Accelerating Network Features Deployment With Heterogeneous Platforms

Tingting Xu, *Student Member, IEEE*, Xiaoliang Wang, *Member, IEEE*, Chen Tian, *Senior Member, IEEE*, Yun Xiong, Baoliu Ye, *Member, IEEE*, Sanglu Lu, *Member, IEEE, ACM*, and Cam-Tu Nguyen

***Abstract*— Enhancing the networking system with appropriate functions is a longstanding goal. Unfortunately, in today's large-scale high-speed data centers, the feature velocity of network functions is slow because it is hard to verify the function in realistic scenarios. Recent advances in programmable switching ASICs have enabled the network data plane to move beyond its traditional role of packet forwarding. However, the current compromise between performance and flexibility results in limitations such as restricted memory/computation resources and programmable models. These limitations make it challenging for programmable switches to offer more features and to be deployed in large-scale production environments. In response, we present CLIP, a framework that works in collaboration with programmable devices and commodity servers to enhance the validation and deployment velocity of features. CLIP defines a cross-platform function definition framework and provides a set of tools to reduce the complexity of manually writing cross-platform programs. We propose an automatic traffic placement and scaling mechanism to coordinate packet processing performance across heterogeneous devices. Compared with software-based Network Functions (NFs), CLIP achieves a throughput ranging from 1.36× to 16.06× under different realistic traffic loads. Through the development and deployment of three self-defined functions within a realistic testbed, we demonstrate the feasibility and efficiency of CLIP.**

***Index Terms*— Programmable network, network functions deployment, hardware–software co-design.**

I. Introduction

CLOUD networks are complex infrastructures that serve a large number of tenants and a variety of applications. The demands on cloud networks are continuously changing, requiring an adaptable solution to achieve fast feature velocity. Traditional network devices such as routers and switches cannot keep up with the fast-changing nature of cloud networks due to their inherent inflexibility. Specifically, these devices are based on vendor-fixed-function chips, and their data plane algorithms are typically impervious to modification [1]. Relying on updates from device vendors, which typically follow years-long release cycles, has proven insufficient to keep pace with the evolving requirements of modern networks.

Programmable data planes enable users to implement their own data plane algorithms, which can be applied through self-defined network control [1], [2]. The community has developed programmable facilities that include software-based development frameworks [3], [4], [5], dedicated hardware such as network processors [6] and programmable switching chips [7], [8], offering significant flexibility for network customization [1], [2], [6], [7], [8]. Although programmable switches strike a balance between adaptability and performance, they do not fully bridge the gap between increasing customer demands and limited hardware capabilities. For example, the limited memory on the chip, around O (10M), is cramped for state-heavy network functions [9], [10]. The primitive computation makes the in-network computation have to scarify the precision [11], [12], [13]. In addition, a limited number of stages and the ability to access the memory only O (1) times [14], make it harder to develop more advanced applications.

Therefore, in practice, we need a solution for quickly addressing users' requirements, rapid prototyping of new protocols, and verifying the functions in production networks, which guides the design of the network chip of the next generation. The fundamental concept to address the requirements involves a collaborative approach between programmable switches and commodity servers, which achieves a synergy of flexibility and performance. However, developing a seamlessly integrated platform that harmonizes switch ASICs and servers is a formidable challenge, primarily due to the divergent design of these devices. Software-based network processing offers high flexibility in terms of expressiveness, while programmable hardware is constrained by existing models. In terms of performance, programmable switches can achieve Tbps throughput with latencies of hundreds of nanoseconds, whereas the capacity of commodity servers is limited to 100 Gbps throughput with latencies of tens of microseconds. This challenge requires us to strike a balance between the strengths and limitations of each device, with the goal of building a platform that supports rapid feature

Received 13 December 2023; revised 14 September 2024; accepted 25 October 2024; approved by IEEE TRANSACTIONS ON NETWORKING Editor T. Qiu. Date of publication 12 November 2024; date of current version 14 February 2025. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB2702803, in part by the National Natural Science Foundation of China under Grant 62172204 and Grant 61832005, in part by the Key Research and Development Program of Jiangsu Province under Grant BE2020001-3, and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization. *(Corresponding author: Xiaoliang Wang.)*

Tingting Xu, Xiaoliang Wang, Chen Tian, Baoliu Ye, Sanglu Lu, and Cam-Tu Nguyen are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: xutingting@smail.nju.edu.cn; waxili@nju.edu.cn; tianchen@nju.edu.cn; yebl@nju.edu.cn; sanglu@nju.edu.cn; ncamtu@nju.edu.cn).

Yun Xiong is with Huawei, Nanjing 210012, China (e-mail: xiongyun1@huawei.com).

Digital Object Identifier 10.1109/TNET.2024.3491840

development while navigating the trade-offs between expressiveness and performance.

In this paper, we propose CLIP, a framework inspired by remote procedure call (RPC) that enables remote processing through the switch data plane (§III). To simplify programming, we design a top framework that incorporates a pre-processing module on the switch, an on-loaded request handler on servers, and a post-processing module on the switch (§IV-A). This design facilitates the creation of user-defined network features with the collaboration of heterogeneous devices. Our suite of tools, which includes a compiler and a controller (§IV-B), generates executable files for the switch and the remote processors for the user-defined program. Using this cross-platform framework empowers remote processors and memory to extend programmable switch capabilities, accelerating the deployment of functions (§IV-C). We employ multiple servers to balance traffic and maintain high throughput across concurrent high-speed channels/links. Furthermore, a load balancer based on programmable switches reduces the server workload, ensuring scalability (§V).

We implement CLIP and evaluate it in the testbed consisting of a Tofino-based programmable switch and servers (§VII). We demonstrate the use of CLIP through the practical requirements in the production network, including the network function of state-heavy NAT, the forwarding module in cloud gateway FIB, and the new feature of overlay network measurement through active TCP retransmission detection. For example, the measurement of the overlay network requires identifying the retransmission packets of specified flows to determine the root cause of network delay jitters. The detection of retransmission packets is not supported by the current programmable switches due to the limited size of buffers and the difficulty of maintaining the state of TCP connections. By using CLIP we successfully deploy this feature in the experimental environment which has demonstrated and verified the effectiveness of designs. Meanwhile, compared to software-based Network Functions (NFs), CLIP improves throughput by 1.36× to 16.06× under realistic workloads.

The contributions of this paper are summarized as follows:

- CLIP addresses the challenge of overcoming the sluggishness of hardware function provision at the data plane. The integration of programmable switches with a cluster of commodity servers to rapidly prototype new protocols, and verify the functions in production networks. It creates a versatile and programmable data plane, adept at meeting the evolving requirements of modern network infrastructures, and guides the design of the network chips.
- The introduction of a framework by CLIP facilitates the efficient partitioning of network functions between switches and servers. This framework streamlines the deployment of new features, offering user-friendly programming interfaces for easy customization of the data plane. Additionally, CLIP demonstrates scalability to dynamic traffic by intelligently balancing the load between servers and offloading resource-intensive tasks to the switch, ensuring adaptability to evolving network demands.
- CLIP achieves fast feature velocity to address the urgent demands of customers in production networks. The system introduces a pioneering approach to prototype and verify new network functions in the realistic system, offering valuable insights for cloud network service providers to determine the effectiveness of new functions.

A short conference version [15] of this paper appeared in INFOCOM 2023. In this extended version, we expand our focus on enhancing the capabilities of programmable switches. One major addition is the usage of remote handlers on servers to mitigate the impact of the limited computational power of switches and the memory capacity of servers. Specifically, we introduce key designs: (i) optimizing variable placement by conducting code optimization, (ii) extending switch memory via table mirroring, (iii) enhancing computational capabilities by supporting more complex operands, such as floating-point values, and (iv) employing stateful variables to maintain state across multiple packets. Additionally, we refine the flow placement strategy to address the critical bandwidth limitations and extend this approach to the longest prefix match (LPM) type. To further validate the efficiency of CLIP, we implement an additional cloud network application, the Multi-tenancy Forwarding Information Base (FIB). The effectiveness of CLIP's flow placement strategies and case studies is modeled and thoroughly evaluated.

II. Motivation

We explain the new feature demand of network functions in modern data centers and discuss why the current software and hardware design of programmable data planes are not sufficient to meet the demands. We then highlight the motivation which centers on enhancing the ability for feature deployment.

A. Demand for New Network Features

Cloud providers have progressively adopted new network functions or new features to support effective networking operation, continuous performance optimization, and the specific demands of users that are emerging. Software-based solutions are applied because of the inherent programmability. Take the disaggregated software routers [16], [17] applied in cloud gateways as an example, the fundamental architecture of which is shown in Figure 1a, it uses a software-based data plane such as DPDK [3], VPP [18], Click [5], FastClick [19] and achieves a feature velocity of a few weeks in the product network.

Nevertheless, in comparison with commodity networking devices, like switches, general-purpose CPUs are an order of magnitude slower and lack the performance-boosting advantages of pipelining and parallelism. Besides, achieving the same throughput as commodity switches, we usually need a cluster of servers which leads to a high cost [9].

B. The Rise of Programmable Switch

The programmable switch consists of a software-based control plane and a dedicated hardware data plane. The data plane is programmable, allowing developers to customize the packet processing pipeline based on user-defined requirements. Users

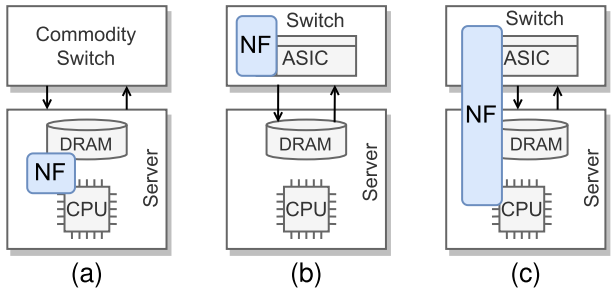


Fig. 1. Network Function Deployment. (a) Software-based NF without switch cooperation. (b) Switch-based NF with server memory expansion. (c) CLIP: switch-based NF with server cooperation.

define data planes using the domain-specific language [2], [20]. The prevalent model for data plane programming is Programming Protocol-Independent Packet Processors (P4), supported by a range of software- and hardware-based target platforms. It allows define the following two programmable functionalities [2].

