Carlo: Cross-Plane Collaboration for Multiple In-network Computing Applications

Xiaoquan Zhang[1], Lin Cui[1], WaiMing Lau[1], Fung Po Tso[2], Yuhui Deng[1] and Weijia Jia[3,4]
[1]Department of Computer Science, Jinan University, Guangzhou.
[2]Department of Computer Science, Loughborough University, UK.
[3]BNU-UIC Institute of Artificial Intelligence and Future Networks, Beijing Normal University (BNU Zhuhai).
[4]BNU-HKBU United International College, Zhuhai, China.

***Abstract*—In-network computing (INC) is a new paradigm that allows applications to be executed within the network, rather than on dedicated servers. Conventionally, INC applications have been exclusively deployed on the data plane (e.g., programmable ASICs), offering impressive performance capabilities. However, the data plane's efficiency is hindered by limited resources, which can prevent a comprehensive deployment of applications. On the other hand, offloading compute tasks to the control plane, which is underpinned by general-purpose servers with ample resources, provides greater flexibility. However, this approach comes with the tradeoff of significantly reduced efficiency, especially when the system operates under heavy load. To simultaneously exploit the efficiency of data plane and the flexibility of control plane, we propose *Carlo*, a cross-plane collaborative optimization framework to support the network-wide deployment of multiple INC applications across both the control and data plane. *Carlo* first analyzes resource requirements of various INC applications across different planes. It then establishes mathematical models for resource allocation in cross-plane and automatically generates solutions using proposed algorithms. We have implemented the prototype of *Carlo* on Intel Tofino ASIC switches and DPDK. Experimental results demonstrate that *Carlo* can compute solutions in a short time while avoiding performance degradation caused by the deployment scheme.**

***Index Terms*—P4, Cross-plane collaboration, Deployment optimization, Programmable data plane**

I. Introduction

The new paradigm of in-network computing (INC) disrupts traditional cloud server-based computing by enabling computations to be partially or fully offloaded to network elements, leveraging their line rate processing capabilities [1]. INC applications encompass not only new computing services, such as in-cache [2], but also traditional network functions, like flow detection [3]. Emerging programmable data planes (PDP), exemplified by the Intel Tofino switch ASIC [4], possess high-performance packet processing capabilities of $\geq$6.4Tb/s and a latency of 10µs, significantly reducing packet latency to the submicrosecond level, enabling line rate processing of packets in INC applications.

However, programming the hardware pipeline of PDP, such as Intel Tofino ASIC, presents certain limitations. Firstly, the available memory resources are substantially smaller (in the order of $\mathcal{O}(MB)$) compared to the $\mathcal{O}(GB)$ typically found in commodity servers. This restricts the deployment of stateful applications that require larger memory storage for network states (e.g., stateful NAT [5]). Secondly, the limited computing capability, primarily supporting addition and bit operations, hinders the implementation of complex functions. Hence, it is challenging, if not impossible, to design intricate functions within the pipeline without making compromises that cause either a performance decrease [6] or higher resource consumption [7].

TABLE I: Comparison of devices across planes.

Device	Throughput	Memory	Operation	Latency	Price
Server (control plane)	$\mathcal{O}(10Gbps)$	$\mathcal{O}(GB)$	Complex	$\mathcal{O}(ms)$	\$3K
Switch (data plane)	$\mathcal{O}(6.4Tbps)$	$\mathcal{O}(MB)$	Simple (bit shift, hash, etc.)	$\mathcal{O}(\mu s)$	\$10K

Table I provides a comparison between PDP switches (data plane) and commodity servers (control plane). Given the distinctive characteristics of the control plane and data plane, collaborative efforts across planes can undoubtedly enhance application performance by leveraging the strengths of both the data plane and the control plane. For example, the high-performance execution of an in-cache application [2] relies on precise matching tables in the data plane, while the dynamic update strategy to maintain a high hit rate necessitates implementation in the control plane.

Unfortunately, the issue of cross-plane collaborative optimization remains largely unexplored. Some works have discussed cross-plane collaboration for individual INC applications. SyNAPSE [8] and CLIP [9] proposed frameworks to generate program code in both planes based on empirical deployment decisions. However, comparing with deploying individual INC applications, we argue that exploiting the implementation disparities of multiple applications can lead to optimizing overall performance and resource utilization more effectively.

Previous research has explored the network-wide deployment of multiple applications, but these studies have not explored the potential benefits of cross-plane collaboration [10]–[13]. For example, MTP [10] focused on optimizing the placement of measurement functions, with the control plane exclusively responsible for handling analysis tasks generated

Corresponding author: Lin Cui. Email: tcuilin@jnu.edu.cn

by the data plane. Similarly, ClickINC [13] deployed applications on multiple data plane paths, with programmable network elements (such as network interface cards and programmable switches) processing the applications along the path. Besides, some works discussed offloading conventional NFV on PDP, they either only allowed deploying an entire network function on a single plane [14], or enhanced the performance of table-based functions (e.g., NAT) by offloading specific logic (e.g., forwarding) on a single switch [15].

In this paper, we propose *Carlo* (Cross-plane Collaborative Solution) for optimizing the deployment of multiple INC applications with cross-plane collaboration. To support a wide diversity of INC applications, *Carlo* provides a fine-grained solution that automatically decomposes INC applications into logic modules and chooses each logic module to deploy on appropriate planes and optimize performance. *Carlo* consists of three key components: application analyzer, deployment model, and optimization. The application analyzer decomposes the logic of each INC application as nodes in Control Flow Graph (CFG) and identifies their required deployment resources in cross-plane. The deployment model establishes an Integer Linear Programming (ILP) model that allows nodes to be placed in different planes to optimize deployment objectives and restricts limitations in cross-plane in case of suffering performance degradation or even deployment failures. The optimization module employs algorithms to balance performance and search time.

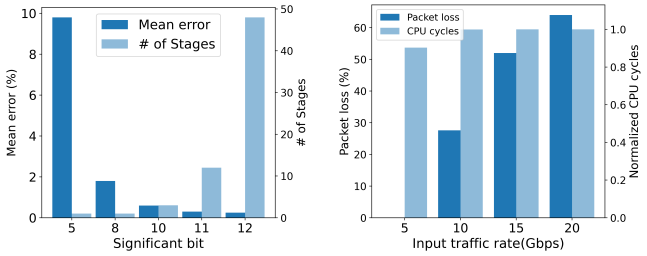
In short, the contributions of this paper are as follows:

1) We propose *Carlo*, a cross-plane collaborative solution for deployment optimization of multiple INC applications. To our best knowledge, this is the first work to study cross-plane collaboration for fine-grained deployment to support a wide diversity of INC applications.
2) We analyze and identify INC applications resource requirements in cross-plane; an ILP is proposed to formulate the deployment problem and algorithms are proposed to generate solutions with cross-plane collaborative policies.
3) We have implemented a testbed prototype of *Carlo* based on Intel Tofino ASIC switches and servers with DPDK. Extensive experiments indicate that *Carlo* can optimize the deployment objective while avoiding performance degradation.

