' \times \text{line rate}}{C} \rfloor$, which does not lose much accuracy. Figure 7 shows the statistics of accuracy loss caused by LiteFlow's quantization towards different NNs. We observe that by using proper scaling, *e.g.*, 1000 × scaling, LiteFlow's quantization loses 2% accuracy on average.

Automatic Layer-wise Code Translation: The idea is based on the observation that NNs are usually composed of repetitive and enumerative building blocks – layers. Therefore, LiteFlow can maintain kernel-space implementations of each type of layer, *i.e.*, layer template. Listing 1 shows the template of kernel-space implementation of a fully connected layer. The template contains only computation logic but leaves all data as placeholders.

LiteFlow further scans the quantized NN to extract the parameters and synthesize them with a certain template.

```
static void fc_{{ prefix }}_comp (s64 *input, s64 *output)
{
  {% for i in range(0, output_size) %}
  output[{{ i }}] =
      {%- for j in range(0, input_size) -%}
         (input[{{ j }}] + {{ input_offset }}) * ({{
             weights[i][j] }} + {{ weight_offset }})
         {%- if not loop.last %} + {% endif -%}
      {%- endfor %} + ({{ bias[i] }});
   ...
   {% endfor %}
}
```

Listing 1. Template of fully connected layers (We use Python Jinja [41] as the template engine).

```
static void fc_5_comp (s64 *input, s64 *output)
{
  output[0] = (intput[0] + 0) * 55 + ... + (intput[15] +
       0) * (-16 + 0) + (-1180);
  ...
}
```

Listing 2. Synthesized kernel-space implementation of a particular fully connected layer.

Listing 2 shows the synthesized kernel-space implementation of a fully connected layer. Next, LiteFlow combines the implementations of all layers into a complete source code file. Eventually, LiteFlow invokes `GCC` [42] to compile the code into a kernel module, which can be installed in the kernel-space. However, some layers are difficult to be converted into kernel-space compatible and optimized ones. For example, these layers use functions that are not supported in the kernel-space, such as `tanh` activation function. For these layers, LiteFlow uses lookup table to approximate these functions with high precision and low computation complexity. Compared to function-based approximation methods, *i.e.* using Taylor Series [43] to convert the function into a polynomial, LiteFlow's design of lookup table has the following two advantages: (1) it can ensure persistent high-precision approximation while function-based approximation methods are accurate only within a certain range, (2) it has a constant computation complexity, which is preferred in the kernel datapath. In contrast, function-based approximation solution has an increasing computation complexity when using higher-degree Taylor Series for higher accuracy.

B. Online Adaptation

To learn and adapt to the network dynamics, LiteFlow further enables online adaptation for the NN in the slow path. To achieve it, LiteFlow has to deliver the training data from kernel-space to userspace, *e.g.*, congestion signals, flow status, *etc*. If we perform such data exchange once we receive new data (receive a new packet), similar to the problem in §II-B, the cross-space communication compromises the performance gain achieved by adaptive NNs.

To design a low-overhead online adaptation mechanism, we observe that for datapath functions, the environment characteristics, such as traffic patterns and flow size distributions, *etc*, usually do not change at sub-second timescales [19], [44]. Thus, NN tuning in a batch mode is sufficient. Based on this observation, LiteFlow accumulates the training data in the kernel-space as a batch and delivers the batched data to userspace for tuning the NN in the slow path every T time. Moreover, the batch data delivery interval decides the performance of LiteFlow. A small interval leads to dramatic overhead caused by frequent cross-space communication (similar to the problem discussed in §II-B) while a large interval

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

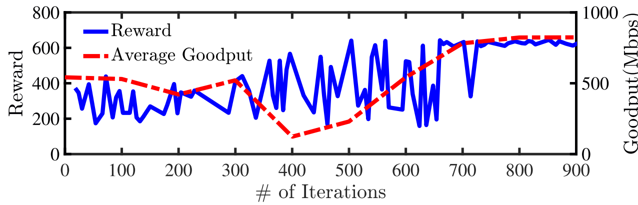


Fig 8. Before Aurora's online adaptation finishes exploration, the performance of Aurora is unstable and sub-optimal.

degrades NN's ability to learn the environmental changes. Micro-benchmark experiment results in §V-A recommend to set the interval T between 100ms and 1000ms. In our implementation, we set $T = 100$ms.

Is the Batched-mode Suitable for Optimizing Datapath Functions? Although the batch-mode used by LiteFlow cannot continuously tune the NN, we show through experiments that NN tuning with batched data is more suitable for kernel datapath functions because online adaptation does not continuously lead to better functional performance. As a result, we do not need to update the NN snapshot in an online manner but should design a mechanism to update the NN at the right time (§III-C). Figure 8 shows the online adaptation progress of Aurora and we can observe that Aurora takes $\sim$ 800 iterations (receiving around 800 packets) to perform thorough explorations. We also generate different snapshot NNs every 100 iteration and measure the average goodput if we deploy these snapshots in the fast path. The goodput is shown on the right Y-axis. From the experiment results, we find that only when the online adaptation finishes the exploration, the NN in datapath can achieve ideal performance.

Worth mentioning, NN tuning via reinforcement learning is usually performed with a simulator in userspace, Thus, in this case, both online and batched modes should yield identical training efficiency because we can emulate the online mode by feeding the batched data sequentially into the simulator.

C. NN Synchronization Evaluation

After model tuning, LiteFlow evaluates whether it should update the NN snapshot with the tuned NN from correctness and necessity perspectives.

Correctness: As discussed in §III-B, the online adaptation takes time to converge to optimal performance. Thus, we will perform NN synchronization until the online adaptation finishes exploration to deploy the correct snapshot in the datapath, *i.e.*, the one with optimal performance instead of an unstable one. LiteFlow achieves it by continuously observing user-defined metrics, *e.g.*, training loss value, to determine if the exploration converges. Note that LiteFlow users can flexibly choose their metrics in LiteFlow (more details in §IV) and we use training loss as the metric as it works well in our implementation.

Necessity: Since updating the NN snapshot influences the datapath function performance (more details in §III-D), we should conservatively update the snapshot only when it's necessary. To evaluate such necessity, we introduce the definition of fidelity loss. Let f denotes the NN in the userspace, and f' denotes the NN snapshot. Given the input data $\mathbf{x}$. We have fidelity loss $L(\mathbf{x})$ defined as:

$$L(\mathbf{x}) = |f'(\mathbf{x}) - f(\mathbf{x})| \quad (1)$$

Similar to previous work [45], our fidelity loss evaluation is based on the assumption that data characteristics in the past (used for training) are similar to the upcoming ones (used for inference). Furthermore, as discussed in §III-B, LiteFlow delivers training data to the userspace in a batch mode to reduce the cross-space communication overhead. We will calculate the fidelity loss over every $\mathbf{x} \in \mathbf{X}$ to obtain the minimal $L(\mathbf{x})$, where $\mathbf{X}$ denotes the set of all data in one batch. It is necessary to update the NN snapshot only when the minimal fidelity loss exceeds a user-defined threshold, *i.e.*, the difference between the two NNs is large enough. Here, we use the minimal fidelity loss as the necessity metric to make the NN snapshot synchronization as conservative as possible to minimize the performance interference caused by snapshot updates. For the threshold, we set it as $\alpha \times (O_{max} - O_{min})$, where O_{max} is the maximum output value of the NN and O_{min} is the minimum. For example, in Aurora, O_{max} is 1 and O_{min} is 0. For α, we empirically set it to be %5, which delivers good performance in our experiments.

D. NN Snapshot Update

As discussed above, deploying a NN snapshot in the kernel-space causes potential performance interference for datapath functions. The key reason is the existence of locks, which causes dramatic waiting time if the locking mechanism is not properly designed.