- **User-defined processing pipeline:** The operation of packet processing is abstracted into a generic pipeline of match-action tables. As the packet traverses this pipeline, it executes corresponding actions when matching the key values. Developers can define the packet processing logic through the match-action tables. Stateful memory can maintain state across packets, such as tables, registers, counters, and meters. The registers can be updated as the packets cross the pipeline. Thus, numerous novel in-network applications leverage the registers to cache data or perform primitive computations at data plane [11], [12], [13], [21], [22].
- **Interaction between data plane and control plane:** The high-speed pipeline for match-action processing is segregated from complex computations, which are handled by the control plane running on general-purpose processors like CPUs. Communication from the data plane (ASIC) to the control plane (CPU) is facilitated either through the digest mechanism or packet redirection. In the former, the data plane reports a concise amount of information to the control plane, whereas the latter involves redirecting the entire packet to the control plane. The control plane gives instructions to guide the packet processing at the data plane.

C. Limitation of Programmable Switch

Compared to traditional fixed-function switches, programmable switches offer flexible packet processing in the data plane, resulting in shorter development cycles and reducing costs for the implementation of new functions. However, deploying network functions on programmable switches in large-scale production environments presents challenges [9], [10], [14]. Key concerns include limitations in primitive computation, limited memory, and a narrow control channel.

- **Primitive Computation.** The computational limitations pose constraints on more advanced applications. While programmable networks facilitate a variety of innovative applications (e.g., stateful load balancing [13], in-network caching [21], in-network aggregation [11], [12]), enhancing performance and reducing costs by offloading specific actions from servers, there are limitations in action execution. However, the action execution supports only a small set of simple ALU (arithmetic and logic unit) operations on *integers but not floating-point values*. The supported operations are addition, subtraction, bitwise operations and comparison, which is sufficient for packet processing, but not enough for more novelty applications. For instance, the machine learning acceleration [11], [12] leverages on-chip memory to aggregate (perform addition on) floating-point gradients but sacrifices precision.
  Moreover, the utilization of a register necessitates strict adherence to the template of computation, stored value, and argument width, presenting challenges in designing applications.
- **Limited Memory.** The limitation of on-chip memory presents challenges in deploying large-scale features. Stateful processing is integral to network systems, enabling applications to store and retrieve data across different packets. The register arrays and match-action tables are organized into physical stages within the pipeline. The total data storage across all stages is constrained, typically ranging from tens to hundreds of megabytes. Additionally, resource utilization is sub-optimal due to placement constraints [14], [23]. In practice, the actual stored data often falls short of the specified capacity. The packet header vector (PHV) serves to convey information from headers and metadata, including temporary values or intermediate results, passing through the pipeline. Although a 200-byte PHV size accommodates traditional protocol processing, it imposes limitations for functions necessitating additional values or operations on the packet contents [14], [24].
- **Narrow Channel.** The programmable model imposes limitations on communication-intensive functions. To ensure high-speed processing of data plane, match-action operations are extracted from network processing, with relatively complex processes left to the control plane. Effective communication between the data plane and the control plane is vital for function deployment. Taking network measurement as an example, more precise network status analysis is possible with increased data collected and reported by the data plane. However, the channel between ASIC and CPU, designed for occasional control plane traffic (e.g., L2 address learning), has a fixed bandwidth lower than the ASIC's per-port capacity. This limitation hinders support for higher traffic rates without hardware modification [9]. The frequency of interaction via this narrow channel must be carefully considered when deploying features.

D. Objective

The deployment of software-based network functions, as depicted in Figure 1a, exhibits high flexibility to improve the velocity of features but suffers from high cost [9], [25]. TEA framework [9] stands out as the pioneering solution, implementing NFs through a programmable switch and using RDMA-accessible memory on servers for state storage.

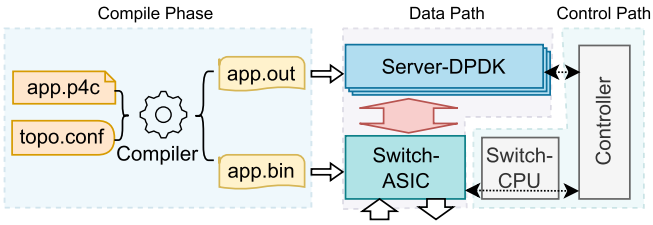


Fig. 2. The CLIP enables the cross-platform program to run at a programmable switch (PS) and several commodity servers (CS) processing planes. The red two-way arrow indicates the request data path.

As illustrated in Figure 1b, TEA extends the memory of the switch by forwarding packets to servers that house the rules needed for packet processing. However, TEA cannot support complex network functions that surpass the capabilities of switches [25]. To address the need for swift feature deployment, our goal is to design a hardware-software co-design packet processing framework that fulfills the following requirements:

- R1 - Expressiveness: Supporting a variety of complex logic for packet processing.
- R2 - Efficient Cooperation: Minimizing the cooperation overhead of heterogeneous processors including programming complexity and communication cost.
- R3 - High Performance: Maintaining high performance through heterogeneous devices. It is crucial to ensure that remote processing does not become the bottleneck, throttling the throughput or significantly increasing latency.

III. CLIP Architecture

To accelerate the deployment of network features, we propose CLIP, a cross-platform system to cooperate with the switch and commodity servers. Through CLIP, the new features can be verified in realistic systems.

A. System Overview

The system consists of a programmable switch (PS) connected to a cluster of commodity servers (CS), which presents a promising avenue to meet the outlined requirements. As depicted in Figure 1c, we offload a portion of the network processing logic to commodity servers, not only to extend memory but also to enhance flexibility and expressiveness (R1). Facilitated by a directly connected data plane, communication between the programmable switch and commodity servers becomes efficient, because it bypasses the narrow control channel (R2). The cluster of commodity servers offers ample memory and adaptable computational resources, narrowing the performance gap between software and dedicated hardware processing, and ultimately improving throughput (R3).

The CLIP architecture, depicted in Figure 2, streamlines the process for cross-platform programming. Users define their network functions in a user-defined program (app.p4c) using the CLIP framework. And a topology configuration (topo.conf) outlines the available physical devices and interconnections. The CLIP compiler then generates executable files for the switch ASIC (app.bin) and remote processors (app.out) based on the user-defined program and topology configuration, respectively. Installation involves deploying app.bin through the switch control plane and app.out to the servers. The controller initiates device setup, establishing the runtime environment, and monitoring device status. The controller manages the switch table content using the remote P4Runtime API [26].

To minimize the overhead of remote processing, CLIP enables the switch data plane to independently request processing, which functions similar to RPC, without the involvement of the control plane. While this procedure is well-defined, manually writing cross-platform program is tedious and error-prone. To simplify programming, the compiler generates essential components, including the control flow, parameter transmission, and the forwarding module to be deployed on the switch. In addition, to prevent the server from throttling the overall performance, we designed a load balancer deployed on the switch to relieve the load on servers and ensure that no traffic stops during scaling.

B. Efficient Communication Design

Efficient resource access is crucial to enable the collaboration of heterogeneous devices in high-speed networks. In this context, a key question arises: How can we access the memory and utilize the computation resources of a server without making any hardware modifications? Existing switch ASICs lack abstractions for inter-program communication mechanisms such as remote procedure call (RPC) [27]. Currently, switch ASICs communicate with the control plane by redirecting packets to the switch CPU or by reporting a digest with several bits of information. Once packets or digests are received, the switch control plane can utilize the traditional RPC mechanism to request remote processing. It can draw support from remote processors by taking existing communication mechanisms while facing the following problems:

- *Throughput bottleneck.* The capacity of packet processing is constrained by the limited bandwidth of the channels between the switch ASIC and the switch CPU, as well as between the switch CPU and the server CPU. The channel between the switch ASIC and the switch CPU, primarily used for control plane traffic like L2 address learning, has a fixed bandwidth of approximately one hundred Gbps. Similarly, the network bandwidth between the switch CPU and the server CPU is also one hundred Gbps. However, the switch ASIC can handle several Tbps of traffic. When traffic requiring remote processing exceeds the channel's capacity, it becomes the bottleneck. Moreover, for packets that need to be processed by remote processors (e.g., the length of requisite data exceeds the PHV capacity), they must be encapsulated into RPC messages since neither the control plane nor the data plane of the switch can store a large number of packets. This transmission of large RPC messages further exacerbates the throughput problem.
- *Unpredictable latency.* RPC messages sent from the control plane must traverse multiple network hops before reaching the server. This is because the control plane and the data

plane of switches and servers are isolated for security purposes. The switch CPU and switch ASIC connect to servers through different links and devices. In contrast to the directly connected data plane, RPC messages generated by the switch CPU must travel through the control plane links. In addition, other control messages that maintain network operations share these same links. As a result, the latency of RPC messages becomes unpredictable due to variations in the control plane's path and network status.

CLIP enables the switch ASIC to directly request remote processing from servers, bypassing narrow control channels and eliminating throughput bottlenecks and reducing the latency associated with traversing multiple control plane hops. Inspired by the RPC mechanism, the request parameters are provided by the switch data plane, delivering the necessary data for the server-side function to perform its operation. After processing the request parameters, the remote procedure returns the response value that contains the result of the operation. In this communication framework, the switch ASIC forwards the original packet carrying the request parameters (i.e., request packet) and receives the modified packet containing the returned value (i.e., response packet). These request and response packets are transmitted through the data plane channel that connects the switch ASIC to the server CPU, completing the entire process without involving the switch control plane.