II. Background and Motivation

A. Background

PISA (Protocol-Independent Switch Architecture) is well-known for its widespread packet processing model in hardware switches. Users can utilize the high-level language P4 to design applications as needed. PISA incorporates several programmable components, namely the parser, pipeline, and deparser. The parser and deparser are responsible for extracting and reconstructing headers from network packets. The pipeline plays a crucial role in PISA as it allows for customization of application functionality by manipulating match-action tables



(a) Division operation in switches. (b) AES operation in servers.

Fig. 1: Motivating experiments with a simple implementation of two common operations.

(MAT). Computing and storage resources are evenly divided into multiple stages within the pipeline (e.g., SRAM, TCAM, and ALUs), and these resources are independently accessible (e.g., the Tofino 1 pipeline consists of 12 stages).

INC applications. The aim of INC is to obtain orders of magnitude higher throughput and lower latency than that can be achieved by the traditional server through offloading a set of compute operations from end hosts into network elements such as switches [1]. Due to the resource constraints of PDP, a complex operation must be either decomposed into small fractions among multiple stages or applied in lookup tables. For example, inference of a decision tree can be encoded into multiple MATs by carefully splitting the feature spaces and mapping them [6]. Using lookup tables can achieve complex operations (e.g., division [16] and entropy [17]). However, these methods are still challenged by scalability and accuracy.

B. Motivating experiments

We conducted the following experiments to explore and understand how limitations in a single plane affect deployments of the INC application. Division and AES are basic operations commonly used in many data plane applications. Division can be used to compute flow rates for rate control [7], [16], and AES is widely used in security-oriented applications [18]. In the first experiment, we implemented a division operation in an Intel Tofino switch [4] using lookup tables. It can be observed from Fig. 1(a) that as the number of significant bits in the operands of division operation increases, the accuracy also increases. Unsurprisingly, the number of stages required for lookup tables deployment also increases. Interestingly, the linear increase in significant bits incurs an exponentially higher deployment cost (e.g., stage usage), while the improvement in accuracy gains diminishes. In the second experiment, we built a topology where a sender server is connected to a receiver server via a 40Gbps fiber link. The sender generated packets with a size of 250 bytes ranging from 5Gbps to 20Gbps via iperf [19]. The receiver collected AES packets through DPDK and used one CPU core to handle them. In Fig. 1(b), we can observe very high CPU utilization and severe packet loss when we increased the traffic rate of AES packets.

These experiments demonstrate that the inherent constraints in a single plane may result in difficulty satisfying the require-

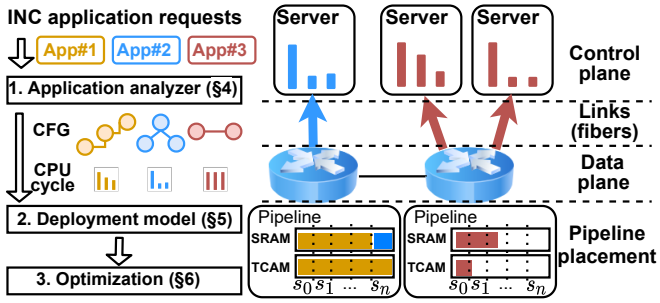


Fig. 2: Overview of architecture.

ments of INC applications. This would produce suboptimal solutions in deployment optimization. Hence, *Carlo* optimizes deployment with the consideration of resource allocation and traffic demands of INC applications.

III. Overview of *Carlo*

Carlo provides a framework for deploying multiple INC applications crossing both the data plane (e.g., switches with Tofino ASIC) and control plane (e.g., CPU-based servers). *Carlo* makes decisions by utilizing the resource across planes, in which the logic of an INC application can be automatically decomposed and deployed in cross-plane. Thus, *Carlo* makes deployment decisions based on two aspects of information. In the data plane, resource allocation must respect limitations in the PISA hardware pipeline to avoid compilation failure (Section II-A). In the control plane, the server needs to allocate resources to process the traffic of an INC application if the application is entirely deployed on the server. Meanwhile, some applications in the data plane would emit traffic to the control plane for analysis (e.g., analyze anomaly flows [20]). Thus, constraints of bandwidth and CPU resources for traffic processing should be considered to avoid performance degradation.

Challenges. However, optimizing the problem in cross-plane is a complicated task. *The first challenge is capturing the various properties of INC applications in cross-plane,* as knowing the properties helps better calculate resource upper bounds as so to avoid performance degradation and deployment failure. A general abstraction for fine-grained placement in the hardware pipeline and a mode summary of traffic needed to be processed in the control plane are needed. *The second challenge is that optimizing deployment objectives in cross-plane is NP-hard.* The solution should generate collaborative policies for deployment in different planes to optimize deployment objectives while respecting constraints in both the data and control planes. Besides, optimization algorithms should make tradeoffs between performance and execution time in different deployment objectives.

Architecture. To address the above challenges, *Carlo* proposes three modules (shown in Fig. 2), including an *application analyzer*, a *deployment model*, and *optimization algorithms*. The application analyzer analyzes the deployment resources required for INC applications by CFG, including pipeline resources, bandwidth, and CPU resources in servers. In the deployment model, *Carlo* establishes a mathematical model that formulates resource constraints in cross-plane, in which application logic can be split to be placed in different planes. Optimization algorithms are proposed to exploit deployment solutions with different objectives while balancing algorithm performance and execution time.

Example. In Fig. 2, App#1 can be a simple sketch application consisting of three logic modules (nodes): hash function, sketch table, and forward table, whose deployment satisfies the resource limitations of SRAM and TCAM. App#3 is deployed in both the data plane and control plane and needs server resources to process traffic.

IV. Application Analyzer

A. CFG abstraction of INC applications

Carlo utilizes CFG to analyze and decompose the logic of an application [11]. Given a P4 code, *Carlo* uses a graph tool [21] to abstract components (e.g., MAT) of a P4 program. A P4 program can be defined as a directed acyclic graph $G = \{V,E\}$, where V is the set of nodes that refers to MAT or conditions in P4 programs, and E is the set of directed edges each of which describes the execution direction of two nodes.

Dependency. Two nodes that have a dependency relationship must be placed in different stages in case of violation [4]. For example, two nodes that should mutually modify the same header fields will be placed in two different stages. Thus, a dependency matrix D is defined to represent relationships among different nodes in the CFG, where $D(n_i, n_j) = 1$ if the two nodes have a dependency.