Specifically, when updating the NN snapshot, LiteFlow has to leverage a kernel lock (usually a spin lock) to temporally prevent the NN snapshot from being used by other functions' control flows. The direct approach follows 3 steps: (1) acquiring a spin lock; (2) deploying a new NN snapshot in the kernel-space; (3) releasing the lock. Such an approach has a critical performance issue: the locking time is significant because NN update requires transferring large data, *e.g.*, model parameters, from userspace to kernel-space. Consequently, functions relying on the NN will wait for the lock, eventually causing performance issues, *e.g.*, TCP timeout.

To solve the problem, LiteFlow adopts an active-standby-switch approach. Basically, LiteFlow maintains one NN snapshot as active, another snapshot as standby. Only the active snapshot is used for inference. Moreover, LiteFlow designs an inference router to switch the role of the two snapshots by forwarding the inference request to different snapshots. The workflow is shown in Figure 9. First, as Figure 9a shows, the inference request is sent to the inference router and then forwarded to the active snapshot. When LiteFlow updates the snapshot, as shown in Figure 9b, it generates and deploys a new snapshot as the standby one instead of directly replacing the active one. Although this process may take significant time, the datapath function can still use the active snapshot for inference, and no lock is acquired. Finally, as Figure 9c demonstrates, after the standby snapshot is deployed, the inference router can change the role of the two snapshots, making the standby snapshot as the active one and forwarding the inference requests to it. During this process, only a small part of the code in the inference router acquires a lock (3 lines of code to change a pointer in LiteFlow's implementation), causing a delay of only several nanoseconds.

Flow Consistency: While promising, the active-standby-switch approach may cause flow inconsistency potentially. Flow inconsistency refers to a problem that some packets of one flow are served by an old NN snapshot while others are served by a new snapshot. Under this circumstance, flows

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

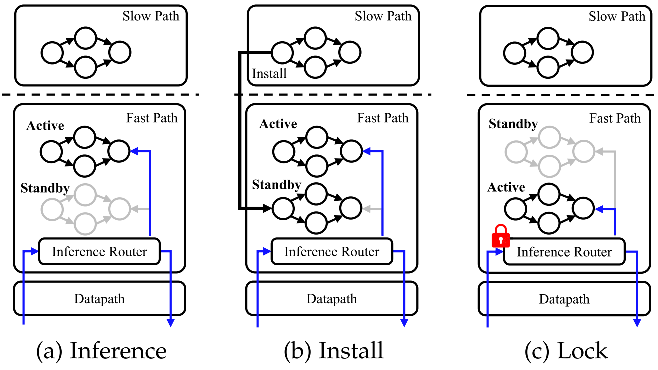


Fig 9. LiteFlow adopts active-standby-switch approach to mitigate the performance impact of locks.

will suffer from performance vibration. Taking CC as an example, a sudden change of flow sending rate may cause queue overflow, leading to performance degradation. Thus, to prevent flow inconsistency, we design a flow cache in the inference router to ensure that packets of one flow will use the same NN for inference.[2]

When a NN inference request comes, LiteFlow calculates its flow ID and uses the flow ID as the key to look up the flow cache. The flow cache is a kernel-space hash table with flow ID as the key and pointer to a NN as the value. If there is a cache hit, then we directly use the pointer to find the NN for inference. If there is a cache miss, then we go back to the inference router to use the active NN for inference. Meanwhile, we move the pointer to the active NN to the cache for future use. When a flow finishes, *e.g.*, by receiving TCP FIN packet, we remove it from the cache. We also set up a timeout mechanism to remove inactive records. Each NN maintains a reference count starting from 0. The counter increases when we cache a NN's pointer and decreases up on removal of a pointer. A NN module can be removed from the kernel-space only when the reference count reaches 0.

E. Model Optimization

After introducing the core idea of LiteFlow, we now demonstrate how to make an arbitrary NN suitable for LiteFlow. Since a huge NN, *e.g.*, NN with thousands of layers, involves a large inference overhead, thus inevitably degrading the datapath efficiency. To make these huge NNs suitable for kernel datapath, we have to perform model optimization to accurately reduce the inference overhead while still keeping high NN fidelity. To achieve this goal, we will first set up a cost model to establish a quantitive relationship between NN complexity and datapath performance to guide our further optimization. Then we will propose our structure pruning algorithm to reduce the complexity of a NN to guarantee specific datapath performance.

1) Cost Model: Similar to LiteFlow's layer-wise code translation technology, we again utilize the layer structure of a NN to evaluate the inference cost of a NN. Specifically, the cost of the whole model is calculated by summing up all costs of layers:$\mathscr{C} = \Sigma_{l \in L}\mathscr{C}_l$ and $\mathscr{C}_l$ denotes the cost of a particular layer l. Similar to code template mentioned before, we provide a cost function for each type of layer. For example, Equation 2

[2]LiteFlow users can easily disable flow cache for a particular datapath function if it does not require flow consistency.

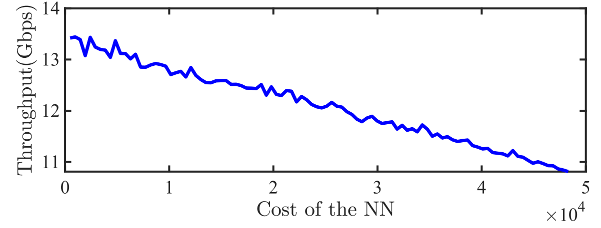


Fig 10. Linear relationship between throughput and computation cost.

calculates the cost of a convolution layer.

$$\mathscr{C}_l = \lceil \frac{W - W_{kernel} + 1}{p} \rceil * \lceil \frac{H - H_{kernel} + 1}{p} \rceil * C_{in} * W_{kernel} * H_{kernel} * C_{out} \quad (2)$$

The W and H denote the weight and height of the input data. W_{kernel} and H_{kernel} denote the weight and height of the kernel matrix. C_{in} and C_{out} denote the size of input and output channel. p denotes the stride. We can set $W = W_{kernel}$, $H = H_{kernel}$, $p = 1$, and $C_{in} = 1$ for a fully connected layer where the Equation 2 becomes $W * H * C_{out}$, which equals the size of the weight matrix.

After obtaining the cost $\mathscr{C}$ of the NN, we further construct a relationship between computation cost $\mathscr{C}$ and datapath efficiency. In our paper, we mainly use maximum throughput of the datapath to demonstrate kernel-efficiency, but our design is general to support other metrics, such as base latency. Figure 10 shows the relationship. We can observe the throughput thr and cost $\mathscr{C}$ follow a linear relationship and can be modeled as $thr = a \times \mathscr{C} + b$, where a and b are server-specific parameters related to hardware configuration. LiteFlow obtains these parameters by performing several testing experiments before going to production environment.

2) Structure Pruning: Why structure pruning? Kernel functions are latency-critical and resource-sensitive, so the NN to be deployed in the kernel must be of moderate size. There are two approaches to generate such a NN: 1) directly training a NN of moderate size or 2) training a deep NN and then pruning it into a smaller one that can fit into the kernel efficiently. We prefer the second approach as it is difficult to pre-decide the size of final NN. In contrast, starting from a deep NN means the model initially has sufficient modeling capacities. As long as we do the NN pruning in a way that minimize the loss of modeling capacity, we has a much better opportunity to generate a NN that is kernel-efficient and in the meanwhile has good performance. Our NN pruning algorithm is described as below.

Algorithm. Our structure pruning algorithm has 2 parameters: the maximum allowed fidelity loss E, and the minimal datapath throughput T. LiteFlow calculates the maximum allowed overhead $\mathscr{C}_{allowed}$ from the T. LiteFlow's goal is to find a structure U to minimize fidelity loss ϵ ($\epsilon < E$) while satisfying the cost $\mathscr{C}_U < \mathscr{C}_{allowed}$.

