Enabling In-Network Acceleration over the Cloud

Hao Wang[1], Decang Sun[1], Jinbin Hu[1 2], Kai Chen[1]
[1]Hong Kong University of Science and Technology
[2]Changsha University of Science and Technology

***Abstract*—Offloading computing and storage to programmable switches, or in-network acceleration (INA), is a recent wisdom to speed up distributed applications. Researchers have proposed a variety of tailored INA solutions for separate applications with different data-plane layouts and network protocols. However, as hardware resource of programmable switches is limited, it is hard to integrate all these INA solutions simultaneously. Consequently, specialized INA techniques cannot realize full potential on the cloud, which needs to support various applications and requires concurrent access by multi-tenants.**

To enable on-demand INA service on the cloud, we present a generic INA framework called INAaaS (INA as a Service). At its core, INAaaS provides a universal INA interface in network and offers application-specific adapters on end-hosts. It further addresses the isolation problem of different applications, and guarantees the reliability and correctness. Our evaluation shows that INAaaS effectively improves the performance of various cloud applications, competing with the specific solutions.

I. Introduction

Cloud computing services support a wide range of distributed applications, e.g., distributed machine learning, MapReduce, and distributed storage. These applications rely on frequent communication among distributed nodes, which can generate significant network traffic. As a result, the network can become a bottleneck that limits the performance and scalability of these applications. Meanwhile, the communication traffic of these distributed applications puts a strain on the data center network and hinders other network applications.

To accelerate these applications, an emerging methodology is to offload operations and functions from end-hosts to network devices, e.g., programmable switches and smart NICs. Literature shows that such in-network acceleration (INA) technique can alleviate end-host overhead while increasing throughput and reducing latency. Specific INA implementations have been proposed for many applications, e.g., key-value stores [14], distributed machine learning [3], distributed storage [23], lock managers [32], consensus [8] and coordination [13]. Recent works also pay attention on providing generic INA for a group of applications, e.g., RPC system [34] and aggregation for key-value streams [11].

However, the current INA solutions still have some way to go before they can be directly deployed on the cloud. Specifically, a cloud INA service needs to address the following three issues:

- *One in-network hardware design for all applications.* INA designs involve programming and configuring on network devices. Such hardware devices need to be reset when running a new program, e.g., FPGA and programmable switch, which may cause several minutes interruption and impact other network applications [35]. Meanwhile, the resource limitations of switches hinder the simultaneous deployment of all application programs. Therefore, to facilitate concurrent operation of different applications, we need a uniformed INA design. Moreover, cloud users are not experts in network devices and transport protocols, we need to conceal the details of INA and provide users with transparent services.
- *Isolation and security among multi-tenants.* For cloud service, network devices are shared by multi-tenants, and INA applications involve accessing and modifying the memory in the network. Therefore, our design needs to consider security and isolation among different tenants.
- *INA service delivery and consumption modeling.* Cloud service providers charge customers based on a pay-per-use or subscription model. For INA services, we need to design a price model for the scarce network hardware resources and provide differentiated services.

The challenge of implementing a generic in-network hardware design lies in the limited resources of network devices. For example, a typical programmable switch [5] has only 12 or 20 stages per pipeline, with each stage containing only 4 ALUs. The entire switch data plane can only provide storage at the Mega-Bytes level. To address this issue, INAaaS abstracts the core operations of applications into three categories: 1) read/query, which allows applications to retrieve previously stored items; 2) write/store, which enables applications to record current items; and 3) compute/reduce, which allows applications to perform computation. We place these core operations on the programmable switch, and for specific applications, we leverage these core operations to implement the corresponding adapters on the end-host, which conceals the in-network implementation details from tenants.

To ensure isolation in a multi-tenant environment, we assign a unique identifier (uid) to each tenant. When accessing the switch memory, the switch logic ensures that a register in use cannot be accessed or modified by packets from other uids.

We also design an INA delivery and consumption model. Using in-network storing (register) and computing (ALU) provided by INAaaS, users can achieve lower latency and higher application throughput. We model the pricing for the use of registers and ALUs. We also analyze the relationship between switch resource usage and the acceleration of latency and throughput for various applications.

The remaining of the paper is structured as follows. Section 2 provides the background for our work. Section 3 motivates our work to provide a cloud-based INA solution. In Section

Application	Solution	Device
DML (dense model)	ATP [3]	prog. switch
DML (sparse model)	Libra [22]	prog. switch
reinforcement learning	iSwitch [20]	FPGA
distributed storage	NetEC [23]	prog. switch
key-value store	NetCache [14]	prog. switch
lock managers	NetLock [32]	prog. switch
consensus system	NetPaxos [8]	SDN switch
coordination system	NetChain [13]	prog. switch
network monitor	ElasticSketch [30]	all

Table I: Taxonomy of in-network accelerated applications. (prog. switch is short for programmable switch.)

4, we present the design of our proposed system, INAaaS. In Section 5, we evaluate the performance of INAaaS. Section 7 provides an overview of related work, and we conclude our paper with a summary in Section 8.

II. Background

In this section, we introduce the background of In-Network Acceleration (INA), cloud service and programmable switch. We further discuss the motivation for designing a cloud-based generic INA solution.

A. In-Network Acceleration

Offloading application functions from end-hosts to network devices, or in-network acceleration[1] is driven by the emergence of programmable network devices and the rising workload of data centers [21], [18], [19], [28]. INA offers several benefits including 1) lowering the latency for application queries, 2) decreasing the volume of traffic within the network, and 3) reducing the processing burden on end-servers. Existing work has proposed customized INA solutions for many applications. Table I lists these applications and the solutions.

The first three applications are related to Distributed Machine Learning (DML). The basic acceleration mechanism is that workers share their model updates (gradients) via the network, instead of aggregating gradients at the end host, an aggregation primitive sums the updates in the network and only distributes the result. Such acceleration is also called in-network aggregation. The difference among the three applications is that for the dense model, each worker transmits all the gradients on each iteration [3]. For the sparse model, since each iteration has only a small number of gradients with non-zero values, only some of the values are transmitted each time and each worker does not necessarily transmit the same position [22]. For reinforcement learning, it has more iterations, but with less gradients of each iteration [20].