Properties of CFG nodes. A CFG node n_i has three types of variables to represent the deployment resources in cross-plane. The first type of variable is resource usage in the hardware pipeline, including SRAM, TCAM, and ALUs resources, which can be inferred by the calculation of the sizes of MAT and registers [12]. The second variable is the maximum traffic to be processed on the server. The third variable is the amount of CPU resources required to process received traffic on the server. Moreover, resource variables of the same node are determined by which plane the node is placed. For example, if the node is placed in the server, it would not consume any resources in the hardware pipeline. In the meantime, we also notice that nodes would have various traffic modes and CPU resource requirements when they are placed on different planes. Next, we introduce how to estimate the two variables in various INC applications.

B. Resource estimation in the control plane

Due to the collaborative deployment strategy in *Carlo*, the control plane needs to collaboratively process network traffic. However, the total network packets of INC applications may exceed the capacity of the server so as to lead to performance degradation. *Carlo* captures different traffic modes of INC applications to compute the upper bound of traffic. We define the input throughput of an INC application as W, and the

TABLE II: Traffic modes of INC applications.

Type	INC applications	Maximum traffic rate
Packet mirror	DDos detection [20]	W
Metadata mirror	DDos detection [20]	$W_p \cdot \sum_{f \in F} l_f$
Matching mirror	Traffic Engineering [22] Anomaly detection [22]	$W_m \cdot \lvert\mathbb{M}\rvert$
Query	Security detection [23] Monitor [24]	$\frac{\sum_{f \in F} l_f \cdot \delta_f}{t}$
Probabilistic	In-cache [2], [25]	$W \cdot (1 - \hat{\phi})$
Aggregating	In-band network telemetry [26], [27]	W_{int} $\theta \cdot W$
Bi-design	Complex operations [7], [16]–[18], [28]–[32] ML apps [3], [6], [33]	W 0

TABLE III: Notation of symbols.

Notation	Description
b_i^c	Maximum traffic rate of node i in the server c
B^c	Bandwidth between server c and its connecting switch
C	Set of servers $c \in C$
$D(n_i, n_j)$	Dependency between node n_i and n_j
G	Control flow graph, $G = \{V, E\}$, including node set $n_i, n_j \in V$ and edge set E
I_i, O_i	Input and output stage for a node n_i
P	Set of programmable switches $sw \in P$
R_i^m	Type m of resource requirement for node i
U^c	Maximum CPU Utilization of server c
z_{sw}	Whether switch sw has node(s) to be placed
z_c	Whether server c has node(s) to be processed
μ^m	Limited type m of resources in each stage
ϵ_{sw},ϵ_c	Deployment cost for the switch sw and the server c
Variables	
$Q_i^{s,m}$	Type m of allocated resource for node i in stage s
x_i^s	Whether node n_i is placed in stage s
y_i^c	Whether node n_i is placed in server c

rate of maximum traffic to be processed in the server can be estimated by the following modes (summarized in Table II):

Mirror mode. Applications mirror packet copies to the control plane for analysis. Specific packet content can be chosen for mirroring. **Packet mirror mode** directly mirrors the whole packet. Its maximum traffic rate equals the throughput of the INC application (W). **Metadata mirror mode** extracts packet headers to compute monitoring features ($f \in F$) for mirroring. Its maximum traffic rate can be calculated by the multiplication of total input throughput in terms of the number of packets (W_p) and the size of each packet which equals the total bit width of features needed (l_f). Another mode, **matching mirror**, is to mirror all packets of a flow (e.g., an anomaly flow) matching in the MAT entry ($m \in \mathbb{M}$). We define W_m as the maximum traffic rate of the flow matching by m, and the maximum traffic rate of this mode equals the multiplication of the size of MAT ($|\mathbb{M}|$) and W_m.

Query mode. Some applications require the control plane to periodically query the flow information stored in the data plane for deep analysis (e.g., machine learning-based applications and monitors). Let δ_f be the number of monitored flows defined by the demands of the applications. The maximum traffic rate for this mode is equal to the total memory size occupied by all features divided by the time t needed to retrieve these features. For example, a register is used to save flow feature f, whose total memory size equals the multiplication of feature bit width l_f and the number of monitored flows δ_f.

Probabilistic mode. In this mode, whether traffic is forwarded to the control plane is determined by a probabilistic value. The most typical application is in-network cache [2], in which the probabilistic value can be represented by the hit rate in the data plane. Assume the hit rate of the given MAT table is $\hat{\phi}$, then the miss rate is $1 - \hat{\phi}$. Thus, the maximum traffic rate can be $F \cdot (1 - \hat{\phi})$. It is noted that the hit rate is an estimated value and it varies due to network dynamics. In our experiments, we identified the lower bound of $\hat{\phi}$ by analysis of the experimental dataset.

Aggregating mode. INT (In-band Network Telemetry) applications require generating and aggregating INT packets that are processed in the control plane. The packet generation mainly includes two patterns. The first is setting a sample rate (θ) for normal traffic [27], then its maximum traffic rate is $\theta \cdot W$. The second is setting a fixed rate of probe packets [26], then its maximum traffic rate is W_{int} which equals the fixed rate of proactive INT packets.

Bi-design mode. Implementing complex functionality in the pipeline usually consumes more scarce resources. For example, entropy operations [32] need to occupy 20 stages in the Tofino 2 ASIC pipeline. *Carlo* allows users to make the tradeoff between deployment cost and performance. Specifically, applications can be entirely deployed in the control plane, in which the maximum flow rate of the mode is equal to the application's maximum processing rate, or entirely offloaded in the data plane, in which no traffic is forwarded to the control plane.

Example. The second variable of CFG nodes can be estimated by capturing the traffic modes of applications. For example, an in-network machine learning (ML) application can be entirely placed in servers or the data plane for lossless or lossy accuracy (*Bi-design*). It can also offload feature statistics in the data plane to improve packet forwarding throughput (*Query* or *mirror*).

CPU resource is calculated by the number of cycles in *Carlo* to identify resource requirements of INC applications for processing traffic in servers. To estimate that of each INC application, *Carlo* performs benchmark experiments to inject traffic and uses perf [34] to measure their CPU cycles.