LiteFlow's algorithm leverages existing structure pruning algorithm: `net-trim` [45], which adopts a layer-wise approach to prune the structure, as shown below:

$$\min_U ||U||_1 \text{ s.t. } ||\sigma(U^T Y^{\ell-1} - Y^\ell)||_F < \epsilon/\ell \quad (3)$$

For ℓ-th layer, `net-trim` tries to find a most sparse weight matrix U, that guarantees the result of original layer output Y^ℓ minus the new output (calculated by multiplying new weight

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Structure Pruning

```
Input  : Model U, maximum fidelity loss E,
         maximum cost 𝒞
Output: Pruned model structure U'
1  fn StructurePruning(U, E, 𝒞)
2    ϵ ← E
3    U' ← net-trim (U, ϵ)
        // leveraging existing algorithm
       to find minimal U that satisfies ϵ
4    if cost-model (U') > 𝒞 then
5      return null
6    else
7      U'' ← StructurePruning(U, E/2, 𝒞)
8      if U'' == null then
9        return U'
10     else
11       return U''
12     end
13   end
```

matrix U with original input $Y^{\ell-1}$) is less than ϵ/ℓ. The $||U||_n$ denotes the n-norms of the matrix U and the σ denotes the activation function.

Different from the original `net-trim` algorithm to find a most sparse *i.e.* lowest overhead structure under the fidelity bound, LiteFlow in turn tries to minimize the fidelity loss while still keeping the NN kernel-efficient. Since in kernel datapath, we does not need to find the most efficient NN but to achieve a balance between modeling capacity and efficiency. Therefore, LiteFlow's structure pruning algorithm is more practical than original `net-trim` algorithm. Based on this idea, LiteFlow leverages existing structure pruning algorithm and adopts an iterative search algorithm to search the optimal results as demonstrated in Algorithm 1.

IV. Implementation

We provide a hybrid implementation of LiteFlow, which contains both userspace and kernel-space implementations.

A. Userspace Implementation

The userspace implementation of LiteFlow provides a set of Python interfaces for users to implement. It further provides a service to accept a user-defined Python class that implements those interfaces. By allowing users to implement the interfaces, LiteFlow is not tightly coupled with any deep learning or reinforcement learning frameworks, therefore, LiteFlow users can use their preferred frameworks to optimize the NNs. Moreover, by providing these standard APIs, LiteFlow can flexibly support new NN-based algorithms (additional implementation efforts may be needed for a new input collector & output enforcer module and we will discuss it later). Specifically, these interfaces are:

- **NN Freezing Interface:** The interface is used by LiteFlow to generate the NN snapshot (§III-A). To implement the interface, users should save the model and return the path to the saved model.
- **NN Evaluation Interface:** This interface is used to evaluate the NN synchronization (§III-C). The interface requires LiteFlow users to realize two functions: (1) returning the stability value, *e.g.*, training loss. LiteFlow monitors the value for some time to determine if the online adaptation converges, *i.e.*, the value changes in a small range; (2) calculating the output of the userspace-deployed NN when given a set of input data. LiteFlow's userspace service further communicates with the LiteFlow kernel-space module (more details in the next section) to calculate the fidelity loss of the NN snapshot.
- **NN Online Adaptation Interface:** This interface is used to enable online adaptation for NNs in the slow path (§III-B). To implement the interface, LiteFlow users have to include the scripts/programs to tune the NNs. As discussed above, users can leverage any deep learning frameworks, *e.g.*, TensorFlow [6], or reinforcement learning utilities, *e.g.*, GYM [7], to implement model tuning logics.

After receiving the user-defined Python class, LiteFlow's userspace service first invokes the NN Freezing Interface to obtain a saved NN. Then LiteFlow further leverages TensorFlow Lite [6] to quantize the NN, and generates the snapshot to be deployed in the kernel-space fast path. Worth mentioning, LiteFlow also uses TensorFlow Lite to perform model optimization if necessary. Second, it fetches data from the kernel-space in a batch mode via `netlink` and invokes NN Online Adaptation Interface to tune the userspace-deployed NN. After each training batch, LiteFlow further invokes the NN Evaluation Interface to determine if the snapshot needs updating based on both the correctness and necessity metrics.

B. Kernel-Space Implementation

LiteFlow is implemented with Linux kernel v4.15.0. As one of LiteFlow's design goals is to be generic to support adaptive NNs for various datapath functions, we follow the modularization principle to design LiteFlow's kernel-space components. Figure 11 shows how LiteFlow is divided into different modules and we will introduce each module in the following sections.

LiteFlow Core Module: This module realizes 4 major functions. The first function is LiteFlow's core logic, including NN evaluation and updating logics discussed in §III-C and §III-D. Second, it implements a NN manager. The NN manager uses a linked list to manipulate all installed NN snapshots. It provides `lf_register_model` API to install new snapshots. The third function is to implement the collector & enforcer manager to integrate NNs with different datapath functions (will be introduced later). Finally, LiteFlow core module provides a unified inference interface `lf_query_model` for other kernel-space modules to use the NN. Table I summaries the APIs provided by the LiteFlow core module.

LiteFlow `Netlink` Server Module: This module is registered with kernel-space `netlink` subsystem to communicate with the LiteFlow userspace service. Two types of messages are transferred through this `netlink` channel: (1) the newly-collected data for online adaptation, and (2) the output of the NN snapshot when given a set of input data for necessity evaluation.

NN Module: Each NN snapshot is a separate kernel module. As discussed in §III-A, LiteFlow generates kernel-space implementation of the snapshot and invokes `GCC` to compile it into a kernel module (`.ko` file). LiteFlow userspace service invokes `insmod` system call to install the module. In the initial function of the module, we have to invoke the

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

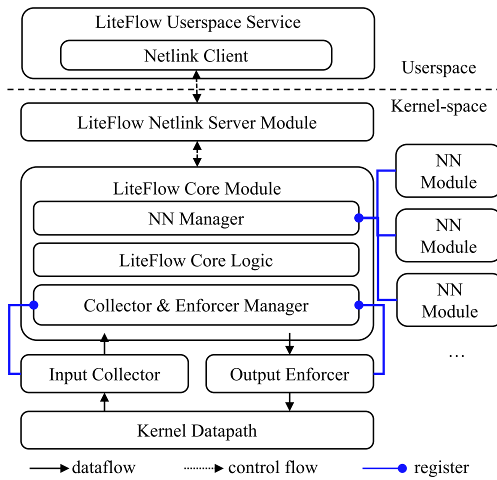


Fig 11. The principle of LiteFlow's kernel-space implementation is modularization.

TABLE I
API OF LITEFLOW CORE MODULE

API	Definition
lf_register_model	Register a new NN snapshot to LiteFlow
lf_register_io lf_unregister_io	Register a new input/output module to LiteFlow Unregister an input/output module from LiteFlow
lf_query_model	Unified inference interface

lf_register_model to register the NN with the LiteFlow core module. During the registration, we need to tell LiteFlow the input and output size of the NN. Worth mentioning, all code of this module is automatically generated by LiteFlow.

Input Collector & Output Enforcer: To support various datapath functions, LiteFlow should give the flexibility to integrate adaptive NNs with different datapath functions. Therefore, LiteFlow requires users to implement their own data collection (*e.g.*, collecting ECN bytes, TCP status, *etc*) and output enforcement logics (*e.g.*, setting the congestion window, flow priority, *etc*, based on the output of the NN) in kernel datapath. Users can use lf_register_io and lf_unregister_io APIs to dynamically add or remove data collection and output enforcement modules. Furthermore, the API will check whether the required input and output size of NN in these user-defined modules are consistent with the installed NN. In this paper, we have implemented three such modules:

- **LiteFlow Congestion Control Module:** The module is plugged into the Linux kernel networking stack as a customized CC algorithm. Each time it receives an ACK, the module collects congestion signals, such as average throughput, *etc*, and uses a NN to predict the target sending rate. To enforce the sending rate in the datapath, the module performs flow pacing by setting the sk_pacing_rate property.
- **LiteFlow Flow Scheduling Module:** The module is plugged into the Linux netfilter subsystem to modify the outbound traffic. It collects metrics such as flow gap, flow start time, *etc*, and uses a NN for flow size prediction. It further tags priority to packets based on the prediction results [46].
- **LiteFlow Path Selection Module:** The module is integrated with XPath [47] for explicit path control. It collects congestion signals, such as ECN bytes, smoothed RTT, *etc*, and uses a NN for path selection. It further leverages explicit path control to enforce path selection.