Distributed storage system [23] leverages erasure code to to provide fault tolerance and data redundancy. In the event of a node failure or data loss, the system can reconstruct the original data from the remaining data. This process creates a many-to-one aggregation pattern, thus is applicable to INA.

[1]A lot of work uses the term In-Network Computation (INC). Given that our scope also includes pure cache applications, in this paper, we use the term In-Network Acceleration (INA) instead.

Key-value store (KV-store) is commonly used in data centers to store and retrieve data quickly and efficiently. It is often used for caching frequently accessed data, storing user preferences, and managing session data. KV-store is sensitive to query latency. Recent work [14] uses switch to cache the data, which achieves high throughput and low latency.

In distributed systems, a lock manager is responsible for managing locks on shared resources to ensure the consistency and correctness of operations. Fast acquiring and releasing lock are crucial to such systems. NetLock [32] builds a centralized lock manager based on programmable switches.

Paxos is a consensus protocol used in distributed systems to achieve agreement among a group of nodes on a value or a sequence of values. NetPaxos [8] moves consensus logic to SDN switches, which increases the messaging throughput.

Coordination services refer to a set of distributed systems that provide a way for multiple nodes in a distributed system to coordinate with each other, i.e., distributed locking, leader election, and distributed transactions. NetChain [13] leverages programmable switches to provide scale-free sub-RTT coordination within the network.

A network monitor captures and analyzes traffic, giving administrators insight for troubleshooting, security threat identification, and performance optimization. Elastic Sketch [30] provides measurement tasks like flow size estimation and heavy hitter detection on the data plane.

In addition to the customized INA solutions for specific applications mentioned above, existing works [11], [34], [9], [35] have also studied providing general INA services for a class of applications and supporting concurrently tasks.

- Key-Value Streams aggregation, such as reduce in big data and gradient aggregation in distributed ML, is supported by ASK [11], which employs a multi-key packet scheme to enhance goodput and a shadow copy mechanism to prioritize hot keys protocol-agnostically.
- NetRPC [34] provides software developers with a pre-built RPC interface that supports in-network computation. Users can use the INC services through simple configuration file editing at the end-host without the need to understand low-level chip design details.
- Lyra [9] and ClickINC [29] present a high-level language and cross-platform compiler to aid in data-plane programming. Lyra offers a one-big-pipeline abstraction, allowing programmers to express their intent with simple statements instead of focusing on hardware details. ClickINC further provides a higher-level language compiler.
- NetVRM [35] explores the dynamic register memory sharing of concurrent applications on the programmable network. It designs a virtual register memory management system that supports dynamic allocation at runtime without the requirement of reloading and recompiling the programmable devices.
- As a concurrent work, SwitchVM [16] proposes a language-level virtualization approach to enable multi-tenant in-network acceleration, which shares a similar idea to our work. In comparison, INAaaS focuses more on

aggregation and caching applications, providing simpler primitives that better suit higher-level applications.

B. Cloud Service

Cloud computing [27] has emerged as a popular computing paradigm in recent years. It refers to the delivery of computing resources over the internet as a service, where users can access and utilize computing resources, such as storage, processing power, and applications, on demand. Cloud services offer several advantages over traditional computing models, including lower upfront costs, flexibility, scalability, and high availability.

Cloud services are categorized into IaaS, PaaS, and SaaS. IaaS offers virtualized resources like servers and storage for application deployment. PaaS provides a platform for developers to build and test applications without managing infrastructure. SaaS delivers internet-based software applications managed by cloud providers.

Cloud services have become ubiquitous in modern computing, with many organizations and individuals using cloud services to store and process data, run applications, and host websites. However, the adoption of cloud services also presents several challenges, including data security, privacy, and regulatory compliance. As such, it is essential to carefully consider the risks and benefits of cloud services when deciding to adopt them.

C. Programmable Switch

The advent of programmable switches [6] allows us to easily manipulate the data plane, and offload some computing or monitoring tasks to the network to reduce bandwidth usage and latency. In order to achieve high packet processing speed (up to 12.9Tbps [5]), programmable switches use a multi-stage pipeline architecture, where each stage can process a fixed number of operations on packets or switch memory, such as memory access, manipulating packet metadata, arithmetic operations. There are two types of processing objects in the switch: stateless objects, such as per-packet metadata, and stateful objects, such as registers.

Although programmable switches can facilitate many applications [14], [32], [3], the usage is constrained by three factors: 1) Limited memory size: Fast memory (e.g., TCAMs, SRAMs) restricts memory to a few tens of MBs. 2) Limited actions: They support only a limited set of operations; for instance, the Tofino switch does not handle floating point operations. 3) Limited operations per packet: With only tens of nanoseconds for processing, the number of operations per packet is restricted, preventing operations like loops.

III. Motivation

A. Providing INA for the cloud

Traditional cloud service use Software-Defined Networking (SDN) to provide programmable network functions, e.g, network virtualization, centralized network management, and dynamic bandwidth allocation [17]. While, SDN has limited data-plane programmability and can not fully support the latest INA applications listed in §II-A. Compared to SDN, programmable switches have significant advantages, including enhanced data processing capabilities, lower data forwarding latency, and greater flexibility in managing complex network tasks [6]. These benefits encourage us to apply programmable switches to provide INA services for cloud environments.

INA can effectively reduce latency and improve throughput, which are critical for cloud services. However, most existing INA solutions focus on dedicated clusters and specific applications. In the cloud scenario, many tenants running multiple applications currently, and cloud users usually cannot manipulate in-network devices, e.g. NICs and switches. Therefore, we cannot directly apply such solutions to the cloud. To enable generic INA on the cloud, one strawman solution is to integrate all dedicate INA solutions and run them simultaneously. However, network devices have limited programming, computing, and storage resources, compared to the requirement of each solutions. Another strawman idea is to set up dedicated racks or clusters for each application. However, this approach has two drawbacks, one is to cause the under-utilization of cloud resources, and the other is that such dedicated racks or clusters may still have multiple application coexistence scenarios, e.g., the distributed ML training includes both computation and distributed data storage, meanwhile, when running user applications, the cloud vendors may also perform some network measurements.