V. Deployment Model

An ILP model is established to convert the deployment problem into placing each node ($n_i \in V$) of a compound CFG G that consists of multiple INC applications (i.e., each subgraph represents an application) on cross-plane while satisfying the resource constraints. In the first place, nodes are allowed to be placed in different planes while allocating required resources, and node variables vary depending on their deployment location as estimated in the application analyzer. For example, if a node is placed in the data plane, it consumes pipeline resources and may generate traffic to servers. If it

is placed in servers, it consumes no pipeline resources. In the second place, the network elements consist of a set of programmable switches ($sw \in P$) and a set of servers as the control plane ($c \in C$). Pipeline resources of a switch are evenly divided and represented in the form of the stage ($s \in S$), where S refers to the set of all stages of switches in P. It is noted that we don't consider latency between the data and control plane since we use fiber to link them, whose latency is far lower than that in servers for processing (e.g., 6*ns* of serialization delay in IEEE standard 802.3bm [35]).

Binary variables. x_i^s and y_i^c, are used to denote whether the node n_i is placed in stage s and is placed in the server c, respectively. z_{sw} and z_c, refer to whether switch sw and server c have been placed nodes, respectively.

Optimization constraints include CFG node placement and resource limitations in cross-plane.

(1) Constraints of node placement. Each node n_i must be assigned to one position either in stages or in servers:

$$\forall n_i \in V : \bigvee_{s \in S} x_i^s + \bigvee_{c \in C} y_i^c = 1. \tag{1}$$

Moreover, a node can be placed among multiple stages in the data plane :

$$\forall n_i \in V : \bigvee_{s \in S} x_i^s = 1 \implies \sum_{s \in S} x_i^s \geq 1. \tag{2}$$

A node may consume different types of pipeline resources $M = \{SRAM, TCAM, ALU\}$ for computation and storage. An integer variable $Q_i^{s,m}$ represents the allocated resources of type $m \in M$ for node n_i in stage s. All resources of each node must be allocated completely:

$$\forall n_i \in V, \forall m \in M : \bigvee_{s \in S} x_i^s = 1 \implies \sum_{s \in S} Q_i^{s,m} = R_i^m, \tag{3}$$

where R_i^m refers to the requirement for resource type m of node n_i.

(2) Constraints in the hardware pipeline. Since each stage in the hardware has limited resources μ^m, the total allocation of resources of nodes in a stage must not exceed its capacity:

$$\forall s \in S, \forall m \in M : \sum_{n_i \in V} Q_i^{s,m} \cdot x_i^s \leq \mu^m. \tag{4}$$

If two nodes have a dependency (i.e., $D(n_i, n_j) = 1$), the output stage of node n_i must precede the input stage of node n_j.

$$\forall n_i, n_j \in V : D(n_i, n_j) = 1 \implies O_i < I_j, \tag{5}$$

where I_i and O_i are defined as the input and output stages of a node n_i.

(3) Constraints in the control plane. The accumulation of traffic rates of INC applications should not exceed the capacity of the link between the switch and the server:

$$\forall c \in C : \sum_{n_i \in V} b_i^c \leq B_c, \tag{6}$$

where b_i^c refers to the maximum traffic rate of node i.

For a server c, its capacity of CPU resources should be higher than the total CPU resources required by all nodes that it processes:

$$\forall c \in C : \sum_{n_i \in V} u(b_i^c) \leq U_c, \tag{7}$$

where $u()$ refers to the computation of CPU utilization of a node.

Objectives. *Carlo* provides two common optimization objectives for model optimization separately. The first objective (#1) is to maximize overall throughput. It can be beneficial by minimizing the amount of traffic needed to be forwarded to servers:

$$\min(\sum_{c \in C} \sum_{n_i \in V} b_i^c \cdot y_i^c). \tag{8}$$

Network providers need to consider deployment costs to increase profits. Thus, the second objective (#2) is to minimize deployment costs:

$$\min(\epsilon_{sw} \cdot \sum_{sw \in P} z_{sw} + \epsilon_c \cdot \sum_{c \in C} z_c), \tag{9}$$

where ϵ_{sw} and ϵ_c refer to the deployment cost for the switch and the server.

NP hardness. The above problem is NP-hard. The NP-hardness can be proven by considering a special case of the problem. If we nullify the control plane resources, the problem becomes a deployment problem in PDP, which has been shown to be an NP-hard problem [11], [12], [36].

VI. Optimization algorithms

Searching space of the problem can be computed to the exponential of the number of CFG nodes to the sum of total stages and servers, i.e., $(|S|+|C|)^{|V|}$ where $|S|$, $|C|$, and $|V|$ are the size of total stages, servers, and CFG nodes. However, searching space explodes on a large scale. For example, $|S|$ equals the multiplication of the number of available switches and stages per switch (e.g., 12 stages in Intel Tofino 1), and $|V|$ equals the multiplication of the number of applications and nodes per application (e.g., no more than 10 nodes). An ILP solver (e.g., Gurobi) can not find the optimal solution to the problem on a large scale within an acceptable time. Thus, *Carlo* proposes two algorithms to trade off between performance and execution time.

A. Node-ranking heuristic search (NRHS) algorithm

A heuristic approach is proposed in Algorithm 1. The main idea focuses on prioritizing nodes of CFG and deploying them in the hardware pipeline and servers. First, it ranks the nodes of a given CFG based on different attributes (e.g., required resources and dependency relationship), and then incrementally searches and places each node while respecting resource constraints.

Node ranking. Given the placement request of a CFG, its nodes can be evaluated one by one using their three types of variables as indicators (i.e., requirements of the pipeline and CPU resources). Sorting to determine which node can be placed earlier is important since it can affect the quality

Algorithm 1: Node-ranking heuristic search

Input: G, the given CFG $G = \{V, E\}$; D, the dependency matrix of G; *Topo*, the given topology; *Obj*, the objective of search
Output: *rel*, the deployment result

1 V.*Normalization*(G,D)
2 V.*Ranking*(*Obj*)
3 **if** *Obj is MinCost* **then**
4 **for** *each* $n_i \in V$ **do**
5 **if** $n_i \in V_{cp}$ *and SearchInCP(*n_i*, Topo)* **then**
6 *rel.add*(*SearchInCP*(n_i, *Topo*))
7 **else**
8 *rel.add*(*SearchInDP*(n_i, *Topo*))
9 **else**
10 **for** *each* $n_i \in V$ **do**
11 **if** *SearchInDP(*n_i*, Topo)* **then**
12 *rel.add*(*SearchInDP*(n_i, *Topo*))
13 **else**
14 *rel.add*(*SearchInCP*(n_i, *Topo*))

of placement [12]. For example, nodes with large resource requirements could be placed earlier to avoid scarce remaining available resources. Therefore, we use min-max normalization to rescale each indicator in the range $[0, 1]$:

$$\kappa_i^k = \frac{\lambda_i^k - \lambda_{min}^k}{\lambda_{max}^k - \lambda_{min}^k}, \tag{10}$$

where λ_i^k is the indicator of type k of node n_i. Afterward, different optimization objectives not only determine how to design placement policies but also decide distinct metrics that affect the placement. The total of weighted metrics ε is used to sort the nodes from large to small (line 1∼2):

$$\forall n_i \in V : \varepsilon_i = \sum_{k \in Metrics} \omega^k \cdot \kappa_i^k, \tag{11}$$

where ω is the metric weight varying in deployment objectives.