Next, we discuss how LiteFlow can support new NN-based algorithms. On one hand, LiteFlow has already provided the above 3 modules, which offer some commonly-used features. For example, for CC, LiteFlow Congestion Control Module can collect signals including average throughput, average latency, and latency gradient, which are used by existing works [1], [2]. The module further collects ACKed bytes, ECN bytes, and other CC metrics for future use. If the input features required by the new NN-based algorithms are already provided by our modules, LiteFlow users can directly use these modules. On the other hand, if LiteFlow users need features beyond our modules or they aim to optimize new datapath functions, *e.g.*, queue discipline, they need to develop their own input collector & output enforcer modules. Specifically, they have to build a new kernel module, in which they collect their required features as a vector and then use LiteFlow's API lf_query_model for inference. Meanwhile, they have to implement the logic to enforce the output of the inference to the datapath as well. However, developing a new input collector & output enforcer requires extensive domain knowledge and involves complex kernel-space programming, which remains a challenge for LiteFlow.

V. LITEFLOW APPLICATIONS

To showcase LiteFlow can enable high-performance adaptive NNs for kernel datapath, we use it to optimize 3 popular datapath functions with 4 different NNs and evaluate their performance. Please note that we mainly compare LiteFlow with prior works [1], [2], [9], [19] which all adopt userspace NN deployment. We do not compare LiteFlow with pure kernel-space NN deployment due to the significant performance degradation as discussed in §II-C, which makes it an impractical solution.

A. LiteFlow for Congestion Control

Congestion control (CC) is among the most important network functions. The CC function collects congestion signals, such as RTT, ECN bytes, *etc*, as the input of the NN, and performs NN inference to obtain the output of the NN — the sending rate of the flow. The sending rate will be further enforced to control the speed of the flow to mitigate network congestion. Moreover, the NN can continuously learn and adapt to the network dynamics to tune itself for better CC in the future. The evaluated NNs are as follows:

- **Aurora:** Aurora uses a reinforcement learning algorithm and builds a NN with two hidden fully-connected layers with 32 and 16 neutrons respectively [1]. Aurora extends GYM [7] to build a Python-based networking simulator for NN training and online adaptation. We use Aurora's original code for evaluation [48].
- **MOCC:** MOCC uses multi-objective reinforcement learning and builds a NN with two hidden fully-connected layers with 64 and 32 neutrons respectively [2]. It improves Aurora's design to adding the multi-objective feature, which shows better performance over Aurora.

We use LiteFlow to enable both Aurora (LF-Aurora) and MOCC (LF-MOCC) in kernel datapath, and also use LiteFlow Congestion Control Module introduced in §IV to integrate the NNs with the kernel datapath. We use the testbed from §II-B.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

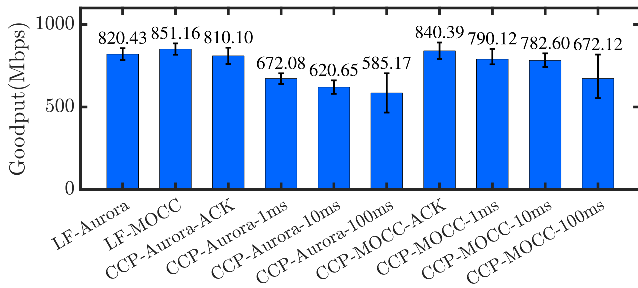


Fig 12. [Congestion Control] Compared to userspace-deployed NNs, LF-Aurora and LF-MOCC can achieve better and more stable flow goodput.

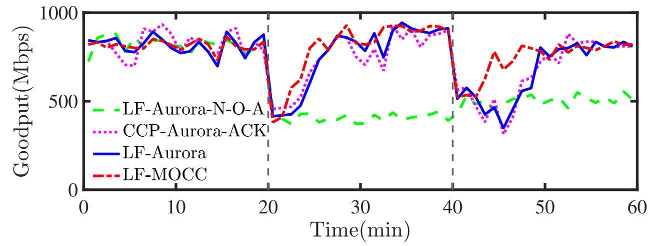


Fig 13. [Congestion Control] LF-Aurora and LF-MOCC can learn and adapt to the environmental dynamics.

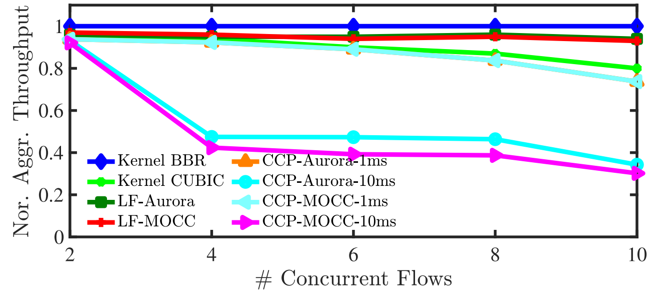


Fig 14. [Congestion Control] LF-Aurora and LF-MOCC suffer from much lower overhead than CCP-Aurora and CCP-MOCC, and achieves comparable overhead as pure kernel-space implementations.

Schemes Compared: We mainly compare LF-Aurora and LF-MOCC with (1) pure userspace deployment of these NNs and (2) traditional CC algorithms in the kernel-space. For pure userspace deployment, we use Congestion Control Plane (CCP) [9] to deploy the NNs, *i.e.*, CCP-Aurora, and CCP-MOCC. CCP requires frequent cross-space communication, and in our evaluation, we vary the communication interval from per-ACK to per-100ms. For traditional CC algorithms in the kernel-space, we choose CUBIC [49] and BBR [5] for evaluation.

Congestion Control Performance: In this experiment, we mainly evaluate the performance of congestion control, *i.e.*, how well can LiteFlow handle network congestion using these NNs? Note that we mainly compare different deployment mechanisms of the same NN-based model instead of comparing different NN-based algorithms and traditional heuristic algorithms. The testbed setting is similar as §II-B, where we use `netem` to set the RTT to be 10ms. In this experiment, we also set the receiver link to 1Gbps (via switch configuration) and generate background UDP traffic (constant rate at 0.1Gbps) to emulate congestion. Then, in each setting, we launch one flow controlled by different schemes and measure its goodput. The results are shown in Figure 12 and the error bar indicates the standard deviation.

We make the following observations. First, flows controlled by LF-Aurora and LF-MOCC achieve higher goodput than those controlled by CCP-Aurora and CCP-MOCC. LF-Aurora and LF-MOCC achieve comparable results to CCP-Aurora-ACK and CCP-MOCC-ACK (we will show in Figure 14 that small intervals cause large overhead) and largely outperforms CCP-Aurora-100ms and CCP-MOCC-100ms by up to 44.4% (from 845.12Mbps to 585.17Mbps) and 26.6% (from 851.16Mbps to 672.12Mbps) respectively. Second, the standard deviation of goodput achieved by LF-Aurora and LF-MOCC is much smaller than CCP-Aurora and CCP-MOCC with large communication intervals. The experiment results show that by eliminating the cross-space communication, LiteFlow makes the NNs more responsive, thus leading to better and more stable performance than userspace-deployed NNs.

Online Adaptation: In this experiment, we mainly evaluate how LiteFlow with these NNs can adapt to environmental dynamics. We use a similar setting as in §II-C. We also disable the NN adaptation function of LiteFlow for comparison (LF-Aurora-N-O-A). Figure 13 shows the results.

Compared to LF-Aurora-N-O-A, which suffers a dramatic goodput drop when the environment changes, LF-Aurora and LF-MOCC can learn and adapt to such dynamics, thus achieving significantly better goodput, which is similar to that achieved by CCP-Aurora-ACK (we will show that small intervals cause large overhead later). Furthermore, LF-MOCC achieves a faster adaptation speed (less than 10 minutes) than LF-Aurora (around 13 minutes), which shows a similar trend as that in the MOCC paper [2].

The experiment results show that by enabling online adaptation, even with batched data, LiteFlow allows the NNs to efficiently learn and adapt to the environmental dynamics.

