electronics

MDPI

Article

Design and Implementation of Programmable Data Plane Supporting Multiple Data Types

Linan Jing, Xiao Chen * and Jinlin Wang

The Institute of Acoustics of the Chinese Academy of Sciences, University of Chinese Academy of Sciences, Beijing 100190, China; jingln@dsp.ac.cn (L.J.); wangjl@dsp.ac.cn (J.W.)
* Correspondence: xxchen@dsp.ac.cn

Abstract: Software-defined networking (SDN) separates the control plane and the data plane, which provides network applications with global network topology and the flexibility to customize packet forwarding rules. SDN has a wide range of innovative applications in 5G, Internet of Things, and information center networks. However, the match-action programming model represented by OpenFlow/Protocol Oblivious Forwarding (POF) in SDN can only process limited types of data such as packets and metadata, making it hard to fulfill future network applications. In this paper, data type and data location are added in the matching fields and actions to make the match-action table (MAT) compatible with multiple types of data, hence improving the data plane's programmability. Data type helps the MAT to perceive multiple types of data, allowing them to be processed by a single MAT. Data location allows MAT to be decoupled from data meaning, quickly locating specific data in the switch. Based on Intel's Data Plane Development Kit (DPDK), we design and implement a pipeline that is compatible with multiple types of data processing. Protocol and data type oblivious match-action tables and atomic instructions are included in the pipeline. Experiments show that representing data with data type and data location makes the pipeline compatible with multiple types of data without sacrificing forwarding performance, fulfilling the needs of network applications to handle a variety of types of data while avoiding repeating hardware design.

Keywords: SDN; match-action model; programmable data plane; multi data types

Citation: Jing, L.; Chen, X.; Wang, J. Design and Implementation of Programmable Data Plane Supporting Multiple Data Types. *Electronics* **2021**, *10*, 2639. https://doi.org/10.3390/electronics10212639

Academic Editor: Martin Reisslein

Received: 24 September 2021
Accepted: 26 October 2021
Published: 28 October 2021

1. Introduction

Software-defined networking (SDN) has become the most popular network programmable solution in recent years [1]. SDN separates the control plane and data plane of the network with a southbound interface (such as OpenFlow [2]), and abstracts the data plane's routing and forwarding with a match-action paradigm, which encourages network application development and innovation. For example, microsecond-level events can be processed via exact measurement of in-band network telemetry [3] in SDN, which cannot be captured by traditional network monitoring tools (such as ping, traceroute, etc.). Furthermore, SDN allows network operators to configure network equipment in real time through software programming. It has a diverse set of new applications in fields including the Internet of Things [4], edge computing [5], and information center networks [6].

Innovative network applications have confirmed the feasibility of SDN while also exposing its flaws. Innovative network applications have emerged with a variety of network state processing requirements, as indicated in Table 1. These requirements originate from the application's desire to detect network status fast during packet processing and adapt packet forwarding behavior in real time to respond to network events. However, the match-action programming model offered by OpenFlow/POF, only supports limited types of data (packet fields and metadata), and thus it is hard to handle network states defined by the application. It is common for network applications to add new tables, instructions, or modules to the switch in order for it to process application data without

incurring additional delay owing to communication with the controller. For example, the Port Knocking method [7] adds an XFSM table to match the port transition state. The extended instruction in the stateful firewall [8] is used to change the flow's forwarding strategy based on the TCP connection state. To identify connection congestion, the load balancing system CONGA [9] employs an enhanced DRE module.

Table 1. Applications that need network status handling on the data plane.

Application	Requirement	Ref.	Date
Urban mobility tracking	Copy the status information of the mobile terminal in the 5G network	[10]	2021
Bandwidth isolation	Collect physical network information (e.g., total and remaining link capacity)	[11]	2021
Network monitoring	Collect statistics (e.g., number of packets per flow entry)	[12,13]	2021
Flow size counter	Report to the controller after the completion of collecting the size of flow in data plane	[14]	2020
Distributed Denial of Service (DDoS) detection	Switches count the features of the background traffic to detect potential attacks	[15,16]	2020
Load balancing	Switches share network traffic with multiple links	[9,17–20]	2020
Heavy-hitter detection	Save a counter for every flow	[21]	2019
Stateful firewalls	Switch filters unsolicited inbound TCP connections without any outboard flow	[8,22]	2019
Link failover	Switches save backup path and monitor link's status	[23,24]	2018
Domain Name System (DNS) detection	Assign a counter to keep track of all the resolved IP addresses for clients	[25]	2017
Synchronize Sequence Numbers (SYN) flood detection	Switches maintain a counter for every flow to detect SYN-Flood	[25]	2017
Network Address Transform (NAT)	Track the status of each flow in the NAT switch	[26]	2017

