CLIP: Accelerating Features Deployment for Programmable Switch

Tingting Xu[1], Xiaoliang Wang[1], Chen Tian[1], Yun Xiong[2], Yun Lin[2], and Baoliu Ye[1]
[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, [2]HUAWEI, China
Email: xutingting@smail.nju.edu.cn, {waxili, tianchen, yebl}@nju.edu.cn, xiongyun1@huawei.com, yun.lin@hisilicon.com

***Abstract*—Cloud network serves a large number of tenants and a variety of applications. The continuously changing demands require a programmable data plane to achieve fast feature velocity. However, the years-long release cycle of traditional function-fixed switches can not meet this requirement. Emerging programmable switches provide the flexibility of packet processing without sacrificing hardware performance. Due to the trade-off between performance and flexibility, the current programmable switches make compromises in some aspects such as limited memory/computation resources, and lack of the capacity to realize complicated computation. The programmable switches can not satisfy the demand for network services and applications in production networks. We propose a framework that leverages host servers to extend the capability of network switches quickly, accelerates new feature deployment, and verifies new ideas in production networks. Specifically, to build the unified programmable data plane, we propose essential design and implementation challenges including a programming abstraction that allows automatically and effectively deploying network functions on switch and server clusters, allocating traffic to fully utilize the server resources, and supporting flexible scaling of the system. The quick deployment of self-defined functions in a realistic system has verified the feasibility and practicality of the proposed framework.**

I. Introduction

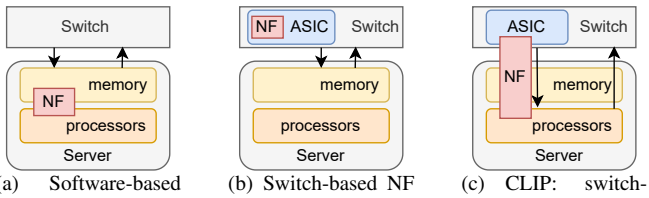
With the rapid growth of diverse applications and users, the cloud datacenter requires fast feature velocity to meet the customers' rising demands. Traditional high performance networking devices such as switches and routers process packets based on standardized protocols. However, the data plane algorithms usually can not be changed after the device has been built [1]. The update of these fixed-function devices is provided only by vendors, which is hard to meet the newly bloomed requirements with regard to the years-long release cycle of switch ASICs.

Programmable data planes allow users to define their own data plane algorithms, which offers great flexibility for network customization [1]–[5]. Compared with the fixed-function switches, the programmable switches allow parsing new fields of packets and deploying customized protocols in a short time. Cloud network operators have successfully deployed the programmable switches to improve the capacity of cloud network gateway [6]. However, it does not fill the gap between customers' increasing demands and limited hardware capabilities, such as the fixed capacity of memory, computation and processing resource, the inefficient computation capability [6]–[9]. For example, in order to count packets belonging to the same message on the switch, it is hard for the current programmable switch to understand the content in the packet headers ("send_first, send_middle, send_last") as well as other application-aware measurements. On the other hand, we also meet the requirement of verifying the effectiveness of new network functions, such as in-network computing/aggregation [10], in production networks. Therefore, in practice, we need a solution for quickly implementing users' requirements in production networks and rapid prototyping of new protocols.

To this end, we propose and implement CLIP, a general framework to overcome the limitation by on-loading parts of the new network feature to servers in datacenters, which offers scalable memory and computation capacity. Designing such a heterogeneous platform is challenging due to the difference between the technologies of devices. This difference is in expressiveness, flexibility and performance. The switch ASIC and server should interoperate and compensate each other for performance and flexibility to make seamless network functions deployment.

To address the challenges, we first propose a framework like the simplified remote procedure call (RPC), enabling remote processing through the switch data plane (§III-B). To simplify the programming, we design a top framework including a pre-processing module running on the switch, an on-loaded request handler running on servers, and a post-processing module running on the switch (§III-C). It helps users define the network features with the co-operations of heterogeneous devices. We implement a tool suit including a compiler and a controller. The compiler generates the executable file for the switch and remote processor by the user-defined program (app.p4c). The controller manages the content of the table using the remote runtime API [11]. We apply multiple servers to balance the traffic to maintain high throughput across concurrent high-speed channels/links. And a load balancer runs at the programmable switch to relieve the burden of the servers and maintain the scalability (§III-D).

We implement a prototype of CLIP and evaluate it in the testbed consisting of a Tofino-based programmable switch and servers (§V). We demonstrate the use of CLIP through the practical requirements in the production network, including the network function of state-heavy NAT and the new feature of overlay network measurement through active TCP retransmission detection. For example, the considered overlay network measurement requires identifying the retransmission packets of specified flows to determine the root cause of network delay jitter. The detection of retransmission packets is not supported by the current programmable switches due



(a) Software-based NF without switch cooperation

(b) Switch-based NF with server memory expansion

(c) CLIP: switch-based NF with server cooperation

Fig. 1. Network Function Deployment.

to the limited size of buffers and hard to maintain the state of TCP connections. However, by using CLIP we successfully deploy this feature in the experimental environment which has demonstrated and verified the effectiveness of designs.