Overhead: We launch N concurrent flows in one experiment in a non-congested environment (N = 2, 4, 6, 8, 10). In different experiments, flows are controlled by different schemes. We measure the aggregated throughput of the network to denote the overhead caused by these NNs. As a baseline, we also launch flows controlled by BBR and CUBIC respectively, both implemented in the kernel-space. Figure 14 shows the results, and all results are normalized to the aggregated throughput achieved by BBR. Here we use normalized values to highlight the throughput loss.

We have two observations. First, LF-Aurora and LF-MOCC achieve comparable performance to kernel BBR (the performance loss is <5%) and outperform CUBIC by 17.5%. The reason why those NN-based solutions can even outperform CUBIC, a pure kernel implementation, is that these NNs are actually less complicated than CUBIC in which the complex CUBIC function needs to be calculated. Second, LF-Aurora and LF-MOCC largely outperform CCP-Aurora and CCP-MOCC by up to 63.5%. These experimental results show that by eliminating the overhead caused by cross-space communication, LiteFlow with NNs can achieve comparable performance to pure kernel-space implementations.

Batch Data Delivery Interval: To understand how the batch data delivery interval T affects the performance of LiteFlow, we further perform a micro-benchmark experiment with LF-Aurora. We choose different parameters and measure (1) the overhead of LiteFlow using `mpstat` when launching 10 concurrent flows (similar to §II-B) and (2) the average goodput of a single flow (the setting is similar to the previous online adaptation experiment). The result is shown in Figure 15.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

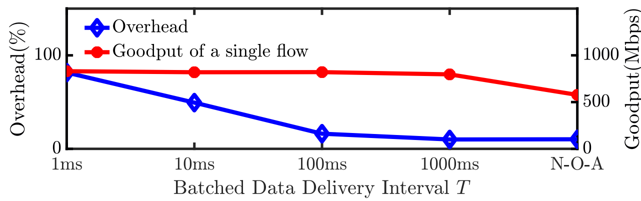


Fig 15. [Congestion Control] Micro-benchmark experiment of how batch data delivery interval affects the performance (N-O-A denotes no online adaptation).

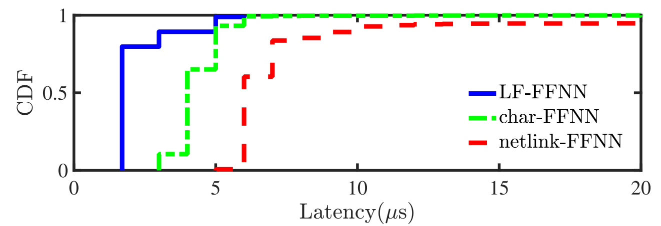


Fig 16. [Flow Scheduling] LF-FFNN can achieve the lowest end-to-end latency when predicating flow size.

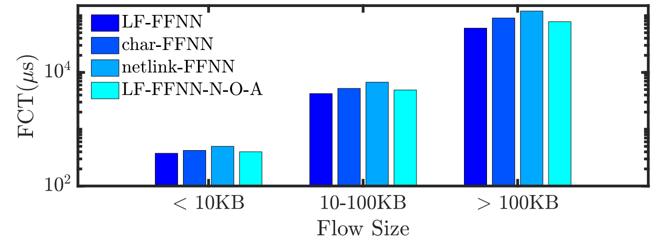


Fig 17. [Flow Scheduling] LF-FFNN can largely outperform other schemes due to its fast predication speed and online adaptation capability.

We observe that when the batch data delivery interval is set between 100ms and 1000ms, the overhead, *i.e.*, the software interrupt time over total CPU execution time, is within $\sim 14.1\%$, achieving similar results as pure kernel CC implementation ($\sim 12.6\%$). Furthermore, the goodput of a flow is not compromised. This experiment confirms that with a proper batch data delivery interval, LiteFlow can reduce the overhead caused by existing userspace NN deployment mechanisms without compromising the NN performance.

High Throughput & Low Latency: Please note that all the above 4 experiments are performed in a network configuration with ∼10ms RTT. Readers may wonder how LiteFlow performs in high throughput & low latency environments, such as DCN. In this experiment, we stop using `netem` for extra latency which creates a high throughput & low latency setting. We have tried to tune Aurora to fit in this setting, but unfortunately, we fail to make Aurora achieve high throughput. Therefore, we create a dummy case, where the NN has the same structure as Aurora but always sets the sending rate to the line rate by modifying the code generated by LiteFlow (LF-Dummy-NN). We launch N concurrent flows in one experiment in a non-congested environment (N = 2, 4, 6) and measure the aggregated throughput. Due to space limitation, we omit the detail results and only present the summary: LF-Dummy-NN can achieve as high throughput as pure kernel-space BBR, where the performance degradation is within 5%.

B. LiteFlow for Flow Scheduling

Flow scheduling is employed to complete flows quickly and/or to meet deadlines [50]. Most flow scheduling algorithms assume the flow size is known, while it is not practical in most cases [51]. Recently, people begin to use learning-based solutions to predict the flow size to achieve better flow scheduling [19]. We choose one of such solutions to evaluate how LiteFlow can improve flow scheduling.

FFNN: FFNN is a feed-forward neural network used to predict flow size in FLUX [19]. FFNN has 2 hidden layers with a ReLU activation function. Each hidden layer has 5 neutrons. We use LiteFlow to enable FFNN (LF-FFNN) in kernel datapath, and also use LiteFlow Flow Scheduling Module introduced in §IV to integrate the NN with the kernel datapath.

Schemes Compared: We mainly compare LF-FFNN with pure userspace inference solutions. We deploy FFNN with TensorFlow [6] in userspace and implements a cross-space communication method to pass the predicted priority to the kernel-space and use a kernel module similar to LiteFlow flow scheduling module to tag priority. The kernel module also collects metrics needed by the FFNN and sends it back to the userspace deployed NN for inference.

To make the comparison comprehensive, we implement the communication in two different ways: one is to use char device (char-FFNN), the other is to use netlink (netlink-FFNN). We also disable the online adaptation for LF-FFNN for evaluation (LF-FFNN-N-O-A).

Experiment Settings & Methodology: As flow scheduling usually needs large-scale testbed with advanced switch hardware, we use both testbed and simulator-based experiments in this section. We first measure the prediction latency of LF-FFNN, char-FFNN, and netlink-FFNN on our testbed. Second, we encode the inference latency in our simulator. For LF-FFNN, we use ns3-gym [52], [53] to further allow NN to adapt to the changing environment. The simulated topology is a 2×2 spine-leaf topology with 32 servers. We use DCTCP [54] as our CC algorithm. The workload we use is also from the DCTCP paper.

Predication Latency: First, we measure the prediction latency of the three mechanisms on our testbed, and Figure 16 shows the CDF of measured latency. The average inference latency of LF-FFNN is around 2.19μs, which is 49.5% smaller than char-FFNN (4.34μs) and 73.6% smaller than netlink-FFNN (8.09μs). Moreover, the prediction latency of LF-FFNN is more stable than userspace deployed NN. The results demonstrate that LiteFlow can largely reduce the predication latency by eliminating the cross-space communication. We will show subsequently that the fast inference can eventually result in better flow completion time (FCT).

Flow Completion Time: We further evaluate how LiteFlow with FFNN performs in large-scale networks through simulated experiments. In this experiment, we launch ~ 4000 flows and measure their FCT. The results are shown in Figure 17 (the Y-axis is in log scale). We further break the results into FCT of short flows (<10KB), middle flows (10-100KB), and long flows (>10KB).