However, there will be many network applications running on the switch at the same time. On the one hand, the expansion for specific application data will result in switch functional redundancy. For example, because the application cannot reuse the subtraction instruction that applies just to the TTL field [27], it must add a new subtraction instruction to handle custom data, resulting in duplicate functionality. Likewise, because the MAT cannot be reused, the application creates a new table to record network status. On the other hand, expanding for specific application data raises the switch's expansion cost. To handle new forms of data, the switch must not only include instructions for processing these data, but also instructions for interacting between the new and original types of data (such as, packet fields, metadata, etc.). Switches frequently expand instructions for various applications, resulting not only in expensive hardware design costs but also increased application testing and deployment time.

To improve the data plane's programming abilities, we need to avoid the problem of functional redundancy and costly growth caused by the switch enabling new forms of data. This article suggests using data type and data location to describe data in the switch in order to achieve compatible data processing with a unified MAT. The data type differentiates various sorts of data, allowing the MAT to be decoupled from specific types

of data. Because of the data location, the matching field and instructions are unaware of the data meaning, decoupling the matching and instruction functions from the specific protocol. After the MAT has been detached from the data type and meaning, data loading and storage may be split into distinct modules. Such a data representation approach is straightforward, and it may rapidly accommodate additional types of data by extending the data loading and storage module. Multiple data interactions can be satisfied by combining data kinds in instruction parameters, thus there is no need to create instructions to interact with new data.

Based on Intel's DPDK [28] architecture, this article designs and implements a pipeline that represents data using data types and data locations. The pipeline is compatible with a variety of data types by using a uniform match-action table and instructions with atomic functions. Experiments demonstrate that utilizing data type and data location to describe data can be compatible with different types of data processing without compromising forwarding performance and can easily accommodate new types of data.

The reset of the paper is organized as follows. Section 2 introduces the data plane's match-action programming model. In Section 3, we go through the architecture and algorithms of data plane-compatible multi-type data processing. Section 4 evaluates the performance of our implementation, and Section 5 reviews the related work. Finally, we conclude the paper with brief future work.

2. Match-Action Model

The network's fast growth is aided by a succession of targeted abstract models. TCP, for example, offers an abstract model of connection queues between terminals. IP provides a simple packet abstract model for data transmission between terminals. SDN separates the control plane and the data plane via open interfaces (such as OpenFlow [2]) and employs the match-action model to take an important step in the abstraction of network functions. It dismantles the traditional network's inflexible structure, which has forwarding and control interwoven in a closed, vertical black box.

The match-action model requires the packet to be matched with the table (MAT) and then processed with the action (or instruction) given by the table's entry. The entry describes the matching value, processing actions, and statistics for the matched packet. As illustrated in Figure 1, the switch's pipeline is made up of match-action tables that instruct the pipeline on how to handle the flow. Based on the match-action paradigm, programmers can use software programming to set the packet forwarding rules. Operators can test and deliver network services in less time, promoting network application development and innovation.

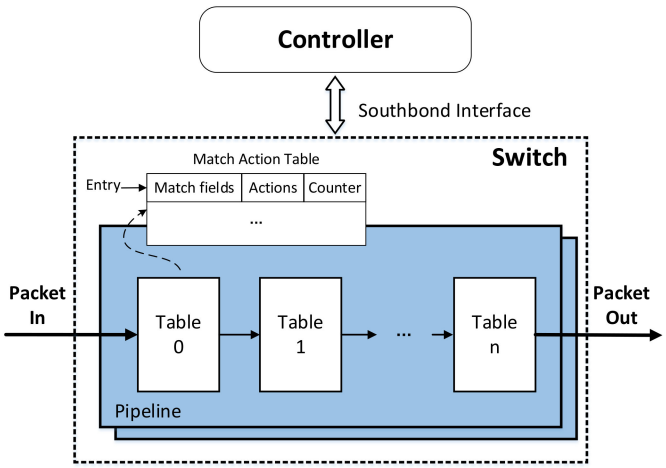


Figure 1. The pipeline uses match-action tables to process the flow.