The contributions of this paper are summarized as follows:

- We address the gap between the growing demands of customers and slow hardware function provision at the data plane. CLIP combines the programmable switches with a cluster of commodity servers to provide a flexible programmable data plane.
- CLIP introduces the framework to ease the deployment of new functions by effectively partitioning the network functions to switch and servers. It provides the programming interfaces to boost the deployment. It is scalable to the dynamic traffic by balancing the load to servers and maintaining the heavy tasks to switch.
- CLIP achieves fast feature velocity to satisfy the urgent demand of customers in the production networks within a short time. CLIP introduces a new approach to prototyping the verification of new network functions in the realistic system, which is important for the cloud network service providers in practice.

II. Background

In this section, we briefly summarize the hardware and software design of the programmable network architecture. And then we point out the motivation of this work, which focuses on enabling remote processing. Finally, we explain why the current solutions can not satisfy our requirements.

A. Software Network Function

Cloud providers have been deploying software-based network functions due to the flexibility, e.g., disaggregated software routers [12], [13] applied at the cloud gateway and the basic architecture is shown in Figure 1(a). They leverage software-based data plane programming models such as VPP [14], Click [15], FastClick [16], and make effort to customize network equipment without compromising performance.

Limitation. However, with regard to the switch device, the general-purpose CPU have remained an order of magnitude slower than switching chips (up to Tbps) without pipelining and parallelism. And at the price of achieving the same throughput, the commodity server has a relatively higher cost [7] than the switch.

B. The Rise of Programmable Switch

The programmable switch consists of a software-based control plane and a programmable data plane. The pipeline of a programmable data plane is usually divided into three parts, i.e. Parser, Match-Action and Deparser. The switch realizes packet processing starting from extracting the header fields and getting the metadata in the Parser. The header and its metadata are sent to Match-Action Units to be processed by lookup logic defined by users. At Deparser, the original packet body will be appended to the new header to assemble the complete packet.

Domain-specific language: The data planes can be defined by users through domain-specific language [2], [17], in a way that is similar to software development. Programming protocol-independent packet processors (P4) is currently the most widespread model for data plane programming, supported by various software- and hardware-based target platforms. It defines the following two programmable functionalities [2].

Match-action processing: Packet processing is abstracted as a generic pipeline consisting of match-action tables. When the packet traverses the pipeline, it matches table entries by keys and performs actions. The developer can define the packet process logic and the match-action table elements.

Stateful packet processing: Stateful memory can maintain state across packets, such as tables, registers, counters and meters. The registers can be read and updated during packet processing. Thus, the numerous novel in-network applications leverage the registers to cache data or perform primitive computations at data plane [9], [10], [18]–[20].

C. The Restrictions of Programmable Switch

In comparison with the traditional fixed-function switches, the programmable switches allow flexible packet processing in data plane, which means a shorter development cycle and fewer costs to develop new functions. It is hard to deploy the network function on the programmable switch in a large-scale production environment [6]–[8]. The widely concerned restrictions lie in limited memory, primitive computation, and narrow control channel.

First, the limitation of on-chip memory makes large-scale features hard to deploy. Stateful processing plays an important role in network systems, allowing applications to store and retrieve data across different packets. The packet header vector (PHV) carries the information from headers and metadata (temporary values or intermediate results) passing through the pipeline. The size of PHV about 200 bytes is enough for traditional protocol processing but becomes the restriction for the functions requiring more values or operating on the packet contents [8], [21]. The register arrays and match-action tables are arranged into physical stages of the pipeline. The amount of data stored in all stages is limited ranging from tens to hundreds of megabytes. Furthermore, resources are under-utilization for the placement constraints [8], [22]. The actual stored data is less than the specification.

Existing works [6], [7] make an effort to extend the programmable switch memory. As shown in Figure 1(b), TEA [7] extends one match-action table by requesting processing

entries from remote memory and does the exact match-action operation for all traffic at switch ASIC. They focus on network functions that require large-scale match-action tables but slightly concentrate on PHV, analyzable header length and register capacity.

Second, the computation ability throttles the more novel applications. A wide range of novel applications (e.g., stateful load balancing [9], in-network caching [19], in-network aggregation [10]) are enabled at the programmable networks to improve performance and reduce cost by offloading specific action from servers. However, the action execution supports only a small set of simple ALU 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 example, the machine learning acceleration [10], [18] uses the on-chip memory to aggregate (i.e., perform addition on) the float-point gradients but scarifies the precision. Besides the primitive operator limitation (same as action execution), using a register has to strictly obey the template of computation, stored value and arguments width, leading to application deployment hindrances.

Last, the programmable model limits communication-intensive functions. To guarantee the pipeline processing within nanoseconds, the programmable model extracts the match-action operations from network processing and leaves the relatively complex processes to the control plane. The communication between data plane and control plane is critical for function deployment. Taking the network measurement as an example, the more data collected and reported by the data plane, the more precise the network status analysis can be. However, the channel between ASIC and CPU processor is designed to process occasional control plane traffic (L2 address learning, etc.). Its bandwidth is fixed and lower than the ASIC's per-port, which can not support higher traffic rates without hardware modification [7]. The frequency of interaction through the narrow channel must be considered when deploying features.

D. Motivation

Though the programmable switch aims to network processing flexibility, there are restrictions for function deployments. We exploit hardware software co-design to achieve fast feature velocity for the switch. As shown in Figure 1(c), our basic prototype is a programmable switch connecting with a cluster of commodity servers. The cluster of commodity servers rather than a single server furnishes abundant memory and flexible computation resources. To achieve this design, we address the challenges as follows:

How cooperate heterogeneous processors to extend the programmable switches? Though the P4 programmable model can run at the servers [23], [24], it not only conducts less performance than the hardware switch but also limits programming flexibility. Different languages should be adopted to express the functions flexibly. Writing distributed programs manually is tedious and laborious because the developers have to explicitly define and manage the cooperation [24], [25].