B. Existing Approaches and Limitations

To the best of our knowledge, no work has yet considered supporting generic INA acceleration in the cloud, while several solutions [11], [34], [9], [35] have been proposed to support running multiple applications simultaneously on the in-network devices, see §II-A. Here we clarify why these efforts are not sufficient for the cloud.

Need to recompile when workload changes. Several works, e.g. Lyra [9] and ClickINC [29] provide date plane programming compilers, and they can merge multiple applications into one program. However, such combination methodology is not suitable for a dynamic workload. When a new application starts in the cloud, the network devices need to be recompiled, which leads to service interruptions. It violates the principle that different users must not interfere with each other.

Lack of isolation and scheduling among users. Recent work, e.g., ASK [11] provides a generic INC interface for key-value pairs and NetRPC [34] designs a remote procedure call for INC. Both of them support multiple concurrent applications. However, they ignore security issues, and the register memory allocation is based on hash mapping, which may lead to a potential risk of data leakage. Moreover, they do not pay attention to the scheduling of programmable resources, which may cause sub-optimal performance on the cloud.

Limited to one single rack or small-scale clusters. Some of existing work only design for single rack deployment or is hard to scale out. For example, although NetRPC [34] can work with multiple switches, it requires to chain the

switches into a longer pipeline in the same rack. For more complex topology, e.g., fat tree, an application's traffic may travel through multiple paths, thus the cloud need a cross-rack design. Some solutions consider the coordination and deployment on multiple racks, e.g., Lyra [9] proposes a multiple device deployment algorithm based on SMT (Satisfiability Modulo Theories). However, this approach is quite slow and its time complexity is exponential, e.g., a machine learning aggregation task with only five device costs more than 30 minutes for Lyra to find out a deployment solution.

IV. System Design

In this section, we introduce the design of INAaaS. We first analyze the three challenges of supporting universal in-network acceleration in cloud scenarios, and provide corresponding solutions. Subsequently, we introduce the packet format and switch layout of INAaaS, as well as the switch logic workflow and four basic primitives of INA. Next, we discuss the reliability mechanisms used to handle issues such as packet loss and overflow to ensure correctness, as well as the security design. Finally, we use several common cloud applications as examples to illustrate the design of the INA adapter at the end-host, and introduce the pricing model.

A. Design Challenges

Diverse cloud applications vs. limited in-network programming resources. Various type of applications coexisting in the cloud environment, e.g., KV store, Map reduce and network measurement. However, the programming and hardware logical resources of network device are limited. For instance, the Tofino switch we use only has 12 stages per pipeline, with only 4 stateful ALUs in each stage, and it does not support loop. Considering the high throughput requirements, the restriction of in-network hardware logical resources is inevitable.

Numerous cloud applications vs. limited in-network memory. The multi-tenancy oriented cloud needs to support numerous concurrent tasks with massive storage requirements. Previous work, NetRPC, divided switch memory into two parts: one for KV-pair storage and the other for sequential storage. In the cloud, to improve the utilization of switch memory, we should unify the storage format and enforce fine-grained memory scheduling based on the application requirement.

Security, privacy requirements vs. shared in-network storing and processing. Unlike dedicated data centers, the cloud provides services and devices to different tenants, therefore, isolation and security among different users matter. INA applications involve switch memory access. Existing INA solutions, e.g., NetRPC and SwitchML, do not take security issues into account. Their switch memory addressing schemes are based on simple hashing, which may lead to malicious theft and tampering of user data.

B. Design Overview

Figure 1 illustrates the methodology of INAaaS. We streamline the switch side to just support four primitives, i.e., read, write, reduce and compute, and isolation mechanism. To guarantee the correctness, reliability and security, we design a goalkeeper mechanism on an end-host which is connected to the switch, the end-host also acts as controller to manage tenants. For cloud users, we provide application specific adapters for them to use the INA service transparently. The adapter sends requests to the generic APIs, which automatically performs recording and pricing. Here we propose three key ideas to tackle the three challenges mentioned above.

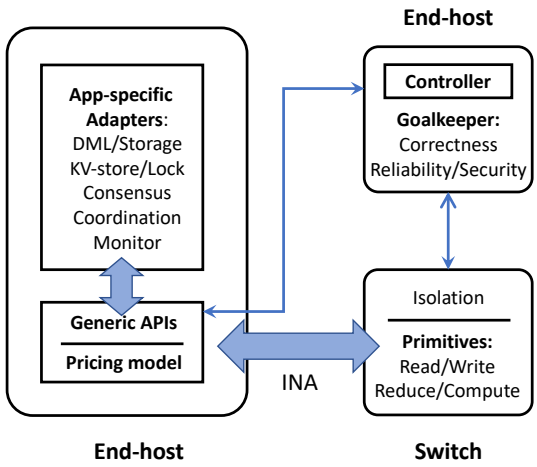


Figure 1: Overview of INAaaS

Switch and end-host co-design. In order to support in-network acceleration for multiple cloud applications at the same time, our methodology is to summarize the basic operations, i.e., primitives, used by the applications, and put these primitives into switches to implement them, and provide adapters based on these primitives on the end-host side for each type of application. We found that only four primitives are needed to support the common applications listed in the figure, i.e., read, write for switch registers, reduce computation on the values of multiple packets, and direct computation on the values of a single packet. We will describe how each application can be accelerated on the intranet with the help of the above four primitives. Meanwhile, to handle corner cases such as packet loss, hash collisions, and preemption, and considering the limitations of switch logic, we employ the end-host goalkeeper mechanism to forward and process these packets.