Early match-action model implementations (such as OpenFlow) were basic, and the switch could only match packets with more than a dozen header fields (such as MAC

address, IP address, TCP/UDP port number, and so on). New protocols (such as the NVGRE, VXLAN, and STT protocols hoped for by data center operators) may only be introduced by extending fields in the new version of OpenFlow.

POF [29] presented a more adaptable implementation of the match-action model. POF locates the packet header field using offset and length, and generic instructions such as field inserting, removing, and modifying are used to substitute instructions that require semantics, such as pushing MPLS labels in OpenFlow, to be explained. POF eliminates the requirement for the switch to understand the protocol format, since it can accommodate any protocol packet. However, offset and length can only represent packet fields. Other types of data (such as Metadata and flow state) cannot be matched and must be handled using specific instructions (e.g., SET FLOW METADATA in POF). This restricts the data plane's programming ability. Extending instructions for certain types of data, on the other hand, will result in the aforementioned concerns of functional redundancy and expansion costs.

On the data plane, in order to match additional types of data besides packets, OpenState [7] introduced state tables and XFSM tables, FAST [30] provided state machine filter tables, state tables, state transition tables, and action tables, SDPA [25] added state tables, state transition tables, and action tables, and FlowBlaze [31] added flow context tables and EFSM tables. These expansions result in several types of tables on the data plane, complicating the southbound interface and making network application development and network management more complicated.

On the data plane, different match-action model implementations can be seen. Despite the fact that the network programming language P4 [32] may shield heterogeneous bottom devices via the compiler (Figure 2). The data plane capabilities supplied by different match-action model implementations varies, which has a direct impact on the network programming language's capabilities. The switch's capability provides the foundation for enabling network programming languages. The upper-level compiler and programming language are powerless if the switch at the bottom layer does not enable recording and processing application-defined data [33]. Furthermore, the P4 compiler is capable of effectively shielding the heterogeneity of data plane devices, so new devices may be added without fear of incompatibility. However, tables and instructions dedicated to certain types of data not only cause functional redundancy and expensive expansion costs, but they also increase the complexity of the southbound interface, which is inconvenient for network management. Faced with the demands of new network applications for the processing of custom network states, match-action models should be more flexible and compatible with different types of data processing.

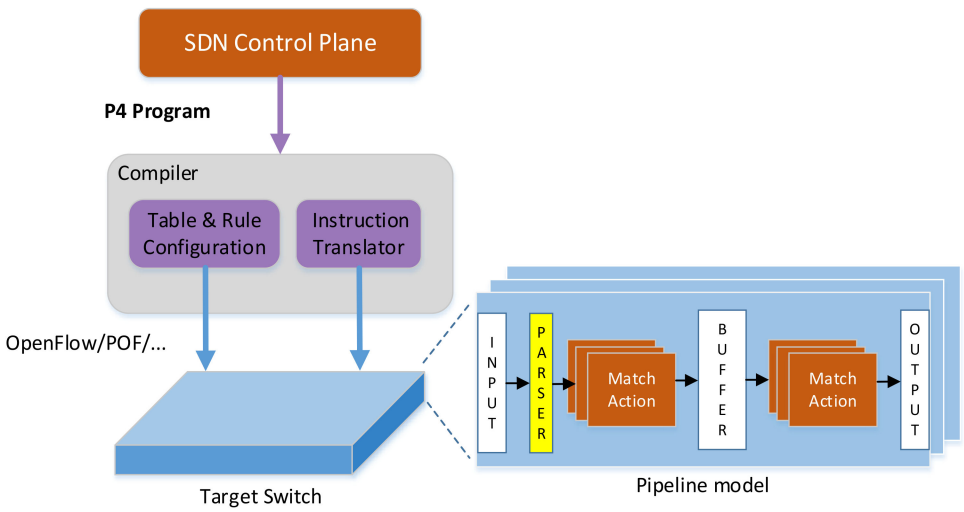


Figure 2. P4 is a network programming language.