IV. Network Features in CLIP

In this section, we introduce network features partition and their deployment over CLIP architecture.

A. Network Features Definition Framework

Based on efficient resource access, we need to tackle the challenge of arranging the new feature on heterogeneous devices. This involves addressing various questions, such as how to partition the NF, what parts of NF should run at the remote processor or local ASIC, and how to fully utilize the resource at both programmable switch (PS) and commodity servers (CS). It is hard to build a universal model for diverse NFs [1], [14]. Considering the performance and flexibility of heterogeneous PS and CS frameworks, we have identified two principles: 1) Efficient Packet Match-Action Processing in PS: programmable switches should handle packet match-action processing, enabling fast and precise packet forwarding based on predefined rules. 2) Rich Resource Capacity in CS: commodity servers are equipped with abundant resources in terms of memory and computation capabilities. They are capable of executing complex computations and handling resource-intensive tasks efficiently. Based on these principles, we propose the approach of partitioning the function into three parts, i.e., a *pre-processing* partition, a *request handler* and a *post-processing* partition. The pre-processing partition handles the initial processing of incoming packets, preparing them for further analysis and action. The request handler processes specific complex tasks related to incoming requests, ensuring timely and accurate responses. The post-processing partition deals with the final processing of packets after the required computations have been performed, ensuring correct packet forwarding and delivery. Based on the above partitions, the top framework definition is shown as follows.

```
1  /* Top layer control */
2  control pipeline(inout header hdr,
       inout metadata md){
3    pre_processing  pre; post_processing
       post;
4    request req; response res;
5    bool flight;
6    apply {
7      pre.apply(hdr, flight, req);
8      /* Call the remote procedure */
9      if (flight) remote_handler(req, res
         );
10     post.apply(hdr, res);     }}
```

The control flow for packet processing is defined in the *pipeline* control block, where the packet header (hdr) and intermediate values (md) serve as input and output parameters. The pipeline encompasses three essential components represented by different control blocks. The pre-processing control block executes P4-defined parts at the switch data plane, responsible for pre-processing tasks on incoming packets. Similarly, the post-processing control block runs P4-defined parts in the data plane switch to handle post-processing tasks. To facilitate cross-platform communication for packet processing, two parameter lists named request (req) and response (res) are defined, serving as references for the remote-handler and post-processing blocks, respectively.

Specifically, to address performance limitations and enhance programming flexibility compared to running solely on servers or switches, our proposed solution involves embedding the C language with the P4 language to define cross-platform programs, denoted by the suffix p4c. This approach allows us to capitalize on the strengths of both languages while minimizing their limitations. The remote-handler block exemplifies this approach, as it represents C-defined parts running on general-purpose processors. This enables us to harness the computational power of servers to efficiently handle resource-intensive tasks, which might be less performance-sensitive or inflexible if executed exclusively on switches using P4-defined parts. Furthermore, we introduce the *flight* flag to aid in filtering packets that necessitate remote processing, effectively offloading the server's burden. By combining this flag with the filtering logic within the pre-processing block, we can efficiently manage packet distribution between remote and local processing. Through an use case like the state-heavy Source Network Address Translation (SNAT) [9], which we present in section VI, we showcase the effectiveness of our proposed architecture in real-world scenarios.

B. Network Features Deployment

The top framework implies the partitions of a program executing, i.e., the remote handler executed after pre-processing and before post-processing. However, the switch ASIC cannot suspend and wait for remote processing in the middle of

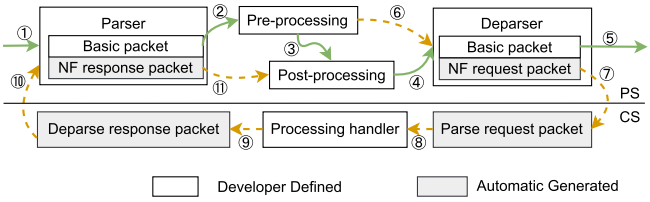


Fig. 3. The processing path when the network feature deployed. The green solid line is the fast path and the orange dotted line is the slow path.

the pipeline. To address this limitation, the top framework requires a different approach to effectively incorporate remote processing without interrupting the switch's packet processing.

To this end, the proposed framework employs an asynchronous model for remote processing. Instead of suspending the switch's pipeline, the switch forwards the packet requiring remote process to the remote handler while continuing its regular processing. The handler processes the packet on the remote server and sends the response. Upon receiving the response, the switch pipeline does with the post-processing for this packet. To implement NFs under the top framework, addressing the challenges posed by remote processing within the pipeline, three critical components need to be designed:

- Control flow. The P4 program needs to distinguish the response packets from the regular packets entering CLIP, and then decide which is next execution: the pre-processing, post-processing, or requesting remote process. The *flight* flag in the framework is also a key factor in influencing the control flow. Careful design is essential to avoid forming path loops between the switch and servers.
- Transmission of parameters. Efficient parameter transmission is vital for the success of remote processing. The framework must generate request and response headers that appropriately store and transmit the parameters and intermediate values required for packet processing. The header format varies depending on the network function.
- Forwarding Module. The forwarding module is responsible for traffic routing, server selection and route isolation between regular traffic and request packets. This module must ensure consistency between flows and their states and avoid disruptions in flows during scaling by efficiently managing traffic distribution among servers.

Writing this cross-platform program manually is tedious and error-prone, so we designed a compiler to automatically generate executable programs, including the aforementioned components. The details are as follows.

Control flow module distinguishes the original packets and request/response packets and decides on incoming packet processing, which is shown in Figure 3. The flight flag, along with the packet header type, plays a crucial role in guiding the control flow and deciding the path for packet processing. The control flow consists of two main paths: the fast path (①∼⑤) and the slow path (①, ②, ⑥∼⑪, ④, ⑤). The fast path is exclusively used for local processing within the programmable switches (PS), enabling quick packet handling. The slow path is designed for remote processing that involves interactions between the switch and the servers.

Parameters transmission module plays a critical role in communication between PS and CS by handling the transmission of parameters required for remote processing. To ensure efficient and error-free parameter transmission, the CLIP compiler takes charge of generating this module, alleviating the burden on the developers. The parameters for remote processing are piggybacked within request/response header of packets and assembled as the Ethernet packet payload. The format of the request/response header consists of a 2-byte load_type, NF-specific parameters, and an option field. The request header is appended to the regular packet in the switch, while the response header is appended to the modified packet in the server. The load_type field records the original packet's load type, copied from the Ether_type field of the Ethernet header. This prevents the loss of the original Ether_type value during the packet reassembly. To accelerate handler processing, two optimizations are applied within the request header: 1) The request header includes options that enhance processing speed. For example, combining the index option (i) and the length option (l) allows fast retrieval of data from the payload of the packet. Servers can directly extract a value with length l, starting from the i-th byte, as opposed to parsing the packet from scratch. 2) To avoid the extra packet copy during remote processing, which can slow down overall performance, the request and response headers are aligned. This is achieved by padding with zeros to ensure both headers match in length and byte alignment. As a result, parameter modifications from request to response only require header rewriting, eliminating the need for an additional packet copy. By automatically generating the parameter transmission module and incorporating these efficiency-enhancing tricks, the CLIP compiler ensures smooth and error-free parameter transmission between PS and CS.

Forwarding module: As regular traffic may adopt either L3 routing or L2 forwarding, and request packets need to be routed to a specific server, the forwarding module is responsible for isolating the request packet forwarding from regular packets. To achieve this, the forwarding module leverages the Equal-Cost Multi-Path (ECMP) group of the L3 Route to distribute the request packets to servers. For packets flagged with *flight*, the forwarding module assigns the corresponding group ID before routing, ensuring that the packet is directed to the appropriate CS for processing. This allows efficient and seamless handling of request packets without disrupting regular traffic flow. Another challenge the forwarding module addresses is the scaling problem further elaborated (§ V-B).

C. Function Expansion

We propose the simplified RPC generated at ASIC to utilize the remote resource. Yet, barriers to maximizing the advantages exist due to the difference in processing logic. Next, we delve into strategies for extending the programmable switch capacity, focusing on both memory and computation optimization.

Memory Optimization. In reconfigurable match action table switches, both the PHV and the match-action units are scarce resources, deciding the packet processing capabilities in the data plane. However, their fixed sizes, approximately

256 bytes for PHV and O(10M) for match-action units, present significant challenges in meeting the ever-growing demands of modern networks [10]. We propose the following mechanisms implemented in the compiler:

- *PHV Occupation Reducing.* The size of the PHV directly impacts the intermediate values and metadata that can be carried through the switch, affecting the logic complexity and depth. Remote processing naturally extends the depth of the pipeline by on-loading a portion of the function. However, PHVs are prone to be saturated if the size of intermediate values exceeds 200Bytes, compiling failures arise for functions in the switch part. Our observation reveals the presence of postponable intermediate values (PIV) in user-defined programs—values declared and assigned during pre-processing but referenced only in post-processing. Notably, these PIVs remain unoptimized by the P4 compiler since pre-processing and post-processing are merged into a single program fed into the P4 compiler. To address the compiling failure caused by PHV shortages, the CLIP compiler steps in when the P4 compiler signals this failure. It conducts code optimization to identify PIVs in pre-processing and move the corresponding declaration and assignment to the remote handler. This proactive measure, deferring PIV generation, enables the CLIP compiler to address the compiling issue caused by PHV allocation. For example, consider a switch implementing L3 routing that selects a group of next-hops based on a packet's destination IP address. It then uses the 5-tuple (Source IP, Destination IP, Source Port, Destination Port, and Protocol) to compute an index for selecting one next-hop to forward the traffic. Initially, the PHV must store the entire 104-bit 5-tuple as input for the hash unit, along with an 8-bit result for the next-hop selection index. In this case, the 4-tuple (the 5-tuple excluding the Destination IP) can be identified as a PIV. To reduce PHV usage, the extraction of the 4-tuple and the hash computation can be on-loaded to a remote handler, leaving the PHV to only store the 32-bit Destination IP for selecting the next-hop group and the 8-bit next-hop index.
- *Match-Action Table Mirroring.* Match-action table plays a crucial role in storing data for matching and processing packets, a pivotal aspect in handling diverse traffic patterns [28]. The expansive DRAM of servers enables the maintenance of extra match-action tables, meeting the growing demand for entries. Therefore, we introduce the concept of *table mirroring*, allowing the match-action table defined in pre-processing to be referenced and executed in the remote handler. This involves performing the same actions in the server's table (mirrored table) as those executed in the switch's table (original table). The mirrored table is a structural replica of the original table, sharing the same definitions of keys, match types, and actions. This approach enables users to install table rules that surpass the capacity of the original table into the mirrored table, mitigating the risk of table content overflow.