Harmonizing switch memory format. Considering the scarcity of switch memory, we adopt a unified switch-side storage format for all applications, namely in the form of key-value pairs. To enhance packet storage efficiency, each packet contains 20 key-value pairs of the same format. This design also takes into account that a switch pipeline has multiple stages capable of processing different data simultaneously. The scheduling and allocation of in-network memory affects the latency, throughput, and accuracy of applications. We quantitatively analyze and test the impact of switch memory on the above metrics in the subsequent sections, which allows us to provide performance-guaranteed INA services.

Switch memory protection and identity verification. In the cloud, it may be necessary to give the user access to the entire end-host, including sending and receiving packets from the network card. The previous INA scheme has no protection mechanism for switch memory access, and its commonly used single hash addressing and checking method based on application id can be a security risk. Malicious users can read or modify other users' data stored in switch memory by hash collision and tampering with application id. INAaaS provides dedicated protection for the key-value pairs in switch memory. Before accessing this data, identity verification is required to ensure user unique identifier consistency. Even if switch memory is preempted, we ensure that the old data can be safely transferred to the endpoint without being accessed by other users.

C. INA Protocol

Packet format. The packet header of INAaaS includes a 32-bit timestamp field, a 32-bit time limit field (T), followed by 20 unique identifier fields (uid_i), each 32 bits in length. Additionally, there are 20 validity fields (valid_i), each 1 bit, to indicate the validity of the corresponding uid_i. The header also includes 20 read flags (read_i), 20 write flags (write_i), and 20 reduce flags (reduce_i), all of which are 1 bit each, to denote the respective operations. Furthermore, 20 compute flags (compute_i), each 1 bit, are incorporated to signify compute operations. An operator field, 4 bits in size, is included to specify the operation type. Lastly, the header contains 20 key fields (key_i) and 20 value fields (value_i), each 32 bits in length, to store the key-value pairs being transmitted.

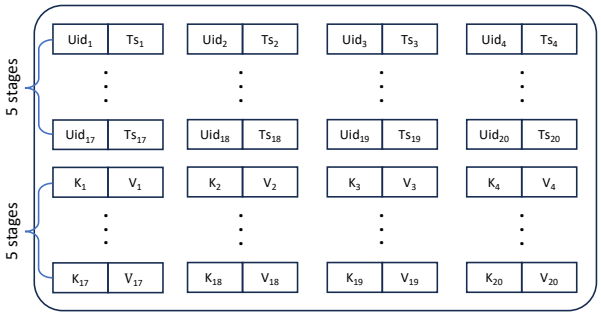


Figure 2: INAaaS switch layout

Switch layout. INAaaS use a unified switch memory structure. As shown in Figure 2, the first half consists of 20 identically structured 64-bit registers, each with the first 32 bits representing the uid and the remaining 32 bits representing the timestamp. The second half contains 20 corresponding key-value pairs. The switch processes 4 registers per stage.

Primitives. We have implemented four primitives in the switch to support in-network acceleration: read, write, reduce, and compute. The workflow is shown in Algorithm 1. The INAaaS switch procedure begins by parsing the header of the incoming packet P. Initially, a bitmap array is initialized to track the state of 20 registers. In the first stages (2 to 6), the algorithm checks each register's validity and timestamp. If the timestamp has expired or the uid is 0, it swaps the packet's uid with the register's uid and updates the timestamp, then sends the packet to Goalkeeper. In the subsequent stages (7 to 11), the algorithm performs uid matching and processes key-value operations based on the packet's read, write, reduce, and compute flags. If the bitmap indicates any changes (failure happens), the packet is sent to Goalkeeper.

With these primitives, we can support the common cloud applications listed above. At the same time, the performance (throughput, latency) of these applications largely depends on the operations corresponding to these primitives.

- *Read.* This primitive allows users to read a 64-bit key and value pair from the switch memory. We will fetch the corresponding 64 bits from the register array of the corresponding stage according to the domain index_i.
- *Write.* This primitive allows users to write a 64-bit key and value pair to the switch memory.
- *Reduce.* This primitive supports the reduce operation of multiple workers, i.e., reducing a set of values to one output, such as gradient aggregation in distributed machine learning. For the reduce operation, our key is stored in 16 bits, with the remaining 16 bits used as a bitmap to record the packet arrival status of each worker (therefore the maximum supported worker number is 16).
- *Compute.* This primitive performs stateless calculations on packet fields, storing results in the value field and setting the op bit to 15 upon completion. Supported operations include addition, subtraction, bitwise operations (shift, AND, OR, XOR, NOT), and max/min, with op field values from 0 to 9.

Switch memory scheduling. The allocation and scheduling of switch memory affects the performance of various applications. INAaaS designs the scheduling algorithm based on two observations.

The first observation is that different applications have different demands on switch memory. For latency-sensitive tasks, e.g., KV store and paxos, their performance is directly related to the hash hit rate, since only hash-hit requests can enjoy in-network sub-RTT responses, and the size of the owned memory determines the hash hit rate. Memory preemption has little impact on such applications and only affects the query speed of a key. For throughput-sensitive tasks, such as DML training, allocated switch memory determines the upper limit of its sending window, which in turn affects the throughput. Memory preemption has almost no impact on such applications, except that it divides the reduce result into multiple packets, which slightly increases network traffic. For accuracy-sensitive tasks, such as bloom filters, the memory size determines the accuracy, e.g., false positive rate. A memory preemption to one of it register may impact all query results.