3. Data Type and Data Location

We propose using data type and data location to represent data in the switch, allowing the switch to handle various types of data with a uniform match-action table, enhancing the data plane's programmability. In particular, data in the switch is described as type, offset, and length, where type denotes the data type, which can be packet fields, metadata, or flow states. Offset and length describe the data's placement; offset is the data's offset relative to the starting position, and length is the data's length. Because various types of data are obviously stored in different locations in the switch, type also denotes the starting position of this type of data in the switch.

A matching field or an instruction parameter can be indicated by type, offset, and length. As a result, the switch no longer needs to comprehend the meaning of the data (for example, whether the data are the TTL field or the MPLS identifier). The data loading and storage can then be handled by a separate module. The Load and Store module is used by the match-action table to load or store data, as shown in Figure 3. To load the field, we first obtain the base address of this type of data in the switch, then add the relative offset to the base address to obtain the absolute position, and then use the absolute position and length to obtain the data. Similarly, data are written back to the absolute location specified by the base address corresponding to the type plus the offset when it is stored.

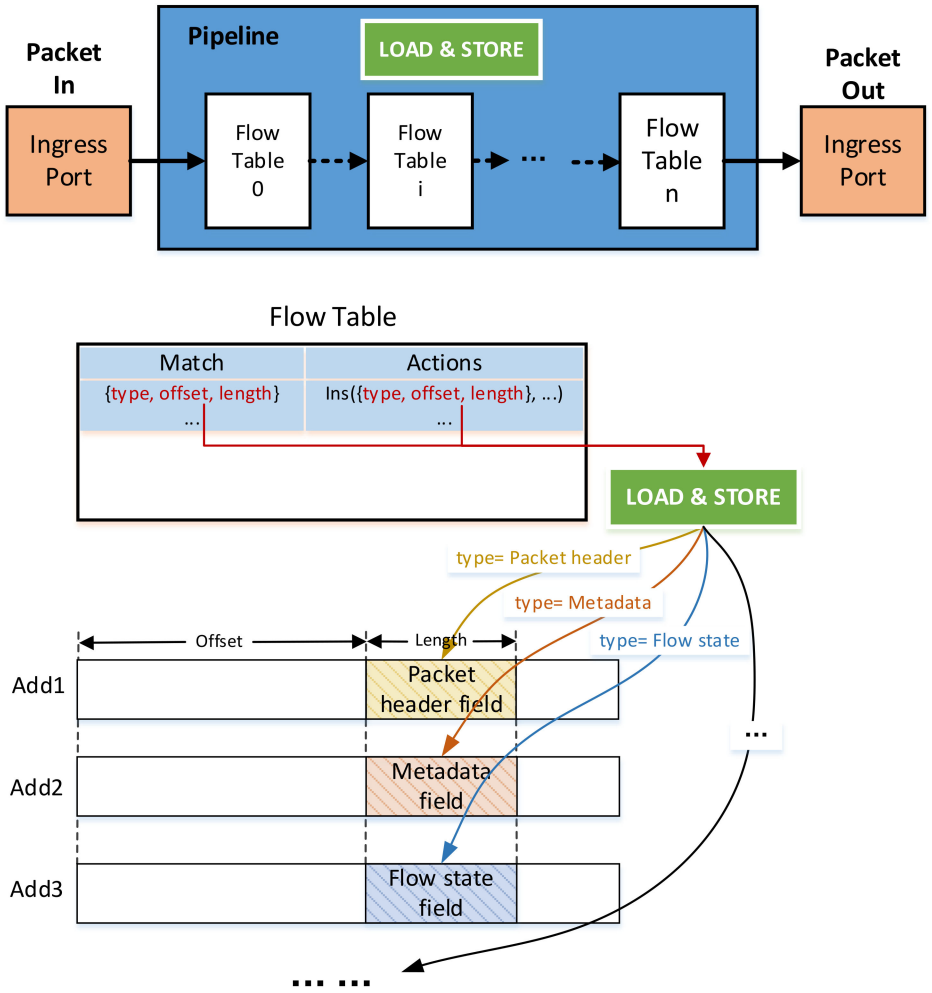


Figure 3. The Load and Store module is used to load and save data in the Match-action table.