Computation Enhancement. The match action table relies on simple ALU operations for actions on integer values. Although sufficient for certain packet processing tasks (e.g., Time-to-Live (TTL) increments), these operations prove inadequate for more advanced applications such as parameter aggregation on float value [11], [12], [14]. Looking at a more flexible computation element, the register that can be updated with packet traversing the pipeline reveals a certain degree of flexibility. Moreover, using a register requires strict adherence to the template of computation, stored value, and argument width. For instance, each register can maintain a value or a value pair, combined with three fixed sizes (less than 64 bits) of data. Both the input and output of the register are restricted to not exceed 64 bits, which includes metadata, packet header, or stored value. Additionally, registers are arranged as a linear data structure, i.e., a program can access a specific register using the continuous integer, which is not extensible. Therefore, allocating a register for a specific flow without control plane involvement is challenging due to the extensive scope of flow IDs. Hence, we extend the remote handler to support more computations as follows.

- *Complex Computation Support.* In CLIP, we extend support for floating-point values and more complex computations than those simple ALU operations by loading these computations onto the remote handler. Here, floating-point operands are transmitted as request parameters, and the computation results are returned as responses. However, P4-defined data types are bit-based (e.g., bit<32> dst_addr) and lack direct support for float values. To address this, we enable users to define float operands as parameters for remote processing. To facilitate this, the compiler translates the user-defined float values into device-specific data types. For example, a float value A will be translated into bit<32> switch_A on the switch side according to the IEEE Standard for floating-point arithmetic and float value server_A on the server side. Notably, the float version value on the switch can be appended to packets rather than engaging in additional computations in the switch due to semantic differences. By assigning bit-based value to parameters, users gain the flexibility to define computations with more diverse data types, accommodating the needs of complex calculations beyond basic ALU operations.
- *Stateful Variable.* To enhance flexibility in stateful computation, CLIP introduces the concept of a *stateful* variable which is maintained at servers and can be both read and updated at remote handlers. Unlike mirrored tables, the stateful variable is explicitly defined in the CLIP program and supports various types, including basic data types, arrays and user-defined data structures. The stateful variable is used for servers to maintain states across multiple packets. To illustrate the necessity of stateful variable, consider the scenario of managing an address pool for NAT. The stateful variable proves important in dynamically updating the status of addresses in response to allocation and release, marking them as 'invalid' or 'valid' accordingly. A stateful variable of bitmap can be used to address the register index challenge. This bitmap mechanism quickly identifies an unused register index, allocating it to a new flow.[1]

[1]Note that this approach cannot be implemented in ASIC switches using registers, as registers can be accessed only once for a packet. Once the target index is occupied, there is no opportunity for a retry.

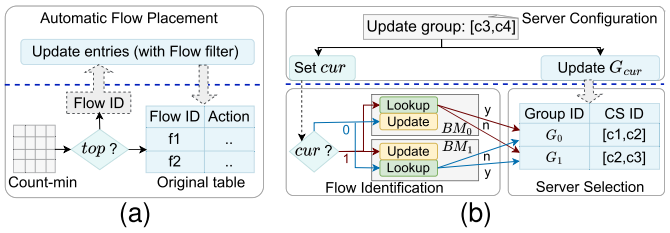


Fig. 4. Load balancing for keeping affinity between flows and their states. (a) Automatic flow placement. (b) Server group scaling.

The variable is distributed among multiple servers, and its consistency is guaranteed (§V-B).

V. Performance and Scalability

There exists a substantial performance gap, which spans multiple orders of magnitude, between programmable switches and general-purpose processors for packet processing in terms of both latency and throughput. It is imperative to ensure that remote processing does not become the bottleneck, impeding the switch's throughput. To address this challenge, one plausible approach is to selectively forward only a portion of packet data or a fraction of packets for remote processing. However, transmitting partial packet data becomes impractical due to the switch ASIC's limitations in buffering high volumes of packets while awaiting remote processing. Directing a fraction of the packets to servers becomes a viable solution.

Therefore, we implement a flow-based load balancer to distribute traffic among the switch and servers. The goal of existing load balancers is dispatching traffic into flatted, distributed nodes. Based on the architecture of CLIP, the instances are organized into a two-layer tree where the switch is the root and the servers are leaves, and the traffic enters the system through the root. Therefore, CLIP load balancer places most of the traffic in the root to gain latency performance by reducing the length of path that traffic traverses and improving throughput performance by adding the leaves.

A. Automatic Flow Placement

For functions that only require memory expansion without the need for remote processing, CLIP leverages the switch's capacity to handle resource-intensive traffic, thus relieving the burden of servers. We design the different placement strategies to place the flows for exact match (EM) and longest prefix match (LPM) types.[2]

EM tables placement. The EM table can be divided into multiple tables based on flows, e.g., five-tuple or destination IP. CLIP optimizes resource consumption by using the original table in the switch to handle the top-N flows. However, directly selecting the top-N flows in the switch's data plane is space- and time-inefficient. This approach requires counting packets for each flow, iterating counters for all flows, and maintaining a large table of counters, resulting in a time complexity of $O(m)$, where m is the size of this table. Another line is counting all flows in the data plane, reporting statistics to the control plane and then selecting the top-N flows in the control plane, but it could consume excessive data-plane-to-control-plane bandwidth.

Rather than reporting specific N top flows, we opt for a more efficient approach through a two-phase selection strategy. First, in the switch data plane, we implement a count-min sketch with compressed space[3] to estimate the sizes of flows. Instead of iterating through the entire table, we report only the flows that surpass a mutable threshold regarding the number of arrival packets and flows. Second, in the switch control plane, it selects the final top N flows from the reported flows for flow migration. The first selection phase can efficiently reduce the number of reported flows, thereby reducing the bandwidth consumption in switch-ASIC-to-switch-CPU communication and the time consumed in the second phase.

The threshold setting should meet the aim of the first phase selection: 1) select at least N flows so that the switch processing resource can be fully utilized; and 2) the number of reported flows does not exceed N by too much, considering the time complexity of the second phase selection. We define the threshold as $\frac{k \cdot pkt}{flow}$, where pkt and $flow$ is the number of arrival packets and flows, respectively. Both pkt and $flow$ can be collected from the switch data plane. By dynamically adjusting the parameter k, we can control the number of reported flows. If the number of reported flows ($flow_r$) exceeds N (i.e., $flow_r - N > \epsilon N$, ϵ is a predefined tolerance factor), we set a more aggressive threshold by increasing k in the next report, which filters out smaller flows. Conversely, if the number of reported flows is less than N (i.e., $flow_r - N < \epsilon N$), we reduce the threshold by decreasing k.

LPM tables placement. Different from EM types, directly distributing entries into the two-layer LPM tables may lead to incorrect results. Take the example of IP routing tables, if an IP address belongs to multiple network prefixes, where a shorter prefix is in the switch and a longer prefix is in servers, a packet may be forwarded based on the shorter network prefix, violating the semantics of LPM. This situation calls for careful consideration and specific handling to ensure accurate flow selection decisions.

To mitigate this issue, we introduce a flow filtering step after reporting the top flows and before updating the entries. The flow is identified using the fields same as the key defined in the original LPM table. Specifically, we select flows that match the longest prefix in servers and then migrate the longest prefix to the switch. Further, this LPM entry migration does not just affect the flow triggering the migration but also other flows matching the same prefix. This approach maintains match correction and can enhance performance by handling more traffic at the switch.

Put them together. As the illustrated process of automatic flow placement (Figure 4a), arriving packets update and look

[2]A ternary match kind on a key field means that the field in the table specifies a set of values for the key field using a value and a mask. This kind of match type has no efficient data structure in the software to achieve O(1) lookup. Therefore, we left it for future work.

[3]Count-min sketch is commonly used for the task of network measurement. It uses a multi-way array, named B, compressing of n rows and m columns buckets to estimate the frequency of items. For a coming item with a key k, it increases counters in buckets indexed by $(i, F_i(k)\%m)$, where $0 < i < n$. The frequency of item with k is identified as $min_i(B(i, F_i(k)\%m))$.

up the count-min sketch based on the flow's identifier. The result of this lookup provides the number of packets associated with that flow. If the packet count exceeds a defined threshold (top), the flow is reported to the control plane. For cases where the original table is an LPM table, a flow filter is applied in the control plane. Subsequently, the table entries corresponding to these (optionally filtered) flows are migrated from the mirrored table to the original table. This process replaces aged flow entries, optimizing resource allocation for the specific flows.

B. Auto-Scaling of Server Group

These remote servers, acting as a supplement to the switch, should be capable of scaling to adapt to varying workloads. In response to dynamic workloads, we deploy multiple servers to enhance platform throughput. Effectively leveraging server resources involves addressing two pivotal issues: (1) How to alleviate the processing burden on servers, and (2) How to handle traffic if the existing servers are unable to process it effectively.