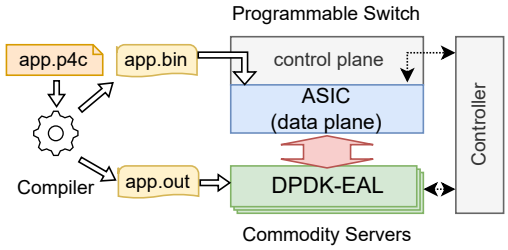


Fig. 2. The CLIP enables the cross-platform program runs at a programmable switch (PS) and several commodity servers (CS) processing plane. The red two-way arrow indicates the request data path.

How to access the memory and utilize the computation resources of the server without hardware modification? Existing switch ASICs lack abstractions for inter-program communication mechanisms such as remote procedure call (RPC) [26]. Requesting remote processing through the switch control plane causes an unpredictable performance problem due to limited channels. The request sending from switch ASIC is feasible but requires considering many details, such as intermediate value transmission, which affect the performance.

How do we partition the traffic among the switch and servers to compensate for performance and support scalability? There is a performance gap of multiple orders of magnitude between the programmable switch and the general-purposed processor for both latency and throughput. It must guarantee that remote processing is not the bottleneck to throttle the throughput of the switch. To address this problem, the straightforward way is sending only partial packet data or a fraction of packets to remote processing. Sending part of packet data is infeasible because the switch ASIC is hard to buffer large volume packets to wait for the remote processing. Dispatching a fraction of packets to servers should guarantee the consistency between flow and its state. The remote servers as the supplement of the switch should be scalable.

In a nutshell, enabling remote processing is not only sending and receiving packets between a switch and servers but also the fine-grained function partition, state synchronization and the trade-off between performance and flexibility.

III. CLIP Design

To accelerate features deployed, we propose CLIP, a cross-platform system to cooperate with the switch and commodity servers. Based on CLIP, the application can be deployed at the programmable switch with the help of remote servers.

A. System Overview

The CLIP architecture is illustrated in Figure 2. We utilize the commodity servers (CS) directly connected to programmable switches (PS) as a basic topology. To simplify cross-platform programming, users can define their network function at a user-defined program (app.p4c) using the CLIP framework. According to the user-defined program, the CLIP compiler generates the executable files for switch ASIC

(app.bin) and remote processors (app.out), respectively. The app.bin is installed through the switch control plane while the app.out is installed into servers. The controller initializes the device to prepare the runtime environment and monitor the device status. At runtime, this controller manages the content of the switch table using the remote P4Runtime API [11].

To minimize the cost of remote processing, CLIP enables the switch to directly request processing like RPC through its data plane without the involvement of the control plane. Though the procedure is defined, cross-platform programming is still hard for task orchestration such as route isolation for normal traffic and request packets. The compiler further generates the base control flow, request/response packet, and the forwarding module deployed at the switch. Besides, to prevent the server throttle the performance, we design a load balancer deployed at switch to relieve the burden of servers.

B. Remote Resource Accessing

The current switch ASIC communicates with control plane by redirecting packets to the switch CPU or reporting a digest with little information. After receiving packets or digests, the switch control plane can request remote processing using the traditional RPC mechanism. It can draw support from the remote resource while facing the following problems:

Unpredictable latency. The switch control plane does not directly connect to a server for isolation demand: The switch CPU and ASIC access the network through different links and devices to isolate the control plane network from the data plane network for device management and maintenance. The switch control plane has a management port through which managers can access this device remotely. For the control plane, only this management port can send and receive the RPC messages. However, the RPC message has to traverse multiple hops network before arriving at the server. Furthermore, other control messages that maintain the network running also share the links with RPC messages. The path length and network status are out of control leading to the latency being unpredictable.

Throughput bottleneck. Channels bandwidth (Switch-ASIC to Switch-CPU and Switch-CPU to Server-CPU) limits the processing efficiency. The channel between switch-ASIC and switch-CPU is designed to process occasional control plane traffic (L2 address learning, etc.). Its bandwidth is fixed at about one hundred Gbps while the switch ASIC can process several Tbps traffic. The channel bandwidth from Switch-CPU to Server-CPU is also one hundred Gbps. Once the traffic exceeds the capacity of the channel, it becomes the bottleneck of remote processing. Furthermore, for packets required to be processed by remote processors (e.g., the length of requisite data exceeds the PHV capacity), the packet has to be encapsulated into the RPC message as neither the control plane nor the data plane of the switch can store a large number of packets. The large RPC message transmission exacerbates the throughput problem.

To address the above problems, CLIP enables the communication between switch ASIC and server from data plane channel. Inspired by RPC mechanism, the switch ASIC directly sends the original packet with the request parameters (called request packet) and receives the modified packet with the returned value (called response packet). The switch data plane takes over the request packet encapsulation, sending and the response packet reception, parsing, etc. The request/response packet goes through the data plane channel between switch-ASIC and server-CPU. The overall procedure does not require the involvement of switch control plane.