Using type, offset, and length to describe data has the following advantages. For starters, the match-action table can now handle a variety of data types and is no longer confined to packet processing. New forms of data can also participate in matching and be processed by existing instructions. Second, it is flexible. By expanding the data loading

and storage module, the match-action table may easily accommodate new types of data. This expansion does not need hardware modification, as application-defined data are usually stored in RAM. It just requires associating the type with the beginning point of the corresponding type of data; no further hardware connection is required. Third, it naturally enables the interchange of diverse types of data. Figure 4 shows how the SET FIELD instruction may be used to make assignments between any two types of data or the same type of data by specifying the types of various parameters. As a result, instruction functions are no longer restricted to certain types of data. Instructions can concentrate on atomic functions such as assignment, comparison, and arithmetic operations. These fine-grained instructions can be used to integrate complicated functions in network applications.

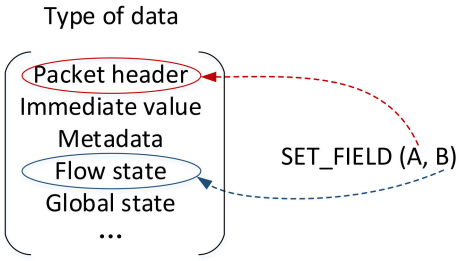


Figure 4. Changing the types of instruction parameters to enable data interoperability across several data types.

It is worth mentioning that utilizing type, offset, and length to describe data may increase packet forwarding latency since load data takes more time to acquire the data's base address. Multiple data may be loaded during packet processing. The time it takes to obtain these base addresses adds to the packet forwarding delay.

To that aim, we offer a data location conversion and interaction mechanism between the application and the switch. To prevent the increased time incurred by getting the base address during packet forwarding, the data location is computed in advance. The key point is that the application must declare the data type in advance and then request space from the switch to record the corresponding data. When adding the match-action table and entry, the switch will record the base address corresponding to the data type and compute the physical address. The method works as follows (Figure 5), (1) the switch reports to the controller the data space that can be used to record application-defined data, (2) the application definition types apply for the required space in the switch, and (3) the switch allocates space and maintains a type-base address table to record the base address corresponding to the type.

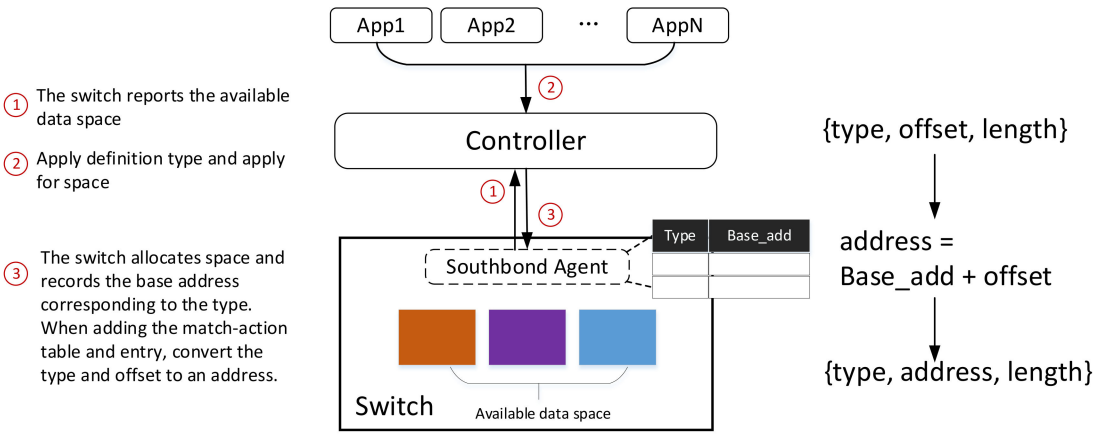


Figure 5. Converting type and offset to data locations while adding the match-action table and entry.