The second observation is that due to the straggler phenomenon, map-reduce tasks may waste switch memory. For example, gradient aggregation will occupy aggregator registers until the gradient of the last worker arrives. When the network

Algorithm 1: INAaaS Switch Procedure

```
Input: Packet P
Output: Packet P
parse header of P → hdr;
meta.bitmap[20] := 0;
if hdr.retrans = False then
    In stage i (2→6);
    j = (i-2)*4, (i-2)*4+1, (i-2)*4+2, (i-2)*4+3;
    pass j if hdr.valid_j = 0;
    if hdr.timestamp − reg[hdr.idx_j].timestamp > hdr.T or
      reg[hdr.idx_j].uid = 0 then
        swap hdr.uid_j and reg[hdr.idx_j].uid;
        reg[hdr.idx_j].timestamp := hdr.timestamp;
        meta.uid_j := hdr.uid_j;
        hdr.read_j := 1, hdr.write_j := 1;
        P.egressPort := the port of Goalkeeper;
    else
        meta.uid_j := reg[hdr.idx_j].uid;
    In stage i (7→11 );
    j = (i-7)*4, (i-7)*4+1, (i-7)*4+2, (i-7)*4+3;
    pass j if hdr.valid_j = 0 or hdr.uid != meta.uid_j;
    if hdr.read_j = 1 and hdr.write_j = 1 then
        if reg[hdr.idx_j].key = 0 or hdr.key =
          reg[hdr.idx_j].key then
            swap hdr.value_j and reg[hdr.idx_j].value ;
            swap hdr.key_j and reg[hdr.idx_j].key ;
        else
            meta.bitmap[j] := 1;
    if hdr.read_j = 1 and hdr.write_j = 0 then
        ··· ;
        hdr.value_j := reg[hdr.idx_j].value ;
        hdr.key_j := reg[hdr.idx_j].key ;
        ··· ;
    if hdr.read_j = 0 and hdr.write_j = 1 then
        ··· ;
        reg[hdr.idx_j].value := hdr.value_j ;
        reg[hdr.idx_j].key := hdr.key_j ;
        ··· ;
    if hdr.reduce_j = 1 then
        ··· % only compare the first 16bits of key;
        reg[hdr.idx_j].value := reg[hdr.idx_j].value +
          hdr.value_j ;
        reg[hdr.idx_j].key = reg[hdr.idx_j].key OR hdr.key_j ;
        ··· ;
    if hdr.compute_j = 1 then
        ··· ;
        hdr.value_j := hdr.op(hdr.value_j, hdr.para_j) ;
        ··· ;
else
    P.egressPort := the port of Goalkeeper;
if meta.bitmap != 0 then
    P.egressPort := the port of Goalkeeper;
deparser;
```

bandwidth is sufficient, a better choice is to commit the current aggregation results and release the registers.

Here our scheduling policy is that 1) assign more memory to hot items of KV store to increase the hash rate, we record the hot items at the end-host, like NetCache, and store the hot items twice in different stages. 2) encourage memory preemption of throughput-sensitive tasks, this can be done by setting smaller time limit (hdr.T). 3) For the accuracy-sensitive tasks, we guarantee that its memory will not be seized after it has been allocated. This can be done by setting the timestamp to run faster than the normal clock

D. Reliability and Security

Packet loss and host failures. Since our switch is simplified as much as possible, we use the goalkeeper mechanism on the end-host to ensure correctness and reliability under packet loss and retransmissions. For packet loss, since the packet sender and the goalkeeper are both end-server, we set a timeout to detect the retransmission. INAaaS tags retransmitted packets on their header at the end-host, upon a switch receives such packets, it will forward the packets to the goalkeeper. It has little impact on the performance because: 1) Retransmissions account for a very small percentage (usually less than 0.1%) and 2) We can do selective retransmissions, making the overhead from retransmissions even lower. Another issue is isolation. Once a tenant uses the cloud service, the controller assigns he/she with a uid. When accessing the memory of switches or goalkeepers, the uid is checked, if it is not the same, the access will be denied.

Quantization and overflow. The Tofino switch we use does not support float point operation on the data plane. Current practice [3], [4] is to apply quantization to change the float point numbers into integers. Previous experiments show that this approach does not impact the application performance. Another issue of computation in programmable switch is overflow, i.e., exceeding the maximum value that can be represented in a fixed-size data type during a calculation. We use the saturating computation of P4 to handle overflow, when overflow happens, the switch will set the number to a predefined value MAX_INT or MIN_INT, then the end-host will know the overflow happens, we also apply the goalkeeper mechanism in this case.

Authentication and memory protection. When a switch receives a packet, it will first determine if it is an INAaaS packet by the type field in the ethernet protocol, if not, it will forward it normally. For INAaaS packet, we first get all the header fields through parser. The next step is to verify the user's identity, we read the uid and timestamp from each of the 20 key-value pair registers. As shown in Algorithm 1, if the current timestamp has expired (to avoid prolonged occupation of a register, we set a maximum usage time T, which varies to the application), we will occupy this key-value pair register, swap the packet's corresponding content with the old content on the switch, and send the packet to Goalkeeper. This mechanism ensures that when a new task acquires a register, it does not leak information from the old task. If the current timestamp has not expired, we proceed with uid matching. Subsequent read/write operations on the key-value pair register can only occur if the uid matches or is 0.

E. End-host Adapters

Here, we briefly introduce four application adapters. For KV store, its INA type is read (query) or write (store), the data structure is KV pair. The end-host is responsible to record the hot items and apply duplicate storing to such hot items. For sparse model aggregation, its INA type is reduce, the data structure is KV pair. Since the memory in each register group (totally 20) is separated, we take the key hash and then take the modulus by 20, dividing the space of all keys into 20 disjoint subsets. We use 20 queues to cache different subsets of KVs. When we assemble a packet we select the heads of the 20 queues. For dense model aggregation, its INA type is reduce, the data structure is array. We set an end-host PS server to handle all partial aggregation results. For bloom filter, its INA type is read (query) or write (add) , the data structure is array. Considering the limitation of switch memory access, we adopt the idea of shift bloom filter [31], and encode the locations that multiple hash functions map to multiple 32 bits integer.

F. INA Services Pricing Model

Two factors matter for our pricing model, one is the cost of providing the INA functions, including the computation resource ALU and memory resource of the programmable switch. Another is the value of the INA to users, including the potential reduction in application latency and increase in throughput. For calculating switch resource occupation, it's important to note that each time a user occupies a new switch register, a packet is sent to the Goalkeeper server (see Algorithm 1). Additionally, when a register is released, the Goalkeeper is also notified. This enables us to accurately track each user's resource usage. Potential acceleration is determined by measuring the throughput and latency metrics of specific tasks.