We mainly make two observations. First, LF-FFNN can largely outperform the other two userspace-deployed solutions in all cases. Particular, LF-FFNN outperforms char-FFNN by 10.9% for short flows (377μs vs 423μs) and 33.7% for long flows (60232μs vs 90823μs). The results show that by reducing the prediction latency, LiteFlow can benefit flow

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

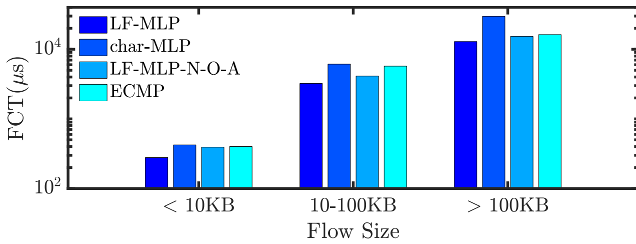


Fig 18. [Load Balancing] LF-MLP can outperform other schemes in supporting load balancing function.

scheduling applications in a large-scale environment. Second, LF-FFNN can outperform LF-FFNN-N-O-A by 6.0% for short flows (377μs vs 401μs) and 23.0% for large flows (60232μs vs 78201μs). The results further demonstrate that by enabling online adaptation, LiteFlow is well suited for optimizing datapath functions in a large-scale deployment full of dynamics.

C. LiteFlow for Load Balancing

Load balancing is critical for data centers with multiple paths to deliver high throughput and low latency. Researchers have started to leverage NNs at end hosts to intelligently and adaptively select paths to optimize the traffic load among all available paths [55], [56]. In this section, we design a multi-layer perceptron model (MLP) for traffic load balancing and evaluate it with LiteFlow to see how can LiteFlow improve the NN-based load balancing function.

MLP: MLP model has 2 hidden layers with a ReLU activation function, and each layer has 12 neutrons. We use supervised learning to optimize the MLP model and the model is optimized for better flow completion time.

We develop and train the MLP model with TensorFlow [6]. We use LiteFlow to enable MLP (LF-MLP) in kernel datapath, and also use LiteFlow Path Selection Module introduced in §IV to integrate the NN with the kernel datapath.

Schemes Compared: Similarly, we design a userspace deployment of MLP model via char device (char-MLP) as §V-B. Furthermore, we also disable the online adaptation feature of LiteFlow for comparison (LF-MLP-N-O-A). We also use ECMP [57] as a baseline.

Experiment Settings: We use a 2×2 spine-leaf topology with 8 servers. We use DCTCP [54] as our CC algorithm. We also use the web search traffic workload mentioned in the DCTCP paper for evaluation.

Experiment Results: We mainly measure the FCT of different schemes and the results are shown in Figure 18 (the Y-axis is in log scale). Similarly, we also break the results into FCT of short flows, middle flows, and long flows.

We make the following three observations. First, LF-MLP can largely outperform other schemes. It outperforms userspace-deployed MLP, *i.e.*, char-MLP, by 34.3% (278μs vs 423μs) on short flows and 56.7% (12922μs vs 29812μs) on long flows. Second, to our surprise, char-MLP even performs worse than the naive load balancing scheme, ECMP. We believe the reason is that char-MLP suffers from increasingly large overhead caused by cross-space communication which leads to severe datapath performance degradation. Third, LF-MLP can perform better than LF-MLP-N-O-A, which again confirms that it is crucial for NNs to learn and adapt to the environment to deliver superb performance in the networking context.

VI. Deep-Dive

A. Model Optimization

In this section, we extend the Aurora and FFNN models used in previous sections to make them have more parameters. Specifically, the original Aurora model contains two hidden FC layers with 32 and 16 neutrons respectively (Aurora-32×16) and we extend it with a 160×80 (Aurora-160×80) and 320×160 (Aurora-320×160) structure. We also extend the FC layer of the FFNN to contain 50 (FFNN-50) and 100 (FFNN-100) neutrons respectively while the original FFNN contains one hidden FC layer with 5 neutrons (FFNN-5). Then we will evaluate how LiteFlow's model optimization mechanism improves the overall performance when deploying these huge models.

As Table II shows, LiteFlow's structure pruning algorithm also preserves high fidelity. When LiteFlow prunes Aurora, it causes the most fidelity loss, reaching around 4%. When the original net-trim paper reports that we can usually prune a model by up to 95%, LiteFlow does not prune that much, *e.g.*, LiteFlow prunes 34.5% for Aurora. The reason is that LiteFlow adopts an iterative search algorithm (§III-E.2) to minimize the fidelity loss while still achieving kernel-efficiency. We then use `iPerf` to launch one single TCP flow from one server to another server and measure the throughput. The results are shown in Table II. We also measure the throughput of BBR with no flow scheduling policy and obtain 13.2Gbps as the baseline.

In general, throughput decreases when the size of the deployed NN model becomes larger, *e.g.*, compared to the original Aurora, throughput of Aurora-320×160 decreases 28.4% (from 13.1 to 9.38Gbps). Then we use LiteFlow to prune a large NN, the datapath can reach comparable throughput as BBR, a pure kernel CC implementation (BBR: 13.2Gbps, Aurora: 13.1Gbps, FFNN: 13.2Gbps, pruned Aurora-320×160: 13.1Gbps, pruned FFNN-100: 13.0Gbps). The results indicate that by performing model optimization, LiteFlow could improve datapath efficiency while not losing too much fidelity.

B. NN Snapshot Generation & Deployment Time

We artificially generate random NNs with different numbers of parameters. Then we measure the time of using LiteFlow to generate and deploy the NN snapshot. We further break the time down into: (1) quantization time, (2) code generation time, (3) code compiling time, and (4) kernel module installation time. Figure 19 shows the results and we have following four observations:

1. The overall time increases when the NN has more parameters, and LiteFlow needs $\sim$ 15s to generate the snapshot and complete deployment for a NN with > 6000 parameters. Taking real-world NN, Aurora, for example, which has 1537 parameters, LiteFlow needs only $\sim$ 7s to generate and deploy the snapshot in datapath. Compared to the time needed for the online adaptation discussed in §III-B, the short NN snapshot generation & deployment time indicates that LiteFlow can well satisfy the needs of NN evolution.
2. The kernel module installation time is very small ($< 1\%$) and can be ignored.
3. The time taken in quantization and code generation preserves stable even when the number of parameters increases. Please note, these 2 parts are the core design

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

TABLE II
[DEEP-DIVE] LITEFLOW'S MODEL OPTIMIZATION MECHANISM

	Original Throughput	Fidelity Loss	Pruned	Throughput
Aurora-32×16	13.1Gbps	-	-	-
Aurora-160×80	12.1Gbps (7.6% ↓)	< 3%	8.0%	13.2Gbps
Aurora-320×160	9.38Gbps (28.4% ↓)	< 4%	34.5%	13.1Gbps
FFNN-5	13.2Gbps	-	-	-
FFNN-50	13Gbps (1.5% ↓)	< 1%	2.1%	13.2Gbps
FFNN-100	12.7Gbps (3.8% ↓)	< 2%	3.7%	13.0Gbps

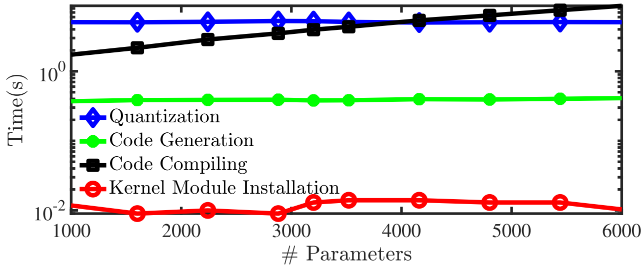


Fig 19. [Deep-dive] NN Snapshot Generation & Deployment Time.

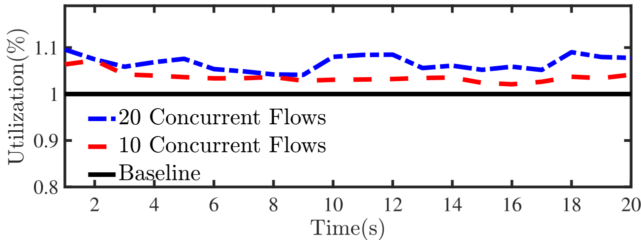


Fig 20. [Deep-dive] CPU Utilization.

of LiteFlow's NN snapshot generation, showing that LiteFlow is able to support even larger NNs with millions of parameters.