C. Network Features Definition and Deployment

Based on the above capability, we consider how to arrange the new feature at heterogeneous devices. Several questions arise like how to partition the NF, what parts of NF should run at remote processor or local ASIC, and how to fully utilize the resource at both PS and CS. It is hard to build a universal model for diverse NFs [1], [8]. According to the performance and flexibility of heterogeneous PS and CS frameworks, we comprehend two principles: 1) PS is efficient at packet match-action, and 2) CS has rich resources in terms of memory and computation, which can realize complicated computation and resource-consumption tasks. To this end, we can partition the function into three parts, i.e., a *pre-processing* partition, a *request handler* and a *post-processing* partition. To fully utilize the capacity of hybrid devices, the top framework definition is shown as follows.

```
1  /* Top layer control flow */
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 *pipeline* in line 2 defines the control flow of a packet processing where the packet header and intermediate values (metadata) as the input and output parameters. To make full use of remote resource flexibility, we allow P4 language embedding with C language to define the cross-platform program with the suffix p4c. The pre_processing and post_processing in line 3 are P4-defined parts running at the switch data plane while the remote_handler in line 9 is a C-defined part at general-purposed processors. The *request* and *response* defined in line 4 are two parameter lists that can be quoted in remote_handler and post_processing, respectively. They indicate the cross-platform communication data for packet processing. To reduce the burden of the server, the *flight* flag can be assigned at pre_processing in line 7 to filter packets requiring remote processing. An example of the state-heavy SNAT [7] using this framework is shown in Appendix A.

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 can not suspend and wait for the remote processing in the middle of the pipeline. To make this framework work seamlessly, there are three parts should be designed:

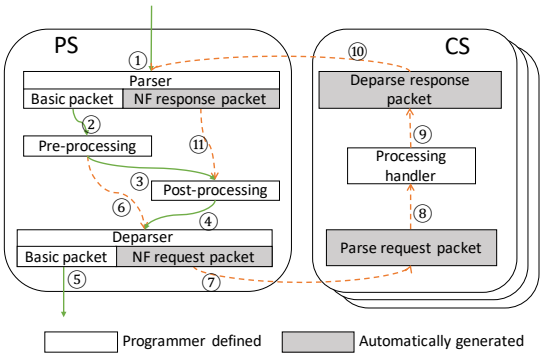


Fig. 3. The processing path when the network feature deployed. The green solid line is the fast path and the orange dotted line is the slow path.

- The control flow. The P4 program has to distinguish the response packets from the original traffic, and then decide that executes the pre-processing, post-processing, or requesting remote process. The *flight* flag also influences the control flow. If the control flow is defined carelessly, it may form a path loop between the switch and servers.
- Parameters transmission. It generates request/response packet headers that store and transmit the parameters and intermediate values. The header format differs for diverse network functions, affecting remote processing efficiency.
- The forwarding module. It provides network traffic routing and server selection, and isolates the routes between the regular traffic and the request packets. In addition, it should guarantee the consistency of flow and its state when the server group scales.

Writing this cross-platform program manually is tedious and error-prone, so we design a compiler to automatically generate the executable programs under the top framework. The control flow is shown in Figure 3. It distinguishes the original packets and request/response packets and decides on incoming packet processing. The *flight* flag and packet header type decide for each packet which path it goes through. The fast path (①∼⑤) is the packet path only for PS local processing, while the slow path (①,②,⑥∼⑪,④,⑤) is for remote processing.

The parameters of the remote request are piggybacked by the packet. The parameters are assembled as the Ethernet packet payload like the IP header and are identified by the Ether_type field. The load_type value is copied from the original Ether_type to identify the original Ethernet payload when removing the request header. The request header also carries the options to accelerate the handler processing. For example, the ”parsed_len” indicates the switch has parsed the ”parsed_len” bytes, and the server can parse the header starting from ”parsed_len”-th byte. In addition, if the response header length differs from the request header, the remote processing has to do an extra packet copy to scale this packet buffer, which slows down the processing. We align the request and response header to avoid this extra packet copy.

The key design is the forwarding module. The regular traffic out of CLIP may adopt L3 routing or L2 forwarding. And it should select a server for the request packets. Therefore, the request packet forwarding should be isolated from regular packets. We leverage the Equal-Cost Multi-Path (ECMP) group of L3 Route and add the servers as members of a specific ECMP group. For packets with the *flight* flag, we assign the group id for them before routing so that the packet can be processed at a CS. The scaling problem is explained in the next section.

D. Auto-scaling of Server Group

To deal with the dynamic workloads, we use multiple servers to improve the throughput of the platform. To effectively utilize the server resources, we address two problems: (1) How to relieve the processing burden of servers? (2) How to deal with the traffic if current servers can not process it?

Strawman solution. The straightforward way is introducing a load balancer at the switch to distribute the traffic to servers. It should keep the affinity between packets and their states if the server group member changes. Existing stateful load balance mechanisms [9] enable connection consistency at the programmable switch by recording the per <connection, server ID> mapping, which offers effective load balance. Other conventional distributed hashing schemes such as consistent hashing [27] and rendezvous hashing [28] partition the hash tables and maintain numerous <bucket range, server ID> mappings, which consume non-negligible on-chip memory space. Besides, though the flow-based traffic is distributed evenly across servers (i.e., each server processes the same number of flows), the skewing flows with a large number of packets lead to server cores being unbalanced [6].