When the switch inserts the match-action table and entry, it completes the conversion of the relative data location {type, offset, length} to the absolute position {type, address, length}, where the data location address is equal to the type plus offset base address. The aforementioned switch-related activities are completed in the southbound interface agent.

Algorithm 1 describes how to load and store data using type, offset, and length. It is worth noting that the packet header and metadata cannot be known until the switch obtain the packet, and the flow state's base address can't be identified until the packet matches the entry. As a result, before beginning the packet processing procedure, the pipeline collects the base address of the packet header, metadata, and flow state (line 1~3). If data are to be loaded (line 4~14). Then, for packet fields, metadata, and flow state data, we simply add the offset to the previously determined base address (line 5~10). Because the location has already been transformed for other types of data (application specified), the second parameter offset has been converted to the data's absolute address (line 12). It should be noted that the base address of the packet and metadata cannot be obtained prior to packet arrival. Similarly, the flow state base address cannot be retrieved until the entry is successfully installed. As a result, these base addresses cannot be translated when adding the table or entry and must be obtained after the packet has been received.

The data can be accessed using the absolute address and data length (line 13). When storing data, (line 16~23), do the same thing as when loading data: first identify the location to save the data in, and then save the data in that address (line 24).

Algorithm 1. Loading and storing data using {type, offset, length} in pipeline

```
Input: packet, match entry, instructions
Output: packet'
1. p_add = base address of the packet header
2. m_add = base address of the packet metadata
3. f_add = base address of the flow state
4. function load (type, offset, length):
5.     if type is packet header:
6.         data_add = p_add + offset
7.     else if type is metadata:
8.         data_add = m_add + offset
9.     else if type is f_add:
10.        data_add = f_add + offset
11.    else
12.        data_add = offset
13.    get the data using data_add and length
14     return data
15. function store (type, offset, length, result):
16.    if type is packet header:
17.        data_add = p_add + offset
18.    else if type is metadata:
19.        data_add = m_add + offset
20.    else if type is f_add:
21.        data_add = f_add + offset
22.    else
23.        data_add = offset
24.    store result in data_add
```

The data type and data location assist the match-action table in dealing with multiple types of data. The method of calculating the data location in advance avoids the increased forwarding latency of calculating the data location in packet forwarding and removes the performance difference that may occur while processing different types of data. We improve the ability of match-action models using the approaches described above, as demonstrated in Table 2. POF and P4 address the problem of OpenFlow's restricted matching fields and provide protocol-independent matching. Additionally, we allow the processing of

different types of data in a single match-action table on the basis of protocol-independent matching, which improves the match-action model's capacity to accommodate innovative network applications.

Table 2. Matching action model implementations.

Implementation	Match Ability	Action Ability
OpenFlow	Specific protocol field	Instruction and protocol tightly coupled
POF, P4 switch	All protocol fields	Instructions support processing limited types of data (packet field, metadata, register)
Our method	All protocol fields and various types of network states	Instruction compatible multi-types of data processing

4. Implementation and Evaluation

4.1. Implementation

We developed a pipeline based on Intel's DPDK framework to focus on proving the approach of utilizing data type and data location to represent data described in this article, as illustrated in Figure 6. The pipeline sends and receives packets using the librte_ethdev library provided by DPDK, and the match-action table function is implemented using the librte_table library. The Execute instructions module in pipeline is in charge of executing instructions to process packets and application data. The pipeline's Load and Store module is in charge of loading and storing data indicated by {type, offset, length}. The pipeline's southbound agent extends the POF southbound interface by: (1) extending the data format to {type, offset, length}; (2) adding the Type-Base address table to record the mapping from data type to base address; and (3) adding the FEATURE_REPORT message for the switch to report available space to the controller. This message describes the available space in the switch that can be utilized to store application data.