4. The code compiling time increases when the number of parameters increases because of increasing size of the generated code. The compiling time can be further reduced by involving parallel compiling, which is one of our future work.

C. CPU Utilization

We measure the CPU utilization of serving N long-lived flows after we deploy LF-FFNN (§V-B) in the kernel-space. In this experiment, we pick N=10 and 20. As a baseline, we also measure the CPU utilization without any NN in the kernel-space. Figure 20 shows the results (normalized to baseline). We observe that although more concurrent flows lead to higher CPU utilization, the CPU utilization is within 10%. The results demonstrate that LiteFlow's implementation is efficient and consumes reasonable CPU resources.

VII. DISCUSSION

A. LiteFlow v.s. Orca

Orca [10] was proposed to combine the control of the NN with the classic TCP control logic, thus leading to the two-level control logic. One of the main reasons that Orca adopts such a two-level method is that pure NN-based solutions suffer from large deployment overhead, which is the exact focus of LiteFlow. However, different from Orca, LiteFlow takes an alternative solution by decoupling the control path of the adaptive NNs into both kernel-space and userspace to solve this problem. Compared to Orca, LiteFlow gains the ability to optimize the CC in an end-to-end way, which could eliminate the restrictions from the TCP control logic, leading to better performance. Moreover, we believe that by using LiteFlow, CC can largely benefit from the increasing popularity of the NN-based algorithm development, such as Spine [58].

B. Limitations of LiteFlow

In this paper, we mainly summarize the limitations of LiteFlow into two aspects. First, since LiteFlow relies on the Linux kernel for implementation, LiteFlow cannot be integrated with the kernel-bypass networking stacks, such as DPDK [59] and RDMA [60]. We observe that these kernel-bypass networking stacks also need a mechanism similar to LiteFlow because the NN inference should be occurred in the fast path with limited resources, such as the RDMA NIC, while the NN tuning can be done in the slow path, such as software driver of RDMA. Second, the current implementation of LiteFlow uses the Linux kernel module, which restricts its flexibility (more details in §IV-B). We can utilize some state-of-the-art Linux kernel facilities, such as eBPF [61], for LiteFlow's implementation to enable easier and safer application integration.

VIII. RELATED WORKS

Userspace-deployed NNs: Recently, NNs have been extensively used to optimize networking datapath functions since they can learn and adapt to environmental variations, making them ideal solutions in networking environment which is full of dynamics. NNs have been used in CC [1], [2], [10], packet classification [18], [62], packet forwarding & routing [3], scheduling [4], [19], load balancing [55], [56], *etc*. Various frameworks are designed to deploy these adaptive NNs in the userspace [6], [20], [21], [63]. An easy way to integrate the NNs with kernel-space datapath is using tools such as CCP [9]. However, these userspace-deployed adaptive NNs suffer from performance degradation as discussed in §II-B.

Lightweight NNs for Inference: Converting a NN into a lightweight one has been a mature technology to achieve efficient NN inference. Specially, in the embedded system/hardware accelerator context, integer quantization [16], [17], [35] has been used to optimize NN inference on low-power IoT devices [36], [37], FPGA [14], GPU [38], [39] and SmartNICs [40]. Moreover, in the networking community, NuevoMatch proposes to convert a NN into a C/C++-based decision tree for efficient execution [18]. While we can leverage these lightweight NNs to deploy the NN in kernel-space

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

for efficient inference, they cannot react to environmental dynamics, thus causing suboptimal performance as discussed in §II-C.

Kernel-space Deployment of both Model Training & Inference: KMLib [11] targets at building a complete NN training and inference library directly in the kernel-space. However, to support these functions, it has to sacrifices NN accuracy by utilizing low-precision model training [64]. Furthermore, using SIMD/FP instructions further degrades the performance as discussed in §II-C. Worth-mentioning, although KMLib can work in a decoupling mode which is similar to LiteFlow, it does not explore the challenges behind such decoupling design. Therefore, it is impractical to be deployed to optimize kernel-space datapath functions.

Online Adaptation: Online adaptation, *i.e.*, online machine learning, presents a set of machine learning algorithms that can optimize NNs over a stream of sequential data [65], [66], [67]. In the networking context, online adaptation is also widely adopted [1], [3], [4], [19], [55], [56]. However, lack of an efficient deployment mechanism leads to rare adoption of these adaptive NNs in the production environment.

IX. Conclusion

This paper proposed LiteFlow, a hybrid solution to enable high-performance adaptive NNs for kernel datapath functions. Experiment results with 3 popular datapath functions have demonstrated that LiteFlow is a viable solution for achieving its design goals. The code of LiteFlow is publicly available via https://github.com/snowzjx/liteflow.

Acknowledgment

The work was done when Shuihai Hu was at Clustar Technology Co., Ltd.

References

[1] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proc. ICML*, 2019, pp. 3050–3059.
[2] Y. Ma et al., "Multi-objective congestion control," in *Proc. ACM EuroSys*, Mar. 2022, pp. 218–235.
[3] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to route," in *Proc. ACM HotNets*, 2017, pp. 185–191.
[4] L. Chen, J. Lingys, K. Chen, and F. Liu, "AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proc. ACM SIGCOMM*, Aug. 2018, pp. 191–205.
[5] 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.
[6] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. USENIX OSDI*, 2016, pp. 265–283.
[7] G. Brockman et al., "OpenAI gym," 2016, *arXiv:1606.01540*.
[8] Y. Gu and R. L. Grossman, "UDT: UDP-based data transfer for high-speed wide area networks," *Comput. Netw.*, vol. 51, no. 7, pp. 1777–1799, May 2007.
[9] A. Narayan et al., "Restructuring endpoint congestion control," in *Proc. ACM SIGCOMM*, Jul. 2018, pp. 30–43.
[10] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the internet," in *Proc. ACM SIGCOMM*, Jul. 2020, pp. 632–647.
[11] I. U. Akgun, A. S. Aydin, and E. Zadok, "KMLIB: Towards machine learning for operating systems," in *Proc. On-Device Intell. Workshop, Co-Located MLSys Conf.*, 2020, pp. 1–6.
[12] (2020). *Linux Kernel V4.1.5*. [Online]. Available: https://lwn.net/Articles/654091/
[13] P. J. Salzman, M. Burian, and O. Pomerantz, *The Linux Kernel Module Programming Guide*. Scotts Valley, CA, USA: CreateSpace Independent Publishing Platform, 2007.
[14] M. S. Abdelfattah et al., "DLA: Compiler and FPGA overlay for neural network inference acceleration," in *Proc. IEEE FPL*, Aug. 2018, p. 4117.
[15] K. Guo et al., "From model to FPGA: Software-hardware co-design for efficient neural network acceleration," in *Proc. IEEE Hot Chips*, Aug. 2016, pp. 1–27.
[16] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE CVPR*, Jun. 2018, pp. 2704–2713.
[17] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018, *arXiv:1806.08342*.
[18] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "A computational approach to packet classification," in *Proc. ACM SIGCOMM*, Jul. 2020, pp. 542–556.
[19] V. Đukić, S. A. Jyothi, B. Karlaš, M. Owaida, C. Zhang, and A. Singla, "Is advance knowledge of flow sizes a plausible assumption?" in *Proc. USENIX NSDI*, 2019, pp. 565–580.
[20] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. NeurIPS*, 2019, pp. 1–12.
[21] T. Chen et al., "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*.
[22] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proc. ACM SIGCOMM*, Aug. 2017, pp. 253–266.
[23] A. Langley et al., "The QUIC transport protocol: Design and internet-scale deployment," in *Proc. ACM SIGCOMM*, Aug. 2017, pp. 183–196.
[24] D. Firestone et al., "Azure accelerated networking: SmartNICs in the public cloud," in *Proc. USENIX NSDI*, 2018, pp. 51–66.
[25] Y. Le et al., "UNO: Uniflying host and smart NIC offload for flexible packet processing," in *Proc. SoCC*, Sep. 2017, pp. 506–519.
[26] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Müller, "BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing," in *Proc. USENIX NSDI*, 2021, pp. 487–501.
[27] J. Ousterhout, "A Linux kernel implementation of the Homa transport protocol," in *Proc. USENIX ATC*, 2021, pp. 99–115.
[28] Q. Xu, M. D. Wong, T. Wagle, S. Narayana, and A. Sivaraman, "Synthesizing safe and efficient kernel extensions for packet processing," in *Proc. ACM SIGCOMM*, Aug. 2021, pp. 50–64.
[29] (2020). *Mellanox SN2100 Switch*. [Online]. Available: https://www.mellanox.com/products/ethernet-switches/sn2000
[30] (2020). *Netem*. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-netem.8.html
[31] F. Y. Yan et al., "Pantheon: The training ground for internet congestion-control research," in *Proc. USENIX ATC*, 2018, pp. 731–743.
[32] (2020). *Mpstat*. [Online]. Available: https://man7.org/linux/man-pages/man1/mpstat.1.html
[33] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. COMPSTAT*, 2010, pp. 177–186.
[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
[35] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021, *arXiv:2103.13630*.
[36] A. Kumar, V. Seshadri, and R. Sharma, "Shiftry: RNN inference in 2 KB of RAM," *Proc. ACM Program. Lang.*, vol. 4, pp. 1–30, Nov. 2020.
[37] (2022). *Neural Network Optimization With AIMET*. [Online]. Available: https://developer.qualcomm.com/blog/neural-network-optimization-aimet
[38] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *Proc. EuroSys*, 2019, pp. 1–15.
[39] S. Gupta, M. Imani, H. Kaur, and T. S. Rosing, "NNPIM: A processing in-memory architecture for neural network acceleration," *IEEE Trans. Comput.*, vol. 68, no. 9, pp. 1325–1337, Sep. 2019.
[40] G. Siracusano, D. Sanvito, S. Galea, and R. Bifulco, "Deep learning inference on commodity network interface cards," in *Proc. NeurIPS*, 2018, pp. 1–8.
[41] (2020). *Python Jinja*. [Online]. Available: https://jinja.palletsprojects.com/en/3.0.x
[42] (2020). *GCC, the GNU Compiler Collection*. [Online]. Available: https://gcc.gnu.org
[43] T. G. Goodwillie, "Calculus III: Taylor series," *Geometry Topol.*, vol. 7, no. 2, pp. 645–711, Oct. 2003.
[44] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. ACM IMC*, Nov. 2009, pp. 202–208.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.