Relieving burden of servers by automatic flow replacement. Match-action table plays an essential role in the P4-defined program and is the most precious resource. We construct the *mirrored table* generated according to the switch match-action table (called the original table) structure. The mirrored table running at servers has the same function as the original one to extend the capacity[1]. For the functions that only require memory expansion (i.e., part traffic without the requirement of remote processing), CLIP leverages the switch capacity to evict the heavy-hitter traffic from servers to balance the loads. All table entries are distributed among mirrored tables by default. As all traffic goes through the switch, we implement a small count-min sketch at the switch to count packets for arrived flows and report the top-N flows regularly. The top-N flows are the most resource-consuming for the general-purposed processors, so we leverage the original table running at the switch to handle them, reducing the burden on the mirrored table. The corresponding table entries of selected top-N flows are pushed from the mirrored table to the original one to replace the aged flow entries.

Server group scaling without traffic halting. For the increasing workloads requiring remote processing, the fixed-size server group may not provide enough resources to process all packets. We propose the dynamic adjustment of server

[1]The CLIP compiler automatically generates the mirrored table according to the user-defined P4 table. The CLIP controller manages two tables using the Runtime API as P4 defined and identifies which table entries are loaded by the original or mirrored table, respectively.

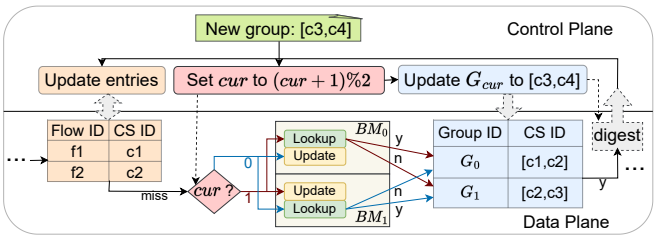


Fig. 4. Load balancing for keeping affinity between flows and their states.

group membership to dynamically adjust the capacity. In detail, we dispatch new flows into light-load servers while the existing flows into their original server. With the requirement fluctuation, we can increase or decrease the member of the CS group to adjust system capacity without stopping traffic.

The pivotal problem is how to guarantee the consistency of flow and its states when the group member changes. A flow-based ECMP is implemented to dispatch traffic among the server group and the flow states are produced and maintained by the server handling the first packet. Changing group member leads to the flow re-dispatching and then the flow-state mismatching. Recording the destination for each flow is memory-consuming. And it is complex for state synchronization through migrating states among servers, e.g., stopping or buffering traffic until the states migrating finish. Notice that if the subsequent packets of a flow can find their first packet destination, we can avoid state synchronization. Therefore, we use a small on-chip memory to maintain the group member for flows arriving at different times like a snapshot. The cost of creating a snapshot for each flow is non-trivial, so we picture the group membership and the arriving flows before the group updating. The procedure is illustrated as follows. 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 ECMPed 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 leverage the bloom filter (BM) to record the existing flows and identify the new flows[2]. But only one BM can not support flow identification for a period of time as explained in Appendix B. We consider multiplexing the BM_0 and BM_1 to alternately identify the flow arrived over a period of time. As shown in Figure 4, 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 this procedure:

- *update*: altering the flow corresponding bits as 1.
- *lookup*: looking up the flow corresponding bits to identify whether a packet belongs to an existing flow. If does, it returns y. Otherwise, it returns n.

[2]BM 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.

When the packet misses the <Flow ID, CS ID> mapping table, it 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. However, before updating the cur from 0 to 1, BM_1 should be cleared to record the arriving flows for the next interval. And the destination of flows that arrived before the last cur updating is lost if BM_1 is cleared. To solve that, for the traffic not to rely on remote processing, we push them to switch by inserting corresponding entries into the switch. For the non-offloaded traffic, we buffer the <Flow ID, CS ID> at the switch data plane before the BM is cleared. There is a time interval to record the unexpired entries so the number of recorded items is reduced.

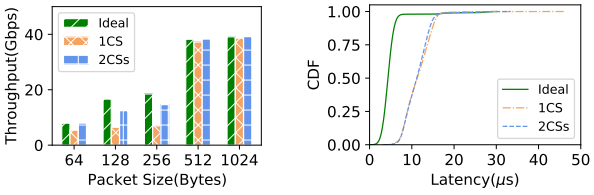
IV. Applications

Through two examples we demonstrate the benefits of using CLIP for the operation of cloud networks to quickly respond to customers' requests.

State-heavy Source Network Address Translation (SNAT). It is a network function where the internal traffic shares a public IP to access the external network. When receiving internal traffic, SNAT allocates a source IP address and port pair and rewrites the packet header. It remembers the mapping of the existing connection to the pair and forwards the subsequent packets using this mapping. This state-heavy network function requires much memory to store the address translation mapping. We leverage the CS memory to supply sufficient space to store NAT mappings. Due to the match-action table only can be updated by the control plane and the CPU-ASIC channel bottleneck as mentioned in section §II, we use the remote handler to perform address-port pair allocation. Thus, the mappings are generated at CS and installed at CS or PS according to the min-sketch selection.

Overlay Network TCP Retransmission Detection (TRD). When network congestion occurs, the TCP packet may be dropped, leading to retransmission. Thus, TCP retransmission can signal that overlay network gray failure happens as it is sensitive to the changing of the network environment. It is hard for the switch data plane to maintain the states of TCP connection, e.g., established or disconnected status. Besides, the sequence number of TCP flow should be maintained and updated at the data plane to probe the retransmission even in time. The *register* array supports this operation but is indexed by an integer rather than the connection id. The hash computation can translate the connection id to an integer but introduces mistakes as the hash collision leads to allocation overlap of different connections.