Incremental placement. Next, the algorithm applies policies of incremental placement to place sorting nodes in sequence according to different objectives. With the objective of minimizing cost, the algorithm prefers to place more nodes in V_{cp} as possible in the control plane, and with the other objective, on the contrary, the algorithm priorities to place nodes in the hardware pipeline to maximum throughput (line 3∼14 of Algorithm 1). Function *SearchInCP()* searches available resources in the server whose linking switch has been used and allocates for placing node n_i. If it succeeds, returns the server ID; otherwise, it returns false. Function *SearchInDP()* searches the switch resources for placement of n_i.

Time complexity of this algorithm is $\mathcal{O}(|V| \cdot (|S| + |C|))$, where $|V|$, $|S|$, and $|C|$ refer to the number of CFG nodes, stages, and servers in the topology. In the worst case, each node must check the resources of all stages and servers for placement.

B. Policy gradient (PG) algorithm

Heuristics may find sub-optimal results due to its low complexity designs. In contrast, after a period of offline training, the policy gradient algorithm can automatically study deployment policies and generate near-optimal solutions in an acceptable time.

1) System design: We build a reinforcement learning (RL) architecture where the policy gradient algorithm is implemented in an RL agent to study deployment strategies via interaction with the RL environment. The environment is built to completely emulate the deployment process in both the data and control plane. At each time step $t \in T$, the agent obtains the current deployment state and selects an action a (i.e., a placing position in switches or servers) for deploying a CFG node. A reward r_t is obtained after executing the action and the state is transferred to the next state. The objective of learning is to maximize the estimation of discounted accumulative rewards $\mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t]$, where γ is the discount factor.

State representation. The state is encoded by a feature vector that includes resources in cross-plane and placement conditions of the given CFG. First, remaining resources should be identified, such as bandwidth and CPU resources in the control plane, and storage and computation resources in the data plane. Second, placement details of the given CFG are recorded, such as attributes of the incoming placing node.

Action space. An action is defined as placing a node in a certain position (i.e., a stage or a server) with resource allocation. A node placement would allocate all resources it needs in the placing position; however, if the remaining resources in that position are fewer than its needs, it would be placed in another position at the next time step until it allocates sufficient resources. The agent samples one action from the action vector each whose index is the placement position and value is the possibility of sampling.

Reward shaping. To accelerate the convergence of deployment policies learning, we implement intermediate rewards (e.g., positive and negative rewards) at each step to encourage the agent to study for higher accumulated rewards. For example, a positive reward is obtained when a node is placed successfully. On the contrary, a negative reward would be set when a new switch or server is used for minimizing deployment costs. The value of intermediate rewards generated by each action will be accumulated into the overall reward. Thus, the agent can optimize objectives by adjusting rewards. For example, we set negative rewards for deployment in new servers or switches to minimize deployment costs.

Search speedup. Due to strict constraints in the switch hardware and server resources, the deployment is easy to terminate caused by constraint violation. This leads to the ineffectiveness of policy study. To solve the problem, we design a masking mechanism to cover those invalid actions [37]. First, invalid actions can happen in deployment locations whose resources are exhausted (e.g., resources of a stage have been

Algorithm 2: Policy gradient training algorithm

Input: G, *Topo*, the given CFG and topology

1 g, *topo* $\leftarrow$ *reset*(G, *Topo*)
2 Initialize: θ_π, $\theta_\mathbb{V}$, parameters of Actor and Critic
3 **for** *epoch in Epochs* **do**
4 *clear*(*Buffer*)
5 **while** *Epoch_flag* $\neq$ *True* **do**
6 $n \leftarrow$ *Get_node*(g)
7 *Obs* $\leftarrow$ *Get_Observation*(g,*topo*)
8 $\log p \leftarrow \pi(a|Obs, n; \theta_\pi)$
9 $\log p \leftarrow$ *mask*($\log p$,*topo*,n)
10 $a \leftarrow \log p$.*sample*()
11 $v \leftarrow \mathbb{V}$(*topo*, n;$\theta_\mathbb{V}$)
12 $g, topo, reward \leftarrow$ *Perform_action*(g,*topo*,a)
13 *Buffer*.*add*($\log p$, a, v, *reward*, *topo*, n)
14 **if** *Trajectory_flag* == *True* **then**
15 g, *topo* $\leftarrow$ *reset*(G, *topo*)
16 Compute *advantage estimates* and *rewards-to-go*
17 $d\theta_\pi \leftarrow$ *Actor_loss*(*Buffer*, *advantage estimates*)
18 $d\theta_\mathbb{V} \leftarrow$ *Critic_loss*(*Buffer*, *rewards-to-go*)
19 Update θ_π using $d\theta_\pi$, $\theta_\mathbb{V}$ using $d\theta_\mathbb{V}$

used up). Second, invalid actions will lead to dependency violations between two nodes. We use a mask vector to set the invalid position in the action vector to the negative *inf* value. Thus, the agent can not sample those invalid positions in the action vector.

2) Policy gradient algorithm: The training algorithm (Algorithm 2) builds two neural networks, where Actor network is responsible for generating actions and Critic network evaluates the current environment. The algorithm trains the agent in two steps for each epoch. In the first step, the agent consistently interacts with the environment (line 4∼15) and saves historical deployment procedure (i.e., trajectory) in the *Buffer*. Specifically, the agent first selects a CFG node and samples an action from Actor network (line 6∼10), and then evaluates the current deployment in Critic network and executes the sampling action (line 11∼12).

In the second phase, RL agent updates parameters in Actor and Critic networks using policy gradient loss computing in the *Buffer*. In Actor network, the gradient loss can be calculated by the mean error between GAE-Lambda advantage estimate [38] and the logit of the corresponding sampled action (i.e., *log p* across the epoch). In Critic network, the mean square error between the reward-to-go and the output of Critic network across the epoch, where the reward-to-go is computed by utilizing the discount factor to intermediate rewards.

VII. Implementation

A testbed prototype of *Carlo* is implemented in Python and targets Intel Tofino ASIC switches. We use P4C [39] to analyze INC applications for generating CFGs and write their corresponding P4 codes in $P4_{16}$ [4]. DPDK [40] is used for achieving high throughput of packet process on servers.