Strawman solution. A straightforward approach involves implementing a load balancer at the switch to distribute the traffic to servers. Conventional distributed hashing schemes such as consistent hashing [29] and rendezvous hashing [30] are alternative options. However, they partition the hash tables and maintain numerous <bucket range, server ID> mappings. To maintain affinity between packets and their associated states, especially in cases where server group members change, stateful load balancing mechanisms like [13] can be employed. It enables connection consistency at the programmable switch by recording the per <connection, server ID> mapping, which offers an effective load balance. However, both of them consume O(N) on-chip memory space, where N is the number of bucket ranges or connections.

Dynamic scaling avoiding inter-server flow migration. The fix-number of servers can not fit the increasing workloads, so a dynamic approach is proposed. However, main challenge is it should ensure consistency in the handling of flows and their associated state. It involves adjusting the composition of the server group in real time to accommodate varying system capacity requirements without interrupting traffic.

This dynamic server group scaling relies on the effective handling of flow consistency when group membership changes. A flow-based ECMP mechanism is employed to distribute traffic across the server group, with flow states created and maintained by the server handling the first packet of a flow. However, when group membership changes, it can lead to the re-distribution of flows and flow state migration. Synchronizing states through state migration is complex, often involving stopping or buffering traffic until state migration is completed. To avoid these problems, a small on-chip memory is used to maintain a snapshot of group membership for flows arriving at different times. This snapshot helps subsequent packets of a flow find their original server, thus eliminating the need for state synchronization. While creating a snapshot for each flow incurs a non-trivial cost, we record the group membership and remember the arriving flows using bloom filter[4] before the group updating, and then distributing new flows and existing flows to their corresponding server.

The procedure of server group scaling is illustrated in Figure 4b. We mark a CS group with a version ID as the membership changes at time point T_i. The newest group is called G_i and the previous one is called G_{i-1}. The flows arrived during T_i and T_{i-1} are routed into the one group of G_{i-1} while flows arrived after T_i are distributed into the newest group G_i. The next problem is how to distinguish the new flows from the arrived flows during T_i and T_{i-1} with a little memory consumption. We consider multiplexing the BM_0 and BM_1 to identify the flow alternately arrived during a period. As shown in Figure 4b, before updating the new group member G_{cur}, the control plane set a register $cur(cur \in \{0,1\})$ to $(cur+1)\%2$ that indicates the current group ID. We define operations on BM for packet arrival to elaborate on this procedure:

- *update*: sets values of BM to 1, using the flow ID of the packet as the key.
- *lookup*: checks BM using the flow ID of the packet as the key to determine if the packet belongs to an existing flow. If so, it returns y; otherwise, it returns n.

Each packet checks the value of a register cur $(cur \in \{0,1\})$. If $cur = 0$, it *updates* the BM_0 and *lookup* BM_1. If the lookup result is y, which indicates this packet belongs to an existing flow, it selects a destination from G_1. Otherwise, it selects a CS from G_0 as the destination. By memorizing the group membership and arriving flows before group updates, the system can efficiently scale without disrupting traffic.

VI. Case Study

To verify the effectiveness of CLIP, we implement three network features using CLIP.

Multi-tenancy Forwarding Information Base (FIB). It serves as a forwarding module within the cloud gateway, responsible for looking up large-scale routing tables for various cloud tenants [17]. With the continuous growth of tenants, we cannot maintain the performance by solely relying on software-based forwarding. To overcome the bottleneck associated with CPU-based packet processing, a hardware acceleration solution is required. However, since the current hardware switch can only support up to 100K LPM entries, it is insufficient to accommodate the large tables needed for cloud-scale tenants.

In CLIP, we combine the CS and PS to support 900K forwarding rules. Initially, all 900K rules are maintained in mirrored LPM tables. As traffic arrives, the top 10K flows can be migrated to PS. Based on that the top 1% of the largest flows contribute to 70% of overall traffic [17], we can efficiently accelerate FIB and alleviate the limitations posed by hardware memory constraints.

Overlay Network TCP Retransmission Detection (TRD). TRD monitors TCP status and reports flows that are experiencing retransmission, aiding in the detection of network

[4]Bloom filter is a probabilistic data structure that indicates the element either *definitely is not* or *may be* in the set. Its basic data structure is a multi-way bit vector and can do read-check-write in one cycle.

failures [31]. In TRD, the sequence numbers of packets belonging to the same TCP connection are recorded. Retransmission of the TCP connection is detected when packets with lower sequence numbers arrive. Additionally, TCP flags are probed to determine the TCP status (establishment or disconnection). When TRD is deployed at switches, both sequence numbers and TCP state must be maintained in the data plane to ensure timely updates. Although the register array can be updated in the data plane, this operation still introduces two challenges.: 1) the register array can only be indexed by continuous integers, while a TCP connection is identified by its four-tuple, and 2) the limited memory of switches means registers are insufficient to support a large number of TCP connections.

In CLIP, the remote handler manages TCP status maintenance and register index allocation. The pre_processing stage maintains: 1) a match-action table called the Flow-Index table, recording the mapping from flow ID (the four-tuple) to a register index; and 2) a register array, named the sequence number (SN) array, storing the sequence numbers of TCP connections. The workflow is as follows: when a TCP packet arrives, it looks up the Flow-Index table. If the packet matches an entry, it indicates that the connection is active. The register index is then used to check and update the SN array. If the recorded number is larger than the current sequence number, the flow is identified as being in a retransmission state. If there is no match in the Flow-Index table or if the packet carries TCP control flags (SYN, FIN, etc.), the packet is sent to the remote handler. The remote handler creates or releases records of TCP connections based on the TCP flags. It allocates a new register index for new connections when a TCP connection is established and releases the index when the connection is disconnected. The response returns the allocated index to the switch, enabling or disabling the slot in the SN array for a connection. To extend the SN array in the switch, CSs record the sequence numbers for overflowed connections once the SN array reaches saturation.

State-heavy Source Network Address Translation (SNAT). SNAT is a gateway middlebox that isolates the internal network from external networks. When receiving traffic from the internal network, SNAT allocates a new <IP address, port> pair and rewrites the packet header accordingly. It memorizes the mapping from the flow ID (five-tuple) to the new pair, allowing it to process subsequent packets based on this mapping. The widespread deployment of SNAT on programmable switches is hindered for two main reasons: 1) SNAT is state-heavy, requiring significant memory to store these address translation mappings. 2) The switch data plane is incapable of executing pair allocation because the match-action table can only be updated by the switch control plane.

Through CLIP, we leverage the CS memory to provide ample space for storing NAT mappings. We can employ a remote handler to perform the allocation of the address-port pair. The workflow is as follows: when a packet arrives at PS, SNAT looks up the mappings during pre-processing. If a mapping hits, the packet is rewritten during post-processing. If a mapping misses, the packet is processed by the remote handler, which allocates a new pair for the flow and inserts the mapping into the mirrored table. The new pair is then returned to post-processing module to rewrite the packet. Once a flow has its mappings, it can be migrated to PS to enhance the performance if it is a heavy flow.

Example. We explain CLIP framework design by taking SNAT as an example. The software-defined and CLIP-defined SNATs are shown as follows, respectively.

```
void snat() {
  FLOW_ID flow_id = pkt->get_flow_id();
  VALUE *value = nat_table.lookup(
      flow_id);
  if (value == NULL) {
    value = allocation(flow_id);
    nat_table.insert(flow_id, value); }
  pkt->set_snat(value);
  pkt->send();  }
```

```
/* Define parameters */
request {flow_id}; response {sub_addr;
    src_addr};
/* Define functions deployed at CS */
void remote_handler(request req,
    response res){
  res = mirrored_nat_table(req.flow_id)
      ;
  if (res == NULL){
    res = allocation(req.flow_id); }}
/* Define P4 program deployed at PS */
control pre_processing(inout header hdr
    , out bool flight, out request req)
    {
  table nat_table = {
    req.flow_id : exact;..}
  apply {
    req.flow_id = hdr.tuple5;
    if (nat_table.apply().miss) flight
        = true;  }}
control post_processing(inout header
    hdr, in~response res){
    hdr.src_addr = res.sub_addr;  }
/* Top layer control flow */
control pipeline(inout header hdr,
    inout metadata md){..}
```

VII. Evaluation

This section evaluates CLIP on the testbed to address the following questions.

(1) What is the overhead of remote processing and performance improvement through load balancing and flow migration? (§VII-B)

(2) How does the CLIP framework facilitate the deployment of new features? What are the performance benefits of functions implemented using CLIP compared with software implementation? (§VII-C)

(3) What is the on-chip resource consumption of the ASIC when offloading forwarding modules through CLIP? (§VII-D)

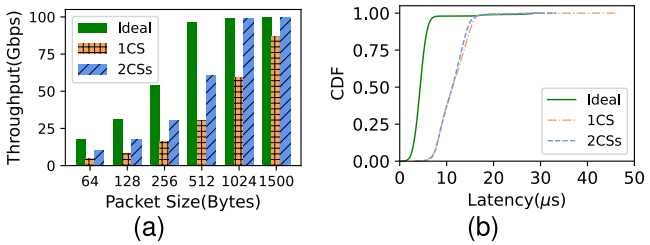


Fig. 5. Performance without application deployed. The "Ideal" shows all traffic traversing the switch. (a) Average throughput. (b) Latency distribution.

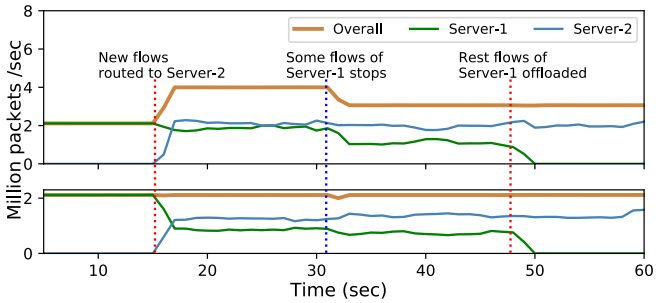


Fig. 6. Throughput changes as the CS group membership changes. The time points of the red dotted line indicate server-changing commands from the controller. The blue one shows the traffic-reducing time point.