We leverage CS to maintain the states and allocate a unique register index for each connection. We assign or release the register index and update the connection status by forwarding a request that carries a packet with TCP control flags (SYN, FIN, etc.) via the switch ASIC. The response takes the allocated index and connection status back to the switch to update the corresponding register.



(a) Average throughput (b) Latency distribution

Fig. 5. Performance without application deployed. The "Ideal" shows all traffic traverse the switch.

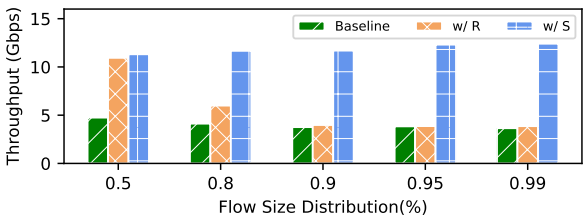


Fig. 6. Performance improvement with offloading under the randomly selecting and count-min sketch.

V. Evaluation

In this session, we evaluate CLIP on the testbed to answer the following questions.

(1) What is the overhead of the remote processors? Do we maintain throughput by the load balancing among multiple servers? (§V-B)

(2) How to use the CLIP framework to deploy new features? Compared with the software implementation, how much the performance benefits of CLIP-based function? (§V-C)

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

A. Experiment Setup.

Testbed setup. The testbed consists of 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 40Gbps, by aggregating four 10Gbps network channels. Each server has Intel Xeon Silver 4110 CPUs (2.10GHz, 8 cores) and a Mellanox ConnectX-5 NIC. Servers run Ubuntu 18.04 with Linux kernel version 4.15. All three servers are connected by the switch via 40 Gbps links. We dedicate two servers to the remote servers and run DPDK version 21.05. DPDK-pktgen [29] is deployed at one server whose one port binds a core to generate the packets and the other binds a core to receive.

Work loads. We use two public realistic packet traces and synthetic traffic. One real trace is from an existing data center network [30]. Though its IP addresses are anonymous and payloads are removed, its flow distribution is available for evaluation. And the other is the web search trace [31] including PCAP files around 100G. Its payload is available for trace replaying. The synthetic traces follow the Zipf distribution regarding the number of packets per flow based on the datacenter measurement works [30], [32]. We inject those traces using DPDK-pktgen at the packet generator.

B. Microbenchmarks.

Forwarding Capacity. The general-purposed processors are introduced to process part of traffic to improve flexibility. Compared with traditional RPC through the control plane, forwarding through the data path has predictable latency and avoids being limited by the channel bandwidth between the switch control plane and the data plane. Notice that the remote processor introduces extra latency and lower throughput than the switch. Multiple CSs are adopted to improve performance. We evaluate the throughput and latency of the hardware-software cooperation platform. We first force all packets to traverse the switch only, which indicates the performance upper bond or ideal situation (packet generating rate up to upper bound). As shown in Figure 5(a), for packets from 512 to 1024 bytes, one CS can achieve performance near the upper bound since software-based processing is packet-based. It can process a fixed number of packets per second. With the packet size increasing, we achieve higher throughput (Gbps). For packets of large size, the bandwidth is the bottleneck instead of the CPU. For smaller packets, the CPU becomes the bottleneck that impacts the processing rate of both the generator and remote servers. The parallel CSs can improve the throughput of small packets.

Figure 5(b) demonstrates the latency distribution. We inject 10000 packets of random size to measure the forwarding latency. With CSs involved, 90% packets can traverse the platform within 16μs. The remote processing introduces an extra latency of about 9μs for 90% packets. Prior work TEA [7] costs about 2μs to expand ASIC memory. Compared with it, the extra 9μs latency lets us expand not only the memory but also the computation resource.

Flow selection efficiency. The heavy-hitter traffic rapidly throttles the performance. CLIP automatically mitigates the heavy traffic from servers to reduce the servers' burden. We evaluate the effectiveness of the operation by generating packets of 64B with different flow size distributions. α indicates traffic skewness following the Zipf distribution.

Figure 6 indicates the throughput under two offloaded flow selection schemes: random (w/ R) and count-min sketch (w/ S) with different skewness. The baseline (non-offloading) shows that more skewing traffic leads to lower throughput as the load imbalance occurs frequently. The randomly selected offloading helps to improve the performance at less skewing ($\alpha = 0.5$ and $\alpha = 0.8$). But it is hard to choose the accurate heavy loads with regard to serious skewing traffic ($\alpha > 0.9$). The count-min sketch selects heavy loads more efficiently and is able to improve the throughput by 2.38x - 3.39x.

Capacity adjustment. 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, start and stop time to simulate real-world traffic. As shown in Figure 7, 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 stop sending set-1 at around 31 sec.

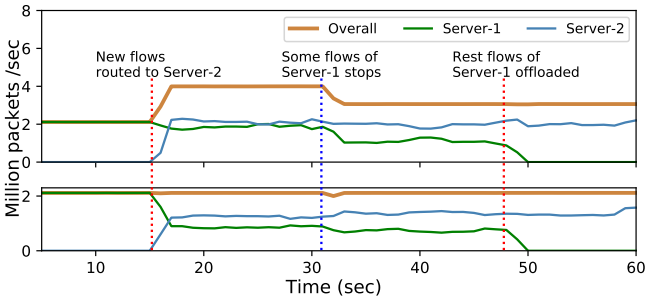


Fig. 7. Throughput changes as the CS group membership changes event. The time points of the red dotted line indicate flow scheduling orders from the controller. The blue one shows the flow changing orders from the traffic generator.

TABLE I
THE AVERAGE LATENCY OF NETWORK FEATURE.

Network Feat.	CLIP w/o offload	CLIP w/ offload	Software-based
SNAT	26.42	8.84	23.16

As shown in Figure 7, 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 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 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.

C. Network Features Deployment

CLIP can accelerate existing software middleboxes such as NAT, Load Balancer, and Firewall. Those middleboxes have a similar processing model: applying for entries from the control plane and doing 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 performance as a representative and compare it with the current software design.