Correspondingly, INAaaS provides two types of INA services pricing. The first is a pay-per-use model. We record the number of times a user's application uses the ALU resources on a programmable switch, and the number of switch registers it occupies multiplied by the duration. The second is a subscription based model. Considering the contention of the INA resources, the user experience may be affected, e.g., cannot guarantee a consistent low latency. To provide cloud users QoS (Quality of Service), we give subscribed user higher priority on accessing switch resources.

V. Implementation

We implement INAaaS switch logic on a 12-stage programmable switch. The INAaaS switch pipeline contains 20 read-write memory segments corresponding to the 20 key-value pairs in the INAaaS packet. Each memory segment contains 40k 64-bit units to restore INC states or the INC map. Depending on the service configuration, we set the packet length to 300 bytes. Note that typically programmable switches have four pipelines. The current design only uses one pipeline. In our future research, we plan to fully utilize all pipelines to increase packet length. We implemented the end-host adapter for each application using C++. Additionally, we provide users with four switch primitives to facilitate the development of their own applications. To prevent users from tampering with the uid, the packet assembly component of INAaaS is not exposed to the tenant environment but is completed by the cloud service provider. In switch side, we used a technique where the 64-bit register in the Tofino switch is divided into high and low parts, allowing for simultaneous read and write operations, and provides two conditional logics. We combined the uid and timestamp together, enabling accurate monitoring of the usage time of a specific uid. Additionally, we combined the key and value together as a key-value pair for simultaneous read and write operations.

VI. Evaluation

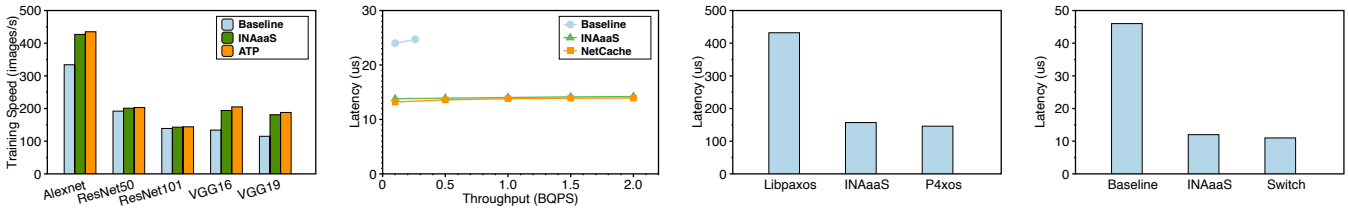
We evaluate INAaaS with a combination of testbed validation and simulation measurement. The principal discoveries from our evaluation are as follows:

- Testbed results show that INAaaS nearly achieves the same level of acceleration as the SOTA in-network acceleration scheme for several popular applications, and INAaaS can support many concurrent applications.
- As a complement to testbed, simulation shows that with more servers under a rack, INAaaS is still effective.
- Deep dive experiments show the improvement of INAaaS scheduling algorithm on key metrics of different applications. Meanwhile, we further show that INAaaS can quickly find the attacker without affecting normal users.

A. Testbed

Setup. Our testbed contains one AS9516-32D programmable switch and ten servers, five of them are GPU servers, each with two V100 GPUs, 40 CPU cores (Intel Xeon Gold 5115), 128GB memory, two Mellanox ConnectX5 100Gbps NICs. The other servers are CPU servers, each has 24 CPU cores, 64GB memory and one ConnectX5 NIC. The switch has 100Gbps links connecting to each NIC of the above servers. To scale up our testbed, we further divide one GPU server into two separated docker containers, each with $1\times$ GPU, $20\times$ CPU cores, 64GB memory and a 100Gbps NIC.

Speedup on different applications. Here we evaluate the performance of INAaaS on four widely-used cloud applications, i.e., distributed ML, KV store, paxos and bloom filter. For distributed ML, we measure the end-to-end training speed of the image classification task. We compare INAaaS with ATP and non in-network acceleration baseline. The implementation is based on BytePS [12], a state-of-the-art machine learning framework. We use all GPU servers and test five popular models, i.e., Alexnet, ResNet50, ResNet101, VGG16 and VGG19. As we can see from Figure 3(a), INAaaS outperforms the baseline and is comparable to ATP on all models. The speedup on ResNet models are tiny, since they have negligible communication compared to computation. For KV store, we show the average latency vs. throughput curve of INAaaS, NetCache and the no in-network cache baseline. We use the same application setup in NetCache. Figure 3(b) shows that INAaaS achieves the similar speedup as NetCache



(a) Distributed ML (b) KV Store (c) Paxos (d) Bloom Filter

Figure 3: Speedup on different applications.

Application	1×	2×	4×
Distributed ML (throughput)	1.45	1.45	1.44
KV store (average latency)	1.74	1.72	1.71
Paxos (tail latency)	2.75	2.75	2.75
Bloom filter (aver. latency)	3.83	3.83	3.83

Table II: The speedup of INAaaS with concurrent applications.

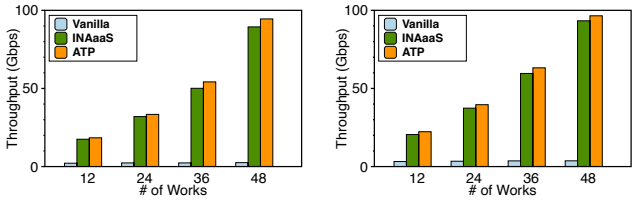
and can also increase the upper bound saturated throughput. Note that the baseline throughput saturates at 0.26 BQPS. For the Paxos consensus system, we compare the tail latency (99th-percentile) of INAaaS with P4xos and libpaxos [1]. We measure the time cost to make one consensus. Figure 3(c) shows that INAaaS successfully cuts the tail latency compared to the server-base solution and the performance is close to the specific INA design. For the network measurement tools bloom filter, we show the average respond latency of a query. We implement classical bloom filter in both switch and end-host. All solutions have the same memory usage. From Figure 3(d) we can see that INAaaS achieve the same speedup as the switch based solution. Such INAaaS implement a variant of bloom filter, we also measure the false positive rate, INAaaS also show the same rate, i.e., 2.4%.