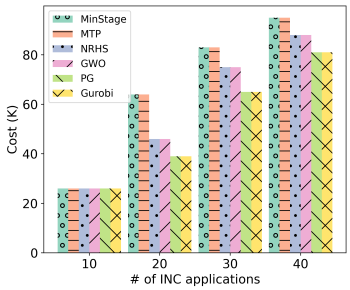


Fig. 3: Comparison of deployment cost.

In the data plane, an *appID* is attached to packets for identifying which application should process them. In each switch, we apply an additional MAT with a very small fraction of resources to manage the application that needs to forward packets to the control plane for processing. Metadata is used to save intermediate states when an application is deployed across multiple switches, which only consumes small additional resources in headers [36].

Dynamic deployment. When a new INC application request arrives, *Carlo* searches switches and servers with available resources for deployment. This allows to minimize disruptions to the current deployment.

INC applications are used as listed in Section IV-B. *Carlo* uses the similar method proposed in CLIP [9] to partition the code of an application in terms of logic modules, which are written in P4 language for the Tofino pipeline and in C language for DPDK implementation. We set reasonable parameters and table entries in traffic modes of applications IV-B. For example, ranging from 0.1 to 0.5 for sample rate θ and miss rate $1 - \hat{\phi}$ are set, and entries of anomaly flow in the public dataset are added in the MAT.

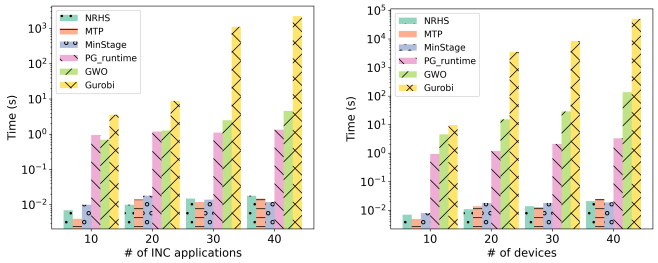
Optimization. We use Gurobi [41] as the optimization solver to find the optimal solution for our ILP model. An NVIDIA Tesla V100 32G GPU is used to train the policy networks in PG algorithm. *Carlo* implements the vanilla policy gradient in PG algorithm based on framework SpinningUp [42]. To evaluate the PG algorithm, different sizes of MLP (Multi-layer Perceptron) are implemented as Actor and Critic networks. In the training phase, we set 1024 to epoch number, 0.99 discount factor, and 0.97 GAE lambda.

A GWO (Grey Wolf Optimizer) is used to automatically generate deployment solutions without any empirical strategies. By balancing exploration and exploitation, the algorithm explores the search space effectively and converges toward an optimal or near-optimal solution. We transform our ILP (Section V) to encode the problem in GWO.

VIII. Evaluation

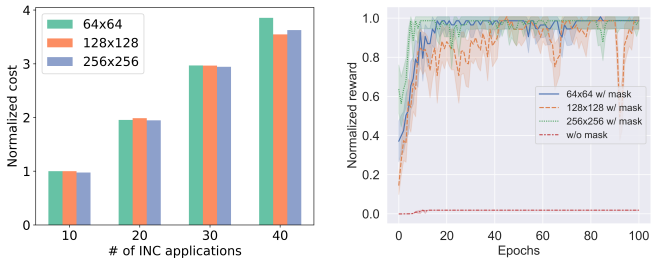
A. Experimental settings

Testbed. Our testbed is built with two programmable hardware switches (Flnet S9180-32X with Intel Tofino ASIC). In the data plane, devices are connected linearly, where the two switches are connected in the middle of the topology and each



(a) Time w.r.t. # of INC applications. (b) Time w.r.t. # of devices.

Fig. 4: Comparison of execution time.



(a) Impact on training cost. (b) Impact on training reward.

Fig. 5: Performance of PG algorithm.

of them connects to a server. Additional servers are used as senders and a receiver. Senders use *iperf* to generate flows with 250-byte packets processing in monitoring applications (e.g., DDoS detection [20] and INT [26]) and compute tasks (e.g., AES [18] and ChaCha [28]), and *Tcpreplay* [43] to replay the real dataset CIC-IDS2017 [44] processing in In-cache [2], ML applications [6] etc., generated a total of 40Gbps traffic rate. Each server is equipped with a dual Intel Silver 4210R CPU with 10 cores and a Mellanox ConnectX-3 40GbE Network Interface Card, and runs DPDK for packet processing. All servers are connected by the switch via 40Gbps fiber (40GBASE-SR4) links.

Comparison schemes. MTP [10] is a framework for measurement task deployment in PDP. MinStage [12] proposed different heuristics to minimize deployment costs in the hardware pipeline. Without the consideration of cross-plane deployment, both methods can only deploy in the data plane.

B. Experimental Results

(Exp#1) Deployment cost of *Carlo*. We first evaluated the performance of *Carlo*'s algorithms with the objective of minimizing cost and comparing them with other methods. Switch and server costs are set according to Table I. We randomly selected INC applications from 10 to 40 for deployments, each of whose input throughput ranges from 1∼40Gbps. As shown in Fig. 3, the PG algorithm achieved the same performance as Gurobi, indicating that it can be a near-optimal solution. On the other hand, the GWO and NRHS heuristic algorithms exhibited performance bottlenecks as the number of INC applications increases. However, MTP and MinStage only deployed in the data plane, requiring more resources to meet the resource requirements, especially when deploying complicated applications. In conclusion, *Carlo* decreases by an average of 18.7% compared with the two methods.

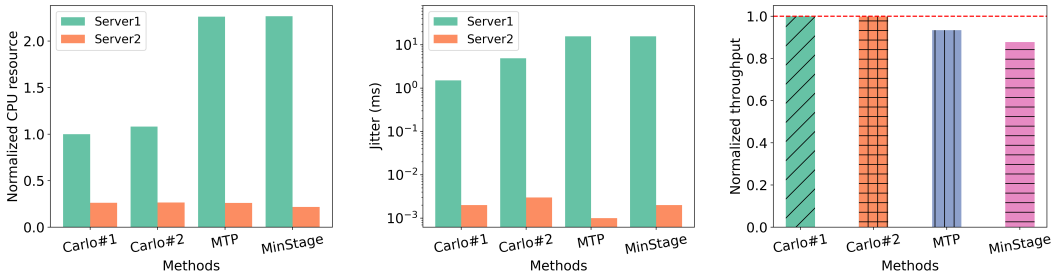
(Exp#2) Execution time of *Carlo*. We measured the execution time of different algorithms of *Carlo* (shown in Figure 4). The number of INC applications and given devices range from 10 to 40. As expected, Gurobi searched for the optimal solution but faced scalability issues, the search time exponentially increasing as the number of devices grows. After a period of offline training, the PG algorithm achieved approximate solutions within a short runtime. For GWO, we fixed the number of generations and search iterations, but it also faced scalability challenges. Finally, heuristic algorithms (i.e., NRHS, MTP, and MinStage) can obtain feasible solutions in the shortest time due to their smaller search complexity.