Feature performance. We run a SNAT implemented using FastClick [16] at one CS. For a fair comparison, both SNATs run at the same server and use one core. TCP retransmission detection is a new feature without the public software realization as we know. We evaluate network feature performance in terms of throughput and latency. Table I shows the latency of CLIP with and without count-min sketch offloading. Compared to software-based SNAT, the latency of CLIP-based has reduced near to 62% with offloading.

We replay two real packet traces from data center and a websearch application with our best transmission rate to evaluate the throughput. The payload and IP addresses of the data center trace have been anonymized. We utilize the traffic header and append payload for the packet to 64 bytes to simulate the traffic distribution of data center. The websearch trace has anonymous IP addresses but a valid payload, which helps inject the actual packets for evaluation. And it has

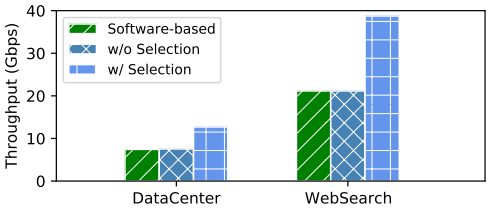


Fig. 8. Throughput with feature deployed.

TABLE II
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%

greater throughput than one from data center for a bigger packet size. As shown in Figure 8, CLIP achieves near to 2x throughput with offloading. And it is the potential to realize higher performance if the traffic generator does not limit the sending rate.

D. Resource Consumption

We evaluate how much ASIC resource is consumed only by CLIP based on the P4 compiler's output. Table II 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 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 about the hash bit is under 2%.

VI. LIMITATIONS AND DISCUSSION

Availability. The potential concerns are server failures or link saturation that may lead to remote request loss. The next step is studying a well-designed mechanism to maintain availability. For example, the server triggers flow control and back-pressure to guarantee a lossless network when the queues are near saturation.

Cost of Remote Processing. The remote computation cost comes from the latency and available port count. The latency is irreparable as part of the packets traverse the extra two hops. We argue the gained feature velocity is beneficial as the trade-off of latency. The overall available port count decreases as small portions connect the servers. We can scale in to leave more ports by reducing the servers in section §III-D.

Plugging Various Remote Resources. Although the current design of CLIP offers the programmable switch the ability of remote processing at connecting servers, we can extend it to support other remote programmable resources like FPGAs. But more considerations in the production environment such as runtime environment and hardware programming expertise of FPGAs should be taken, which lost part of the agility and generality of CLIP.

VII. Related Works

Network Function Offloading. Virtual Network functions (VNFs) were originally proposed to enhance the scalability and availability of traditionally standalone hardware appliances like [33]–[35]. The hardware middleboxes were partially or fully replaced by the server cluster [13], [36], [37]. To serve the cloud-scale volume of traffic, NFs have been accelerated using programmable switches, FPGAs, or smart NICs to reduce the cost of CPU-based design [6]–[8], [38].

Programmable Dataplane Limitations. Improving the network's programmability is the cloud network trend. One typical case is the introduction of a programmable switch which enhance the adaptability of commodity switches [13], [32]. However, the specific hardware makes a trade-off among performance, resources and flexibility. It leads to restrictions affecting the deployment of functions in the production networks. Some works [6], [7], [39] extend and excavate the match-action tables but slightly concentrate on PHV, analyzable header length and register capacity. P4All [22] defines elastic data structures and interprets them into the native P4 program to improve expressiveness. But primitive computation limitations, such as float computation, exist. Other works concentrate on other restrictions. For example, IPSA [21] intends to achieve online updating for the programmable switch by rebuilding its architecture.

Heterogeneous platforms cooperation. Considering combining the strengths of different programmable hardware, some works [8], [24], [25], [40] design the architecture, language, compiler, or toolchains to help cross-platform cooperation. Gallium [8] translates the existing C++/C-defined middleboxes into separating programs running in the programmable switch and the server. It focuses less on new feature deployment limitations and the performance gap of heterogeneous platforms. Flightplan [24] aims to automatically disaggregate a P4 Program into ASIC, FPGA and CPU. Lyra [25] designs the cross-platform language and compiler for heterogeneous ASICs of the switch. Both limit the function expressiveness in P4 or NLP [17] defined behaviors.

VIII. Conclusion

In this paper, we explain our practice to enable data plane programmability. Motivated by the flexibility of software network functions, we propose to use the common server cluster to extend the capacity of network switches quickly. Given the requirements of network customers, which can not be processed by current commodity switches or programmable switches with limited resources, we forward the packets to server clusters with packet processing functionality. By so doing, the proposed framework is able to provide a rich-resource environment. To meet the performance and complexity challenges, we introduce balancing the traffic to multiple servers and carefully dividing the network functions to fully utilize the resource in both hardware switches and software functions on servers. Finally, we demonstrate some applications of leveraging a programmable data plane.

Acknowledgements

This work was partially supported by National Key R&D Program of China No.2022YFB2702803, National Natural Science Foundation of China under Grant Nos. 62172204, 61832005, the Key R&D Program of Jiangsu Province under Grant BE2020001-3. Collaborative Innovation Center of Novel Software Technology and Industrialization.

Appendix

A. Example

We explain our design by taking SNAT as an example. Some details are omitted for simplicity. The C/C++ defined and CLIP-defined SNATs are shown as following, 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_flow_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 is same as section III-C*/
control pipeline(inout header hdr, inout metadata md){ ..}
```