A. Experiment Setup.

Testbed Setup. The testbed comprises a Wedge 100BF-32X 32-ports programmable switch with a two-pipeline Barefoot Tofino P4 ASIC and three servers. The network interfaces of the switch are configured to run at 100Gbps, achieved by aggregating four 25Gbps network channels. Each server in the setup is equipped with Intel Xeon Silver 4110 CPUs (2.10GHz, 8 cores) and a Mellanox ConnectX-6 NIC. Servers run Ubuntu 18.04 with Linux kernel version 4.15. All three servers are interconnected through the switch via 100 Gbps links. For the backend servers, two dedicated servers are allocated. DPDK [3] version 21.05 is deployed on these servers. DPDK-pktgen [32] is deployed on one server as the packet generator. Specifically, one port is assigned three cores for packet generation, while another port is allocated additional cores for packet reception.

Work Loads. We evaluate our system using synthetic traffic, six traces of realistic public workloads, and a public route dataset. The synthetic traces are generated to mimic the Zipf distribution in terms of the number of packets per flow [33], [34]. For realistic workloads, three are collected from an access point [35], backbone [36], and a data center [33]. The remaining three workloads are derived from scenarios of instant messaging [37], web search [38], and video streaming [39]. Additionally, we use a public route dataset containing IPv4 Prefix-to-Autonomous System mappings derived from RouteViews data [40]. This dataset includes nearly 964,000 prefixes, with 24-length prefixes being the most common, comprising about 59.32% of the entries.

B. Microbenchmarks.

Forwarding Capacity. To improve forwarding capacity, multiple server instances are employed because packet processing at a single server can provide no more than 100Gbps throughput, which is significantly lower than what the capacity of commodity switch. We continuously generate packets of varying sizes at the maximum speed of the generator and forward them to the switch. In an ideal scenario where all packets traverse only the switch, the performance upper bound is established, indicating the maximum packet generating/reception rate. As depicted in Figure 5(a), for packets ranging from 512 to 1500 bytes, the packet generator utilizing three cores can achieve a near 100Gbps reception rate. In the case of a single CS with one core, it can achieve 50Gbps forwarding efficiency for 1024-byte packets. As the packet size increases, a higher throughput in Gbps is attained. This trend is attributed to the nature of software-based processing, which operates on a packet-by-packet basis and can process a fixed number of packets per second. The use of parallel CS instances contributes to the overall improvement in throughput.

The introduction of remote handlers can enhance processing capabilities but also introduces additional latency. We measure the latency of packet transmission to evaluate this extra latency. Figure 5(b) demonstrates the latency distribution, where 10000 packets of random size are injected to measure the forwarding latency. With CSs involved, 90% packets can traverse the platform within 16μs. The remote processing introduces an additional latency of approximately 9μs for 90% of packets. Prior work like TEA [9] introduces about 2μs to expand ASIC memory. The extra 9μs latency in our approach allows for the expansion of not only memory but also possible computational ability.

Capacity Adjustment. To validate that the scaling operation of SC does not disrupt traffic, we monitor the throughput during membership changes. To validate that the CS group membership change would not disrupt the traffic processing, we monitor the throughput of server-1, server-2 and the whole system during the membership transition from one group (server-1) to another (server-2). We inject four sets of flows with different transmission rates and start and stop times to simulate real-world traffic. As shown in Figure 6, we first send flow set-1, set-2, and then set-3, set-4 at around 15 sec with the same rate (about 1 million packets per flow set per second). Finally, we stopped sending set-1 at around 31 sec.

As shown in Figure 6, all traffic is initially routed to server-1 until the group is changed to server-2 at around 15 sec. The newly arrived flows (set-3 and set-4) are distributed to server-2, while the flows reached before the membership change (set-1 and set-2) are still sent to server-1. If another change happens, it triggers the redirection of the remained flows on server-1 (set-2) to be offloaded to the switch. The destination of flow set-3 and set-4 is still server-2. While the bottom figure indicates that the overall throughput fluctuates very slightly during the changes of backend server.

Throughput Improvement with Flow Migration. By migrating heavy traffic from servers to the switch, we can improve the system throughput and avoid the servers from becoming the bottleneck. We evaluate the throughput improvement with flow migration by generating packets of 64B

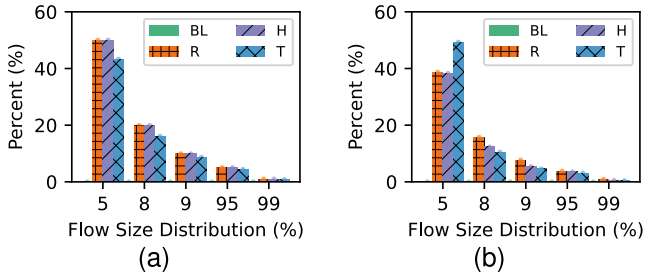


Fig. 7. Flow selection percentage under different strategies. (a) EM. (b) LPM.

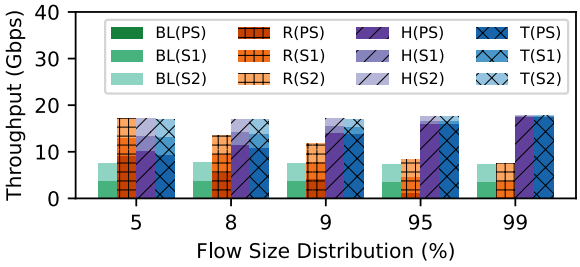


Fig. 8. EM Performance of under different selection strategies. The "BL" shows all traffic traversing the switch. The darkest color is the throughput of traffic handled by the switch (PS) and the two lighter colors are the throughput of traffic handled by servers (S1 and S2).

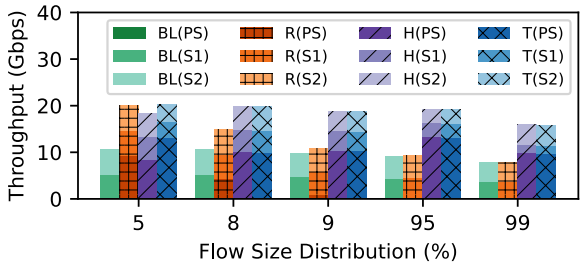


Fig. 9. LPM Performance of under different selection strategy.

with different flow size distributions, where α denotes traffic skewness following the Zipf distribution.

In Figure 7, the flow selection percentages are depicted for three flow migration strategies: random (R), count-min sketch with max-heap (H), and count-min sketch with threshold (T). The baseline (BL or non-migrating) represents results where all traffic is handled by servers (S1 and S2). Strategy-R randomly selects flows to migrate. In strategy-H, the data plane of the switch reports statistics for all flows and then executes a max-heap at the control plane to select top flows. In strategy-T, the data plane of the switch only reports flows whose statistics exceed the threshold, reducing the overhead of flow selection compared to strategy-H.

Strategies R and H of EM select $(1-\alpha)\%$ flows. We aim for strategy-T to achieve the same performance as strategy-H, which strictly selects top flows, and consumes fewer table resources and channel bandwidth. The threshold of strategy-T is initialized as $\frac{2\alpha\#pkt}{\#flow}$, where the number of packets ($\#pkt$) is collected at the switch and flows ($\#flow$) at servers. Theoretically, strategy-T reduces $\alpha\%$ of channel bandwidth compared to strategy-H if both strategies select the same number of flows. This is because strategy-T only reports selected flows rather than statistics for all flows like strategy-H. The selected flow numbers for EM are shown in Figure 7 (a). Strategy-T selects fewer flows than strategy-H and strategy-R under all flow size distributions. Therefore, strategy-T reduces the bandwidth consumption of the channel by about 56.7% $\sim$ 99.2% compared to strategy-H.

Figure 8 illustrates the throughput of EM under the three strategies. Strategy-R helps improve performance under lower skewing conditions ($\alpha = 0.5$ and $\alpha = 0.8$). However, it becomes challenging to accurately identify heavy loads under severe skewing traffic ($\alpha > 0.9$). Strategies T and H push the switch (PS) to handle more traffic under highly skewed scenarios, thereby reducing the processing burden on servers. We believe the overall throughput can be improved with a more powerful packet generator. Furthermore, compared to strategy-H, strategy-T can select a smaller number of flows and achieve similar throughput.

In contrast to the parameters' setting in EM, LPM introduces a filter that selects prefixes with specific lengths to prevent incorrect matches. For this evaluation, we perform IPv4 lookup based on LPM using IPv4 route prefixes from the public dataset [40]. Therefore, for the threshold of the LPM application, we use the threshold of strategy-T with an extra discount, i.e., the percentage of 24-prefix. This threshold setting allows the selected flow numbers of strategy-T to be close to strategy-H, as shown in Figure 7 (b).

Figure 9 shows the throughput of the LPM table under the three strategies and selected flows. Similarly, strategy-R performs well at lower skewing levels ($\alpha = 0.5$ and $\alpha = 0.8$), while strategy-H and strategy-T can achieve better performance even under severe skewing traffic conditions ($\alpha > 0.9$). Unlike EM, LPM still leaves some flows to be handled by servers; therefore, it does not achieve as high as the throughput gained at EM.

TABLE I
THE ACCESSING LATENCY UNDER DIFFERENT STRATEGIES

	BL	R	T	H
EM	17.92±5.94μs	7.45±7.62μs	3.07±3.06μs	2.74±1.93μs
LPM	18.92±5.77μs	17.02±7.38μs	12.11±8.94μs	9.78±8.13μs

Latency Improvement with Flow Migration. Flow migration reduces the processing burden of servers so that it can improve latency distribution of tables in CLIP. Thus, we evaluate the latency improvement under different migration strategies. We marked packets with timestamps at sending and receiving, capturing 1 million packets with a flow size distribution at $\alpha = 0.9$ to evaluate the end-to-end latency when accessing the EM table and LPM table. The latency of accessing for end-to-end is shown in Table I, where the numbers after $\pm$ denote the standard deviations. By migrating flows to the switch, CLIP reduces average latency by 84.71% and 48.3% for EM and LPM, respectively, based on strategy-T.