Speedup on concurrent applications. Here we emulate the cloud environment with multiple concurrent applications of different users. We run four type of applications concurrently, i.e., distributed ML with VGG16, KV store, Paxos and bloom filter. We compare INAaaS with the end-host based baseline. To show the impact of the number of applications, we show the speedup of run 1×, 2× and 4× instances of each application. Table II shows the results. The conclusions are 1) INAaaS achieves similarly speedup in the shared environment as the exclusive one. 2) Increasing of instances number does not mitigate the performance improvement.

B. Simulation

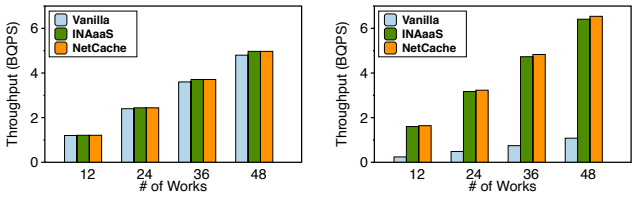
Topologies: We set up a single switch topology, which has one switch with 48 100Gbps links to 48 hosts, and one 100Gbps links to the controller host. For each experiment, we set the number of servers under each rack as 12, 24, 36 and 48, respectively. We assume the switch data-plane memory size is 10MB. The packet size is 300Bytes.

Workload: For DML, we simulate the traffic pattern of parameter server in one iteration. We assume there are only one PS server, as the setting in ATP [3]. Here we chose two ML models. The first is ResNet50, each worker generates 96MB



(a) ResNet50 (b) VGG16

Figure 4: Distributed Machine learning.

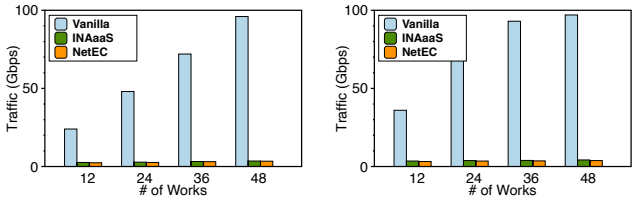


(a) Uniform (b) Skewed

Figure 5: Key-Value Store

data in one iteration. The second is VGG16 with 512MB. For key-value store, We assume the ratio of client to service is 5:1 and evaluate two workloads, i.e., uniform and skewed (zipf-0.95) [14], For distributed storage, we test two cases, replication and RS(3,2) [23]. The SSD write speed is 2 Gbps.

Distributed Machine learning. Here we compare the communication throughput of INAaaS with ATP [3] and vanilla, i.e., baseline without INA on DML. Figure 4 shows the INA performance on DML. Y-axis indicates the gradient aggregation throughput, and X-axis indicates the total number of workers. We can see that with the number of workers increase the baseline is almost consistent, while INAaaS and ATP increase linearly. This indicate that INA successfully address the downlink bottleneck by aggregating gradients in the network and for DML, INAaaS performs INA correctly.



(a) Replication (b) RS(3,2)

Figure 6: Distributed Storage

Application	INAaaS	FCFS	FQ
Distributed ML (throughput)	1.45	1.22	1.25
KV store (average latency)	1.72	1.31	1.37
Paxos (tail latency)	2.75	2.59	1.78
Bloom filter (ave. latency)	3.83	3.64	1.93

Table III: The speedup of INAaaS with concurrent applications.

Key-Value Store. Here we compare the throughput (billion query per second) with NetCache [14] and vanilla on key-value store. Figure 5 shows the case of key-value store. For the uniform workload, the three algorithms are almost the same, which is in line with the finding in NetCache [14]. For the skewed workload, i.e., zipf distribution with the parameter equal to 0.95, the INA solutions significantly out-performs the baseline and INAaaS is close to the SOTA.

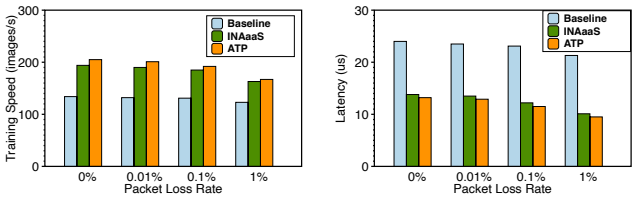
Distributed Storage. Here we compare the traffic volume with NetEC [23] and vanilla on distributed storage. Figure 6 shows the results of network traffic volume generated by reconstruction. In both two cases, we can find that the INA solutions significantly reduce traffic volumes, which helps to mitigate the network pressure.

C. Deep Dive

[simulation] Speedup from switch memory scheduling. Due to the scarcity of switch memory, INAaaS uses a fine-grained scheduling algorithm. Here we verify the effectiveness of the algorithm. We first chose two baseline scheduling, i.e., FCFS (First Come First Serve) and fair queuing. We run four type of applications concurrently, each with 2× instance. Table III shows the speedup of each scheduler compared to the end-host based solutions. We can see that INAaaS outperform the two baseline in all cases, especially on the KV store and bloom filter. The reason are that INAaaS schedules the switch memory based on the demand of application, the cache rate of KV store directly impact the performance, thus it gets more memory in INAaaS. Bloom filter has static and persistent memory demand, thus INAaaS allocation less but persistent memory to it.

[Testbed] Impact of malicious switch memory access. INAaaS prevent access to switch memory by malicious users by checking the application key stored in packet header, if the key mismatches, INAaaS will forward the packet to a specific server and logs it. When the accumulate counting of a user over a period of time exceeds a threshold, we mark it as a malicious user. We simulate the attaching behave of users by filling in random value as the application key. Result show that INAaaS can find out the malicious user immediately before they have chance to access switch memory.

[Testbed] Impact of packet loss. Packet loss may happen in the cloud, especially when the network load is heavy. Here we measure the impact of packet loss on two applications, i.e., distributed ML training on VGG16 and query on KV store. For the KV store, we fix the query throughput to be 0.1 BQPS and measure the average latency. As shown in Figure 7, when the packet loss rate is less than 0.1%, there are almost no performance degradation of INAaaS.