B. New Flows Identification

Suppose a sequence of packets $< p_1^a, p_2^b, p_3^a, p_4^b, p_5^c ..., p_s^m >$ arrive at time t_1, ..., t_s, where p_s^m indicates the arrival of s-th packet in flow $f^m (m \in \{a, b, c\})$. The group membership changes at $t'(t_1 < t' < t_2)$. The BM executes *update* operation at t_1 and, therefore, records f^a. At time t_2 and t_3 after t', it performs *lookup* operation that identifies the p_2^b belongs to a new flow while the p_3^a belongs to an existing flow at t'. Notice that from time t', the BM should not execute *update* operation. If does, a mistake is made, because the latter arrived p_4^b will be identified as a packet belonging to existing flow for time t'. If the BM does not execute *update* operation after t', the records of new flows (f^b and f^c) are lost, leading to flow identification being invalid when the next membership changes.

REFERENCES

[1] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2016.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, 2014.

[3] R. Giladi, *Network processors: architecture, programming, and implementation*. Morgan Kaufmann, 2008.

[4] Tofino, "Intel programmable ethernet switch products," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html, 2021.

[5] "Trident4 / BCM56880 series, high-capacity strataxgs trident 4 Ethernet switch series," 2020. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series?_ga=2.101838278.670967709.1597289176-917906606.1597289176

[6] T. Pan, N. Yu, C. Jia, J. Pi, L. Xu, Y. Qiao, Z. Li, K. Liu, J. Lu, J. Lu *et al.*, "Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2021.

[7] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "Tea: Enabling state-intensive network functions on programmable switches," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020.

[8] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020.

[9] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[10] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "ATP: In-network aggregation for multi-tenant learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[11] T. P. A. W. Group, "P4runtime specification," https://p4.org/p4-spec/p4runtime/v1.0.0/P4Runtime-Spec.html, 2019.

[12] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain *et al.*, "Taking the edge off with espresso: Scale, reliability and programmability for global internet peering," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

[13] H. Shao, X. Wang, Y. Lu, Y. Yu, S. Zheng, and Y. Zhao, "Accessing cloud with disaggregated Software-Defined router," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[14] FD.io, "VPP," https://wiki.fd.io/view/VPP.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[16] C. S. Barbette Tom and M. Laurent, "Fast userspace packet processing," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2015.

[17] "Github: Npl-spec," 2021. [Online]. Available: https://github.com/nplang/NPL-Spec

[18] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," *arXiv preprint arXiv:1903.06701*, 2019.

[19] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles(SOSP)*, 2017.

[20] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[21] Y. Feng, Z. Chen, H. Song, W. Xu, J. Li, Z. Zhang, T. Yun, Y. Wan, and B. Liu, "Enabling in-situ programmability in network data plane: From architecture to language," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[22] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker, "Modular switch programming under resource constraints," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[23] P. Voros, D. Horpacsi, R. Kitlei, D. Lesko, M. Tejfel, and S. Laki, "T4p4s: A target-independent compiler for protocol-independent packet processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2018.

[24] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo, "Flightplan: Dataplane disaggregation and placement for p4 programs," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[25] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics," *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020.

[26] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, 1984.

[27] 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," 1997.

[28] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Trans. Netw.*, 1998.

[29] "pktgen-dpdk: Traffic generator powered by dpdk." https://git.dpdk.org/apps/pktgen-dpdk/, 2011.

[30] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010.

[31] B. Spa, B. Pv, P. B, and B. Dt, "Encrypted web traffic dataset: Event logs and packet traces," *Data in Brief*, 2022.

[32] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer *et al.*, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[33] M. Mahalingam, D. G. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks." *RFC*, 2014.

[34] D. Katz, D. Ward *et al.*, "Bidirectional forwarding detection (BFD)," RFC 5880, June, 2010.

[35] S. Hanks, D. Meyer, D. Farinacci, and P. Traina, "Generic routing encapsulation (GRE)," 2000.

[36] M. T. Arashloo, P. Shirshov, R. Gandhi, G. Lu, L. Yuan, and J. Rexford, "A scalable VPN gateway for multi-tenant cloud services," *ACM SIGCOMM Computer Communication Review*, vol. 48, no. 1, pp. 49–55, 2018.

[37] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[38] e. a. Kim, Daehyeok, "Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems," *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[39] H. Zhu, T. Wang, Y. Hong, D. R. Ports, A. Sivaraman, and X. Jin, "NetVRM: Virtual register memory for programmable networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022, pp. 155–170.

[40] P. Bressana, N. Zilberman, D. Vucinic, and R. Soulé, "Trading latency for compute in the network," in *Proceedings of the Workshop on Network Application Integration/CoDesign*, 2020.