Looking at the latency distributions shown in Figure 10, the latency for 90% of flows (for EM) and 37% of flows (for LPM) is below 8 μs for strategy-T and strategy-H, which is a significant contribution to latency improvement. Furthermore, there is an improvement in tail latency. This is attributed to the reduction in the number of packets in each processing batch, effectively minimizing the queuing time.

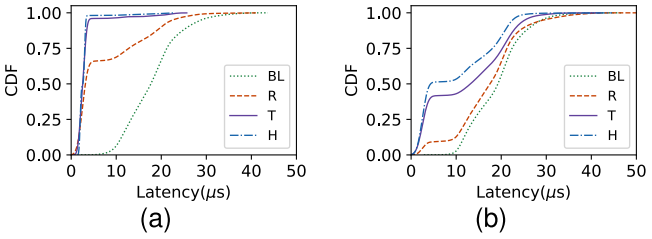


Fig. 10. Latency under different selection strategies. (a) EM. (b) LPM.

TABLE II
THE LATENCY OF TABLE OPERATION

Match Type	Insert-1	Delete-1	Insert-1k	Delete-1k
EM	1.55±0.49ms	1.32±0.39ms	499.90±16.01ms	366.86±12.56ms
EM-m	9.39±1.19μs	2.36±0.47μs	121.06±23.17μs	90.32±31.38μs
LPM	1.32±0.33ms	0.99±0.27ms	285.27±8.41ms	143.62±8.93ms
LPM-m	7.72±1.51μs	3.93±0.49μs	279.84±16.39μs	215.77±9.34μs

TABLE III
THE LATENCY OF NETWORK FEATURE

Network Feat.	CLIP	Software-based
SNAT	7.37±1.92μs	22.71±11.00μs
FIB	7.14±1.05μs	25.06±15.06μs

TABLE IV
THE ADDITIONAL RESOURCE USAGE OF SYSTEM

Resource	**Selecting**	**Scalability**	**Forwarding**
Match Crossbar	0.33%	0.59%	1.88%
SRAM	3.75%	0.83%	2.08%
TCAM	0	0	0.35%
VLIW Instruction	0.52%	2.08%	1.82%
Hash Bits	0.68%	1.37%	4.06%

Flow Migration Overhead. The flow migration procedure consists of two steps: 1) migrating heavy flows from servers to the switch and 2) migrating aged flows from the switch to servers. This process involves inserting and deleting table entries in both the original and mirrored tables. Because the operations are non-blocking, i.e., migrations do not involve traffic halting, the latency of operations in both tables does not introduce performance degradation. However, the latency of table operation affects the efficiency of flow migration. Thus, we created programs with different match types and various numbers of flows to test migration latency. Specifically, we measured the latency of updating 1 and 1,000 entries for EM and LPM tables required for migrating flows from servers, as well as the latency for updating the same entries in mirrored EM (EM-m) and LPM (LPM-m) tables locally. The results are shown in Table II. Updating an entry in the mirrored table takes 10 μs, which is negligible compared to the same operation on the original tables. The latency of operations on the original tables is tens of milliseconds. This is because match-action tables in the switch can only be updated by the control plane, not the data plane. Consequently, the latency for updating original tables primarily results from the time taken for entry-install messages to be delivered from the switch CPU to the switch ASIC, the entries to be inserted into the original table, and the results to be returned to the switch CPU. The migration efficiency can be improved using batch updating, which has an average latency of 516 μs per flow.

C. Network Features Deployment

CLIP can accelerate existing software middleboxes such as NAT, Load Balancer, and Firewall. These middleboxes have a similar processing model: applying for entries from the control plane and performing match-action with high speed for particular flow ID, five-tuples, destination IP, etc. The main problem with deploying them to programmable switches is the on-chip memory limitation. We evaluate SNAT and FIB performance as representatives and compare them with the current software design.

Feature Performance. We run SNAT and FIB implemented using FastClick [19] at one CS. For a fair comparison, both applications run on the same server and use one core. TCP retransmission detection is a new feature without the public software implementation as we know it. We evaluate applications' performance in terms of throughput and latency. Table III shows the latency of applications in CLIP and Fastclick. Compared to software-based SNAT and FIB, the average latency of CLIP-based has reduced by nearly 67.5% and 71.5%, respectively.

To evaluate the end-to-end throughput, we replay six packet traces.[5] To analyze the benefits of CLIP on different workloads, we first show the packet size distributions of those traces, as shown in Figure 11 (a). We classify the six traces into two groups: (AP, BB, DC) and (IM, WS, VS). Within each group, flows with large packets become increasingly frequent in sequence. The throughput of two applications is shown in Figure 11 (b) and (c). Compared to the software-based implementation, CLIP achieves 1.36× to 4.05× higher throughput for SNAT and 2.85× to 16.06× higher throughput for FIB. Throughput improvement (blue line) is more significant for workloads dominated by small packets. This suggests that flow migration effectively reduces the bottleneck caused by server CPU consumption when processing small packets. This observation is not obvious in FIB, as it uses a filter to select the longest prefix to ensure look-up correction, rather than relying solely on traffic distribution.

D. Resource Consumption

We evaluate how much ASIC resource is consumed only by CLIP based on the P4 compiler's output. Table IV shows the resource consumption of throughput-intensive traffic Selection (count-min sketch), Scalability, and Forwarding module. We observe that there are plenty of resources remaining to implement other functionality on the ASIC along with CLIP. The SRAM and Hash Bit consume the most in a short time for the scale of entry. The selecting module is made of registers that are deployed at SRAM. Thus, its SRAM space usage depends on the total number of count-min size, and in this

[5]We utilize the traffic header and append payloads ranging from 64 bytes to 1500 bytes to simulate the data center traffic distribution because the payload of the trace has been anonymous.

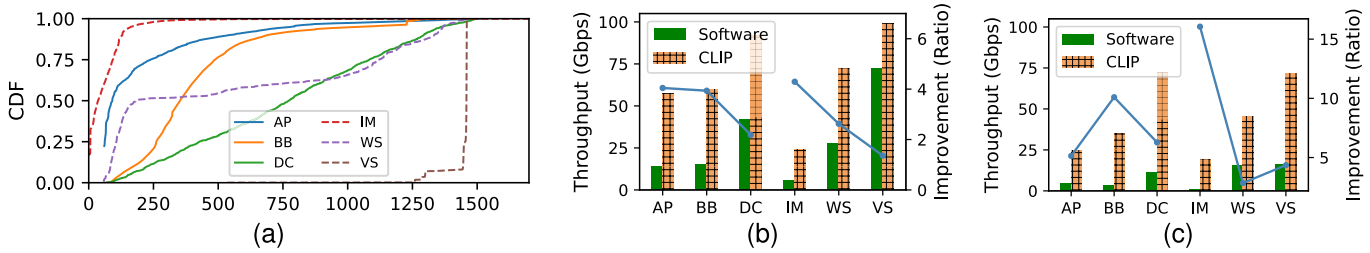


Fig. 11. The throughput of application performance under six traces: Access Point (AP), Backbone (BB), Data Center (DC), Instant Message (IM), Web Search (WS) and Video Stream (VS). (a) Packets size (byte) distribution of traces; (b) SNAT; (c) FIB.

evaluation, we set 1024 2-way count-min sketch. Besides, it consumes some other amount of TCAM, VLIW instruction, and hash bits, all less than 5%. The scalability module and the forwarding module consume SRAM and hash bits to store metadata and resolve the bucket. The forwarding module uses the TCAM to support LPM lookup. As the Scalability module shares the hash calculation with Forwarding, its occupation of the hash bit is under 2%.

VIII. Limitations and Discussion

Cost of Remote Processing. The primary costs of remote computation are the additional latency and the reduced number of available switch ports. The added latency is unavoidable since some packets must traverse extra hops. However, we argue that the increased feature velocity justifies this trade-off. The overall number of available switch ports is decreased because some ports are needed to connect the servers. This introduces a trade-off between processing ability and the available ports. We provide the dynamic scaling mechanism(§V-B) that allows users to determine which servers connected to switch ports can be included in CLIP. This enables users to make decisions based on their actual requirements.

Machine Learning Acceleration. CLIP provides a solution to enhance distributed machine learning (ML) architectures. For example, the programmable switch can perform gradient aggregation as the parameter server for deep neural network training. While servers can function as workers to compute gradients. Due to ML applications being latency-intensive, the system latency should be optimized. The propagation latency between the programmable switch and servers is comparable to the propagation latency between a traditional parameter server and workers. Moreover, programmable switches can accelerate gradient aggregation, thus reducing processing latency. To further enhance computational efficiency and flexibility, remote processing should contain various resources, such as graph processing units (GPUs).

IX. Related Works

Programmable Dataplane Enhancement. The trend in cloud networks is to enhance programmability, but deploying current programmable devices in real production environments comes with challenges. Some works, such as [9], [10], [41], aim to extend and explore programmable match-action tables, but not focus on PHV, analyzable header length, and register capacity limitations. P4All [23] defines elastic data structures and translates them into native P4 programs to improve expressiveness. However, it still faces limitations in primitive computations, such as floating-point computation. FPISA [42] attempts to enable floating-point representation in programmable switches but encounters architectural limitations in existing hardware. Other works focus on different restrictions; for instance, IPSA [24] aims to achieve online updating for programmable switches by restructuring their architecture.