[45] A. Aghasi, A. Abdi, N. Nguyen, and J. Romberg, "Net-trim: Convex pruning of deep neural networks with performance guarantee," in *Proc. NeurIPS*, 2017, pp. 1–10.

[46] M. Alizadeh et al., "pFabric: Minimal near-optimal datacenter transport," in *Proc. ACM SIGCOMM*, Aug. 2013, pp. 435–446.

[47] S. Hu et al., "Explicit path control in commodity data centers: Design and applications," in *Proc. USENIX NSDI*, 2015, pp. 15–28.

[48] (2020). *Aurora Codebase*. [Online]. Available: https://github.com/PCCproject/PCC-RL

[49] 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.

[50] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling mix-flows in commodity datacenters with karuna," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 174–187.

[51] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proc. USENIX NSDI*, 2015, pp. 455–468.

[52] (2020). *NS3-Gym*. [Online]. Available: https://www.nsnam.org/news/2018/12/07/ns3-gym-app.html

[53] J. Zhang, W. Bai, and K. Chen, "Enabling ECN for datacenter networks with RTT variations," in *Proc. ACM CoNEXT*, Dec. 2019, pp. 233–245.

[54] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.*, Aug. 2010, pp. 63–74.

[55] S. WilsonPrakash and P. Deepalakshmi, "Artificial neural network based load balancing on software defined networking," in *Proc. INCOS*, Apr. 2019, pp. 1–4.

[56] H. Yao, X. Yuan, P. Zhang, J. Wang, C. Jiang, and M. Guizani, "A machine learning approach of load balance routing to support next-generation wireless networks," in *Proc. IWCMC*, Jun. 2019, pp. 1317–1322.

[57] C. Hopps et al., *Analysis of an Equal-Cost Multi-Path Algorithm*, document RFC 2992, Nov. 2000.

[58] H. Tian, X. Liao, C. Zeng, J. Zhang, and K. Chen, "Spine: An efficient DRL-based congestion control with ultra-low overhead," in *Proc. ACM CoNEXT*, Nov. 2022, pp. 261–275.

[59] (2023). *DPDK*. [Online]. Available: https://www.dpdk.org,

[60] C. Guo et al., "RDMA over commodity Ethernet at scale," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 202–215.

[61] (2023). *eBPF*. [Online]. Available: https://ebpf.io

[62] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *Proc. ACM SIGCOMM*, 2019, pp. 256–269.

[63] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proc. USENIX NSDI*, 2017, pp. 613–627.

[64] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, 2017.

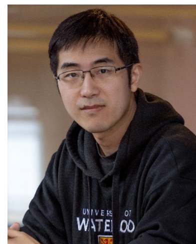
[65] M. Zinkevich, "Online convex programming and generalized infinitesimal gradient ascent," in *Proc. ICML*, 2003, pp. 928–936.

[66] N. Cesa-Bianchi and G. Lugosi, *Prediction, Learning, and Games*. Cambridge, U.K.: Cambridge Univ. Press, 2006.

[67] D. Sahoo, Q. Pham, J. Lu, and S. C. H. Hoi, "Online deep learning: Learning deep neural networks on the fly," in *Proc. IJCAI*, Jul. 2018, pp. 2660–2666.

Junxue Zhang received the B.S. and M.S. degrees from Southeast University 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. His research interests include data center networking, machine learning systems, and privacy-preserving computation. His research work has been published in many top conferences, such as SIGCOMM, NSDI, and CoNEXT.

Chaoliang Zeng received the B.S. degree from the University of Science and Technology of China, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, supervised by Prof. Kai Chen. His research interests include datacenter systems with a focus on hardware acceleration, high-speed networking, and machine learning systems.



Hong Zhang received the Ph.D. degree in computer science and engineering from The Hong Kong University of Science and Technology (HKUST). He is currently an Assistant Professor with the David R. Cheriton School of Computer Science, University of Waterloo. Previously, he was a Post-Doctoral Scholar with UC Berkeley. His research interests include computer systems and networking, with a special focus on distributed data analytics and ML systems, data center networking, and serverless computing.

Shuihai Hu received the B.S. degree in computer science from the University of Science and Technology of China (USTC), China, in 2013, and the Ph.D. degree from the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, in 2019. He is currently a Researcher at Huawei. His research interests include real-time media communication, datacenter networks, and machine learning systems.

Kai Chen (Senior Member, IEEE) received the B.S. and M.S. degrees from the University of Science and Technology of China (USTC), China, in 2004 and 2007, respectively, and the Ph.D. degree from Northwestern University in 2012. He is currently a Professor with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology (HKUST), the Director of the Intelligent System and Networking Laboratory (iSING Lab) and WeChat-HKUST Joint Laboratory for Artificial Intelligence Technology (WHAT Lab), and the Executive Vice-President of the Hong Kong Society of Artificial Intelligence and Robotics (HKSAIR). His work has been published in various top venues, such as SIGCOMM, NSDI, and IEEE/ACM TRANSACTIONS ON NETWORKING (TON), including a SIGCOMM Best Paper Candidate. His current research interests include data center networking, machine learning systems, and privacy-preserving computing. He is the Steering Committee Co-Chair of APNet. He serves on Program Committees of SIGCOMM, NSDI, and INFOCOM, and editorial boards of IEEE/ACM TRANSACTIONS ON NETWORKING, *Big Data*, and *Cloud Computing*.

Authorized licensed use limited to: Hong Kong University of Science and Technology. Downloaded on March 01,2024 at 07:14:05 UTC from IEEE Xplore. Restrictions apply.