(Exp#3) Performance of PG algorithm. We evaluated the effects of different MLP sizes on deployment cost with objective #2. In Fig. 5(a), the smallest neural network achieved a high cost in deployment requests, as it was unable to model more complex relationships when the number of applications increased. In comparison, *Carlo* can find better solutions with the other two sizes of MLP.

We observed rewards in different sizes of MLP with and without the masking mechanism. Results in Fig. 5(b) show that MLP with the largest hidden size has a faster convergence speed with respect to the number of epochs because it can be trained more efficiently to capture the more complicated relationships. In contrast, without the masking mechanism, the agent cannot study policy effectively because the deployment is too complicated and failure always happens.

(Exp#4) Performance in testbed experiments. We conducted deployment experiments in our testbed for comparison with the other two methods (shown in Fig. 6), and *Carlo* was configured with two objectives: maximizing throughput (#1) and minimizing cost (#2). Since *Carlo* with objective #1 tended to deploy more INC applications on the data plane, resulting in less CPU resource usage than objective #2. In comparison, without the cross-plane collaboration deployment strategy, MTP and MinStage failed to implement all applications in the data plane. For fairness, we deployed the remaining applications on the server for the two methods. However, packet loss and jitter increase are observed when their deployments exceed the capacity of servers, i.e., about a 3∼10x jitter increase in MTP and MinStage, as shown in Fig. 6(b). Eventually, throughput in *Carlo* was not affected, whereas both MTP and MinStage experienced a throughput decrease (the dashed line represents the send rate Fig. 6(c)). In experiments, we observed the jitter increase in *Carlo* with #1 and #2. This result is in a reasonable range since the DPDK batch processing method requires packets to wait for the completion of high CPU cycle tasks before returning collectively. In comparison, unreasonable deployments of MTP and MinStage obtain higher increases in jitter and packet loss.

(Exp#5) Performance of decision trees in testbed experiments. In this experiment, we implemented a decision



(a) CPU resource (b) Jitter (c) Throughput

Fig. 6: Performance in testbed experiments.

TABLE IV: Performance of decision trees in testbed experiments.

Methods \ Attacks	Precision				Recall				F1 score			
	Infiltration	BF	SQL	XSS	Infiltration	BF	SQL	XSS	Infiltration	BF	SQL	XSS
Carlo	0.783	0.824	1.0	0.676	0.805	0.743	0.285	0.680	0.794	0.781	0.444	0.678
MTP	0.757	0.816	1.0	0.676	0.694	0.706	0.285	0.679	0.724	0.757	0.444	0.677
MinStage	0.75	0.814	1.0	0.666	0.66	0.697	0.142	0.650	0.705	0.751	0.25	0.658

tree algorithm and replayed the real dataset to evaluate how packet loss affects accuracy. In the switches, we collected flow features (e.g., packet timestamp, packet length, and flag counter [6]) and used Digest [4] to send them to the control plane. The dataset includes four attack types: infiltration, web attack brute force (BF), SQL injection (SQL), and cross-site scripting (XSS). Three metrics are used to evaluate its performance, including precision, recall, and F1 score. Due to packet loss, both MTP and MinStage experienced a decrease in precision (as shown in Table IV). The prediction precision of all anomalous flows decreased slightly, which can be attributed to the small distribution of anomalous flows in the dataset. However, the recall and F1 score for SQL attacks suddenly dropped in MinStage (e.g., 14.2% in recall and 25% in F1 score), due to packet loss occurring during the collection of these flows. Therefore, a reasonable deployment of *Carlo* can ensure the performance of INC applications.

IX. Related Works

Deployment in PDP. Previous work has proposed frameworks to address the network-wide deployment for PDP [10]–[13], [36]. They utilized various graph tools to abstract network functions (e.g., table dependency graph [12], [36] and CFG [11]) and proposed models with resource constraints to optimize PDP deployment. In comparison, *Carlo* utilizes resources in both the control and data plane to support a wide of INC applications and optimize network-wide resource allocation.

Collaboration with the control plane. Current research has discussed cross-plane collaboration merits and proposed compilers to generate code in different planes [8], [9]. In comparison, *Carlo* not only demonstrates that cross-plane collaboration can effectively optimize the deployment of multiple INC applications, but also ensures the effectiveness and high-performance deployment by analyzing various INC applications. Thus, *Carlo* complements the above works to automatically provide optimization strategies for deployments.

Offloading NFV on PDP aims to offload specific *simple* logic on the data plane for conventional network function acceleration [14], [15]. However, the current INC applications have chosen to deploy complex logic on the data plane [6], [7], [18], which comes at the cost of consuming more precious resources, causing increased deployment costs. *Carlo* employs a collaborative strategy to address this issue in the deployment of multiple applications. *Carlo* also enriches NFV offloading deployment by supporting a wide of INC applications.

X. Conclusion

Carlo optimizes the deployment problem of multiple INC applications in cross-plane. It first analyzes deployment resources of a wide of INC applications, then establishes an ILP model that restricts resources on cross-plane, and finally balances time and algorithm performance through proposed algorithms to compute solutions. Extensive experiments indicate that *Carlo* effectively utilizes resources in different planes to solve the deployment optimization problems of multiple INC applications.

Acknowledgments

This work is partially supported by National Natural Science Foundation of China (NSFC) under Grant 62172189 and 62272050; the Guangdong Key Lab of AI and Multi-modal Data Processing, United International College (UIC) under Grant 2020KSYS007; the Guangdong Basic and Applied Basic Research Foundation under Grant No. 2021B1515120048; the Innovate UK grants 47198 and 10040850; the Outstanding Innovative Talents Cultivation Funded Programs for Doctoral Students of Jinan University; and the Interdisciplinary Intelligence SuperComputer Center of Beijing Normal University (Zhuhai).

REFERENCES

[1] S. Kianpisheh and T. Taleb, "A survey on in-network computing: Programmable data plane and technology specific applications," *IEEE Communications Surveys & Tutorials*, 2022.

[2] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of Symposium on Operating Systems Principles*, 2017, pp. 121–136.

[3] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "pHeavy: Predicting heavy flows in the programmable data plane," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4353–4364, 2021.

[4] "Intel Tofino switch ASIC," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, accessed on: Jul. 30, 2023.

[5] B. Steinert, M. Häberle, J.-O. Nick, D. Farinacci, and M. Menth, "P4-LISP: A P4-based high-performance router for the locator/identifier separation protocol," 2023.