Heterogeneous Platforms Cooperation. Considering combining the strengths of different programmable hardware, several works [14], [25], [43], [44], [45] have designed architectures, languages, compilers, or toolchains to facilitate cross-platform cooperation. Gallium [14] translates existing Click-defined software middleboxes into separate programs running on programmable switches and servers. It focuses less on new feature deployment limitations and the performance gap of heterogeneous platforms. Flightplan [43] aims to automatically disaggregate a P4 program into ASIC, FPGA, and CPU. Lyra [44] designs a cross-platform language and compiler for heterogeneous ASICs on the switch. RIBOSOME [25] splits packet processing into headers processing and payloads buffering, and deploys processing on multiple kinds of devices. All of them have limitations in function expressiveness concerning P4 or NLP [20]-defined behaviors. CLIP leverages a small scope of topology to explore the network features expressiveness and deployment.

X. Conclusion

Inspired by the inherent flexibility of software network functions, we advocate for harnessing a shared server cluster to swiftly augment the capacity of network switches. This architectural innovation caters to the demands of network customers that may exceed the capabilities of existing commodity or programmable switches with limited resources. The framework establishes an environment abundant in resources. To address challenges tied to performance and complexity, we introduce traffic balancing across multiple servers and strategically allocate network functions to fully maximize resources in both hardware switches and software functions on servers. Lastly, we present applications that capitalize on this advanced programmable data plane.

Acknowledgment

The authors would like to thank anonymous reviewers for their valuable comments.

REFERENCES

[1] A. Sivaraman et al., "Packet transactions: High-level programming for line-rate switches," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 15–28.

[2] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[3] I. DPDK. (2014). *Data Plane Development Kit*. [Online]. Available: https://www.dpdk.org/

[4] D. R. Barach and E. Dresselhaus, "Vectorized software packet forwarding," U.S. Patent 7 961 636, Jun. 11, 2011.

[5] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[6] R. Giladi, *Network Processors: Architecture, Programming, and Implementation*. San Mateo, CA, USA: Morgan Kaufmann, 2008.

[7] Intel, Tofino, BC, Canada. (2021). *Intel Programmable Ethernet Switch Products*. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html

[8] (2020). *Trident4 / BCM56880 Series*. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series?_ga=2.101838278.670967709.1597289176-917906606.1597289176

[9] D. Kim et al., "TEA: Enabling state-intensive network functions on programmable switches," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 90–106.

[10] T. Pan et al., "Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 194–206.

[11] A. Sapio et al., "Scaling distributed machine learning with in-network aggregation," in *Proc. 18th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2021, pp. 785–808.

[12] C. Lao et al., "ATP: In-network aggregation for multi-tenant learning," in *Proc. 18th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2021, pp. 741–761.

[13] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, Aug. 2017, pp. 15–28.

[14] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 283–295.

[15] T. Xu, X. Wang, C. Tian, Y. Xiong, Y. Lin, and B. Ye, "CLIP: Accelerating features deployment for programmable switch," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2023, pp. 1–10.

[16] K.-K. Yap et al., "Taking the edge off with espresso: Scale, reliability and programmability for global Internet peering," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, Aug. 2017, pp. 432–445.

[17] H. Shao, X. Wang, Y. Lu, Y. Yu, S. Zheng, and Y. Zhao, "Accessing cloud with disaggregated software-defined router," in *Proc. 18th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2021, pp. 1–14.

[18] FD.IO. *VPP*. Accessed: Jul. 22, 2024. [Online]. Available: https://wiki.fd.io/view/VPP

[19] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2015, pp. 5–16.

[20] (2021). *GitHub: NPL-Spec*. [Online]. Available: https://github.com/nplang/NPL-Spec

[21] X. Jin et al., "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Operating Syst. Princ. (SOSP)*, Oct. 2017, pp. 121–136.

[22] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, Aug. 2018, pp. 561–575.

[23] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker, "Modular switch programming under resource constraints," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 193–207.

[24] Y. Feng et al., "Enabling in-situ programmability in network data plane: From architecture to language," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp.‘635–649.

[25] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa, "A high-speed stateful packet processing approach for tbps programmable switches," in *Proc. 20th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2023, pp. 1237–1255.

[26] T. P. A. W. Group. (2019). *P4RunTime Specification*. [Online]. Available: https://p4.org/p4-spec/p4runtime/v1.0.0/P4Runtime-Spec.html

[27] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.

[28] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, "An efficient design of intelligent network data plane," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur.)*. Anaheim, CA, USA: USENIX Association, 2023, pp. 6203–6220.

[29] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annu. ACM Symp. Theory Comput. (STOC)*, 1997, pp. 654–663.

[30] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Trans. Netw.*, vol. 6, no. 1, pp. 1–14, 1998.

[31] B. Arzani et al., "007: Democratically finding the cause of packet drops," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 419–435.

[32] (2011). *Pktgen-DPDK: Traffic Generator Powered by DPDK*. [Online]. Available: https://git.dpdk.org/apps/pktgen-dpdk/

[33] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, Nov. 2010, pp. 267–280.

[34] M. Dalton et al., "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 373–387.

[35] *Tcpreplay*. Accessed: Jul. 11, 2024. [Online]. Available: https://tcpreplay.appneta.com/wiki/captures.html

[36] J. Luxemburk, K. Hynek, T. Čejka, A. Lukačovič, and P. Šiška, "CESNET-QUIC22: A large one-month QUIC network traffic dataset from backbone lines," *Data Brief*, vol. 46, Feb. 2023, Art. no. 108888. https://www.sciencedirect.com/science/article/pii/S2352340923000069

[37] *IP Network Traffic Flows Labeled With 75 Apps*. Accessed: Jul. 12, 2024. [Online]. Available: https://www.kaggle.com/datasets/jsrojas/ip-network-traffic-flows-labeled-with-87-apps?resource=download

[38] S. Špaček, P. Velan, P. Čeleda, and D. Tovarňák, "Encrypted web traffic dataset: Event logs and packet traces," *Data Brief*, vol. 42, Jun. 2022, Art. no. 108188.

[39] *YouTube Traces From the Campus Network*. Accessed: May 8, 2024. [Online]. Available: https://traces.cs.umass.edu/index.php/Network/Network

[40] (2019). *Routeviews Prefix to as Mappings Dataset for IPv4 and IPv6*. [Online]. Available: https://www.caida.org/catalog/datasets/routeviews-prefix2as/

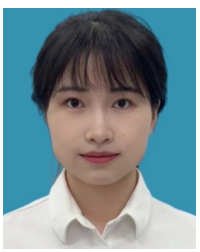
[41] H. Zhu, T. Wang, Y. Hong, D. R. Ports, A. Sivaraman, and X. Jin, "NetVRM: Virtual register memory for programmable networks," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 155–170.

[42] Y. Yuan et al., "Unlocking the power of inline floating-point operations on programmable switches," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 683–700.

[43] N. Sultana et al., "Flightplan: Dataplane disaggregation and placement for P4 programs," in *Proc. 18th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2021, pp. 571–592.

[44] J. Gao et al., "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 435–450.

[45] P. Bressana, N. Zilberman, D. Vucinic, and R. Soulé, "Trading latency for compute in the network," in *Proc. Workshop Netw. Appl. Integr./CoDesign*, Aug. 2020, pp. 35–40.



Tingting Xu (Student Member, IEEE) received the B.E. degree from Hunan University, Hunan, China, in 2019. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Nanjing University, under the supervision of Prof. Xiaoliang Wang. Her research interests include programmable networks, data center networks, and network function virtualization.

Xiaoliang Wang (Member, IEEE) is an Associate Professor with the Department of Computer Science and Technology, Nanjing University, China. He has published more than 30 technical articles at premium international journals and conferences, including IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE INFOCOM, ACM SIGCOMM, USENIX NSDI, FAST, and OSDI. His research interests include networking systems and distributed computing.

Chen Tian (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. He is a Professor with the State Key Laboratory for Novel Software Technology, Nanjing University, China. Previously, he was an Associate Professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology, China. From 2012 to 2013, he was a Post-Doctoral Researcher with the Department of Computer Science, Yale University. His research interests include data center networks, network function virtualization, distributed systems, internet streaming, and urban computing.

Yun Xiong is a highly experienced Network System Architect with nearly 20 years of expertise in researching and designing high-performance networking. He has was a Senior Expert and a Principal Architect with Huawei and Broadcom.

Baoliu Ye (Member, IEEE) received the Ph.D. degree in computer science from Nanjing University, China, in 2004. He is a Full Professor with the Department of Computer Science and Technology, Nanjing University. He was a Visiting Researcher with the University of Aizu, Japan, from March 2005 to July 2006; and the Dean of School of Computer and Information, Hohai University, since January 2018. His current research interests mainly include distributed systems, cloud computing, and wireless networks, with over 70 papers published in major conferences and journals. He served as the TPC Co-Chair for HotPOST12, Hot-POST11, and P2PNet10. He is the Regent of CCF and the Secretary-General of CCF Technical Committee of Distributed Computing and Systems.

Sanglu Lu (Member, IEEE) received the B.S., M.S., and Ph.D. degrees from Nanjing University in 1992, 1995, and 1997, respectively, all in computer science. She is currently a Professor with the Department of Computer Science and Technology and the Deputy Director of the State Key Laboratory for Novel Software Technology. She was the principle investigator of many national funding, including the National Key Research and Development Program of China; the National Natural Science Foundation of China; and the Key Research and Development Program of Jiangsu Province, China. Her research interests include distributed computing, pervasive computing, and wireless networks. She has published more than 100 papers in refereed journals and conferences in the above areas. She is a member of ACM.

Cam-Tu Nguyen received the bachelor's and master's degrees from Vietnam National University, Hanoi, in 2005 and 2008, respectively, and the Ph.D. degree in information science from Tohoku University, Japan, in 2011. From 2012 to 2015, she was with the LAMDA Group, Nanjing University, as a Post-Doctoral Researcher. From 2005 to 2017, she was a Lecturer with Vietnam National University. From 2017 to 2019, she was an Assistant Researcher with the Software School, Nanjing University, where she is currently an Associate Professor with AI School.