(a) Distributed ML (b) KV Store

Figure 7: Impact of packet loss.

VII. Related Work

Compiler for Data Plane ASIC Programming. Existing programming on data plane ASIC are coded in low-level languages, e.g., P4 [7] and NPL [2], which creates barriers for users. To aids the programmers, μP4 [26] enables modular programming in the data plane of PISA switches through the composition of reusable libraries. For programmable switching chips architectures, e.g., RMT and FlexPipe, work [15] designs the compiler for P4 programs. Domino [25] extends the Banzai machine model to support a wide range of data plane algorithms, including stateful packet processing. Chipmunk [10] leverages slicing to optimize Domino's compilation time and resource usage. Lyra [9] introduces a chip-level details independent new language to enable data plane programming over multiple switches.

Virtualization for P4 Data Planes. Virtualization enables multiple P4 programs to share underlying resources efficiently. Approaches like Hyper4 and HyperVDP [33] implement P4 data plane virtualization by introducing a hypervisor P4 program. P4VBox [24] facilitates parallel execution of virtual switch instances and supports hot-swapping these instances at runtime. However, these methods are primarily for simple network function virtualization and hard to handle complex in-network acceleration tasks like caching and aggregation.

VIII. Conclustion

In this paper, we propose a generic INA framework called INAaaS to provide on demand INA service for the cloud. INAaaS propose one in-network hardware design for all applications based on programmable switches. It further addresses the isolation and security among different cloud users and deploys a INA service pricing model. We expect that INAaaS will accelerate the development of next-generation cloud services and unlock the potential of in-network computing.

Acknowledgement

We thank the anonymous reviewers for their feedback and suggestions. This work is supported in part by the Hong Kong RGC TRS T41-603/20R, GRF 16213621, ITF ACCESS, NSFC 62062005, NSFC 62472050 and the Science and Technology Project of Hunan Provincial Department of Water Resources under Grant XSKJ2024064-36.

REFERENCES

[1] General purpose paxos library: bitbucket.org/sciascid/libpaxos, 2013.

[2] Broadcom's new trident 4 and jericho 2 switch devices offer programmability at scale., 2019.

[3] ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[4] Scaling distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[5] Anurag Agrawal and Changhoon Kim. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–32. IEEE Computer Society, 2020.

[6] Roberto Bifulco and Gábor Rétvári. A survey on the programmable data plane: Abstractions, architectures, and open problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–7. IEEE, 2018.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Vahdat, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[8] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–7, 2015.

[9] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. 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*, pages 435–450, 2020.

[10] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. 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*, SIGCOMM '20, page 44–61, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A generic service to provide in-network aggregation for key-value streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 33–47, 2023.

[12] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 463–479, 2020.

[13] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 35–49, 2018.

[14] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[15] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 103–115, 2015.

[16] Sajy Khashab, Alon Rashelbach, and Mark Silberstein. Multitenant In-Network acceleration with SwitchVM. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 691–708, Santa Clara, CA, April 2024. USENIX Association.

[17] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[18] Wenxue Li, Xiangzhou Liu, Yuxuan Li, Yilun Jin, Han Tian, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. Understanding communication characteristics of distributed training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, pages 1–8, 2024.

[19] Wenxue Li, Junyi Zhang, Yufei Liu, Gaoxiong Zeng, Zilong Wang, Chaoliang Zeng, Pengpeng Zhou, Qiaoling Wang, and Kai Chen. Cepheus: accelerating datacenter applications with high-performance roce-capable multicast. In *2024 IEEE International Symposium on High-Performance Computer Architecture*, pages 908–921. IEEE, 2024.

[20] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 279–291, 2019.

[21] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 795–809, 2017.

[22] Heng Pan, Penglai Cui, Ru Jia, Penghao Zhang, Leilei Zhang, Ye Yang, Jiahao Wu, Jianbo Dong, Zheng Cao, Qiang Li, et al. Libra: In-network gradient aggregation for speeding up distributed sparse deep training. *arXiv preprint arXiv:2205.05243*, 2022.

[23] Yi Qiao, Menghao Zhang, Yu Zhou, Xiao Kong, Han Zhang, Mingwei Xu, Jun Bi, and Jilong Wang. Netec: Accelerating erasure coding reconstruction with in-network aggregation. *IEEE Transactions on Parallel and Distributed Systems*, 33(10):2571–2583, 2022.

[24] Mateus Saquetti, Guilherme Bueno, Weverton Cordeiro, and Jose Rodrigo Azambuja. P4vbox: Enabling p4-based switch virtualization. *IEEE Communications Letters*, 24(1):146–149, 2019.

[25] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[26] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μp4. 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*, pages 329–343, 2020.

[27] Ali Sunyaev and Ali Sunyaev. Cloud computing. *Internet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*, pages 195–236, 2020.

[28] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. Scalable and efficient full-graph gnn training for large graphs. *Proc. ACM Manag. Data*, 1(2), June 2023.

[29] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, et al. Clickinc: In-network computing as a service in heterogeneous programmable data-center networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 798–815, 2023.

[30] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[31] Tong Yang, Alex X Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. A shifting bloom filter framework for set queries. *Proceedings of the VLDB Endowment*, 9(5):408–419, 2016.

[32] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using 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*, pages 126–138, 2020.

[33] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. Hypervdp: High-performance virtualization of the programmable data plane. *IEEE Journal on Selected Areas in Communications*, 37(3):556–569, 2019.

[34] Bohan Zhao, Wenfei Wu, and Wei Xu. NetRPC: Enabling In-Network computation in remote procedure calls. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 199–217, Boston, MA, April 2023. USENIX Association.

[35] Hang Zhu, Tao Wang, Yi Hong, Dan RK Ports, Anirudh Sivaraman, and Xin Jin. {NetVRM}: Virtual register memory for programmable networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 155–170, 2022.