[6] C. Zheng, M. Zang, X. Hong, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Automating in-network machine learning," *arXiv preprint arXiv:2205.08824*, 2022.

[7] M. Jose, K. Lazri, J. François, and O. Festor, "Stateful InREC: Stateful in-network real number computation with recursive functions," *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 830–845, 2022.

[8] F. Pereira, G. Matos, H. Sadok, D. Kim, R. Martins, J. Sherry, F. M. Ramos, and L. Pedrosa, "Automatic generation of network function accelerators using component-based synthesis," in *Proceedings of the Symposium on SDN Research*, 2022, pp. 89–97.

[9] T. Xu, X. Wang, C. Tian, Y. Xiong, Y. Lin, and B. Ye, "CLIP: Accelerating features deployment for programmable switch," in *IEEE International Conference on Communications*. IEEE, 2023.

[10] X. Chen, H. Liu, D. Zhang, Q. Huang, H. Zhou, C. Wu, and Q. Yang, "Eliminating control plane overload via measurement task placement," *IEEE/ACM Transactions on Networking*, 2022.

[11] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "Compiling service function chains via fine-grained composition in the programmable data plane," *IEEE Transactions on Services Computing*, 2023.

[12] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th USENIX Symposium on Networked Systems Design and Implementation*, 2015, pp. 103–115.

[13] W. Xu, Z. Zhang, Y. Feng, H. Song, Z. Chen, W. Wu, G. Liu, Y. Zhang, S. Liu, Z. Tian *et al.*, "ClickINC: In-network computing as a service in heterogeneous programmable data-center networks," *arXiv preprint arXiv:2307.11359*, 2023.

[14] X. Chen, Q. Huang, P. Wang, Z. Meng, H. Liu, Y. Chen, D. Zhang, H. Zhou, B. Zhou, and C. Wu, "Lightnf: Simplifying network function offloading in programmable networks," in *IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 2021, pp. 1–10.

[15] J. Ma, S. Xie, and J. Zhao, "Flexible offloading of service function chains to programmable switches," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1198–1211, 2022.

[16] R. MacDavid, X. Chen, and J. Rexford, "Scalable real-time bandwidth fairness in switches," in *IEEE INFOCOM*, 2023.

[17] Â. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time DDoS attack detection to programmable data planes," in *IFIP/IEEE Symposium on Integrated Network and Service Management*. IEEE, 2019, pp. 19–27.

[18] X. Chen, "Implementing AES encryption on programmable switches via scrambled lookup tables," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020, pp. 8–14.

[19] "iPerf - the ultimate speed test tool for TCP, UDP and SCTP," https://iperf.fr/, accessed on: Jul. 30, 2023.

[20] F. Musumeci, V. Ionata, F. Paolucci, F. Cugini, and M. Tornatore, "Machine-learning-assisted DDoS attack detection with P4 language," in *IEEE International Conference on Communications*. IEEE, 2020, pp. 1–6.

[21] "P4language," https://github.com/p4lang, accessed on: Jul. 30, 2023.

[22] J. Hyun, N. Van Tu, and J. W.-K. Hong, "Towards knowledge-defined networking using in-band network telemetry," in *IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–7.

[23] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric DDoS attacks with programmable switches," in *30th USENIX Security Symposium*, 2021, pp. 3829–3846.

[24] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow," in *USENIX Annual Technical Conference*, 2018, pp. 823–835.

[25] E. Cidon, S. Choi, S. Katti, and N. McKeown, "Appswitch: Application-layer load balancing within a software switch," in *Proceedings of the First Asia-Pacific Workshop on Networking*, 2017, pp. 64–70.

[26] N. Van Tu, J. Hyun, G. Y. Kim, J.-H. Yoo, and J. W.-K. Hong, "INT-collector: A high-performance collector for in-band network telemetry," in *14th International Conference on Network and Service Management*. IEEE, 2018, pp. 10–18.

[27] S. Tang, D. Li, B. Niu, J. Peng, and Z. Zhu, "Sel-INT: A runtime-programmable selective in-band network telemetry system," *IEEE transactions on network and service management*, vol. 17, no. 2, pp. 708–721, 2019.

[28] Y. Yoshinaka, J. Takemasa, Y. Koizumi, and T. Hasegawa, "On implementing ChaCha on a programmable switch," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 15–18.

[29] S. Patel, R. Atsatsang, K. M. Tichauer, M. H. Wang, J. B. Kowalkowski, and N. Sultana, "In-network fractional calculations using P4 for scientific computing workloads," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 33–38.

[30] V. S. Thapeta, K. Shinde, M. Malekpourshahraki, D. Grassi, B. Vamanan, and B. E. Stephens, "Nimble: Scalable TCP-friendly programmable in-network rate-limiting," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research*, 2021, pp. 27–40.

[31] D. Ding, M. Savi, and D. Siracusa, "Estimating logarithmic and exponential functions to track network traffic entropy in P4," in *IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[32] Y.-K. Lai, S.-Y. Yu, I.-S. Chan, B.-H. Huang, C.-H. Chang, J. H. Chen, and J. Mambretti, "Sketch-based entropy estimation: a tabular interpolation approach using P4," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 57–60.

[33] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, "An efficient design of intelligent network data plane," in *32nd USENIX Security Symposium*, 2023.

[34] "Perf WiKi," https://perf.wiki.kernel.org, accessed on: Jul. 30, 2023.

[35] I. S. Association *et al.*, "IEEE standard for ethernet amendment 3: Physical layer specifications and management parameters for 40 Gb/s and 100 Gb/s operation over fiber optic cables," 2015.

[36] X. Chen, H. Liu, Q. Huang, P. Wang, D. Zhang, H. Zhou, and C. Wu, "Speed: Resource-efficient and high-performance deployment for data plane programs," in *IEEE 28th International Conference on Network Protocols*. IEEE, 2020, pp. 1–12.

[37] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser *et al.*, "Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[38] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[39] "p4lang/p4c," https://github.com/p4lang/p4c, accessed on: Jul. 30, 2023.

[40] "Intel corporation. Data Plane Development Kit." https://www.dpdk.org/, accessed on: Jul. 30, 2023.

[41] "Gurobi - The Fastest Solver," https://www.gurobi.com/, accessed on: Jul. 30, 2023.

[42] "Open AI Spinning Up," https://spinningup.openai.com/en/latest/, accessed on: Jul. 30, 2023.

[43] "Tcpreplay - pcap editing and replaying utilities," https://tcpreplay.appneta.com/, accessed on: Jul. 30, 2023.

[44] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization." *ICISSp*, vol. 1, pp. 108–116, 2018.