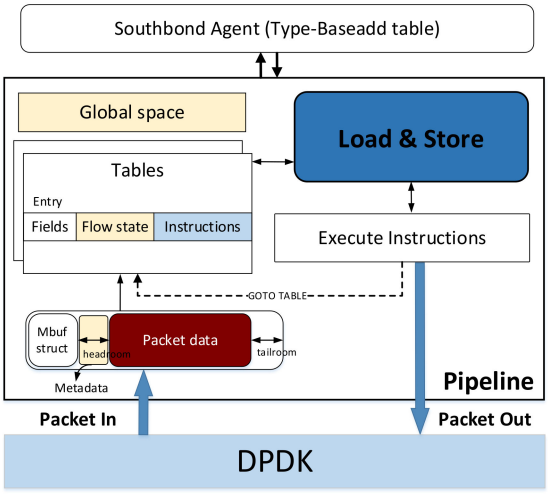


Figure 6. Pipeline design based on the DPDK framework that allows multi-type data processing.

The pipeline contains three areas for recording application-defined data: (1) Metadata, which follows the packet and delivers information between flow tables. Metadata is realized by using the DPDK packet structure Mbuf's 128-bytes headroom. (2) Flow state, which keeps track of the flow status. The flow state of 32 bytes is part of the entry structure. Applications can use instructions to read or write the flow state. (3) Global space, which all flows can access. There is a 1 M bytes global space in the pipeline.

The pipeline enables apps to record their own data. As seen in Figure 7, the application can request global space in the switch to record application data and assign a custom type

to these data. In the pipeline, we used 1 byte to represent the data type, therefore up to 256 data kinds may be recognized. We kept the first five kinds, which are NULL, immediate data, packet field, metadata, and flow state, and the remaining 251 types provide enough room for unique data types for applications.

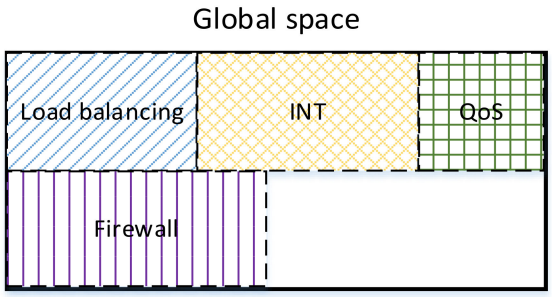


Figure 7. The global space is used to record application-defined types of data.

As illustrated in Table 3, the pipeline supports atomic function instructions for packet and application data processing. (1) Field editing instructions are used for field operations such as modifying, inserting, removing, computing checksums, adding, subtracting, multiplying, dividing, shifting left, shifting right, and, or, xor, and not. (2) Forwarding instructions are used for packet forwarding, including flow table jumps, port forwarding, and floods. (3) Entry instructions are used to perform entry operations such as modifying an entry, adding an new entry to the flow table, and removing an entry. (4) Branch instructions, such as unconditional jump and comparison jump, are used to construct branch logic.

Table 3. Instruction set in the pipeline.

Category	Instructions
Field editing	set_field, insert_field, del_field, calculate_checksum, add, sub, srl, sll, and, or, xor, nor, not
Forwarding	goto_table, output, flood
Branch	jump, compare
Entry	add_entry, del_entry, set_entry

4.2. Evaluation

Three tests were carried out: the effect of data types on forwarding performance, the performance of data loading and storage modules, and the overhead of data location conversion. Table 4 shows the experimental platform that was utilized to run the pipeline and POX controller in the experiment. Spirent Testcenter is used in the experiment to create test packets.

Table 4. System under test data sheet.

Central Processing Unit (CPU)	Intel Xeon CPU E7–4809 v4 @2.10 GHz
Caches	32 k L1i and L1d, 256 KB L2, 20 MB L3
Memory	128 G DDR3 @ 1333 MHz, 4-channels
Network Interface Controller (NIC)	Intel XL710, PCI Express 3.0/x8, 4*10 Gb Intel I350, PCI Express 3.0/x8, 4*1 Gb
DPDK	v19.11

4.2.1. The Impact of Data Type on Forwarding Performance

This experiment investigates the effect of data type on forwarding performance via packet forwarding latency. Figure 8 depicts the flow table utilized in the experiment.

Entries in the flow table alter the destination IPv4 address of the packet using various types of data (immediate data, packet fields, metadata, flow status, application data). The impact of data type on forwarding performance is seen by observing the forwarding delay of packets with varied destination IPv4 addresses.

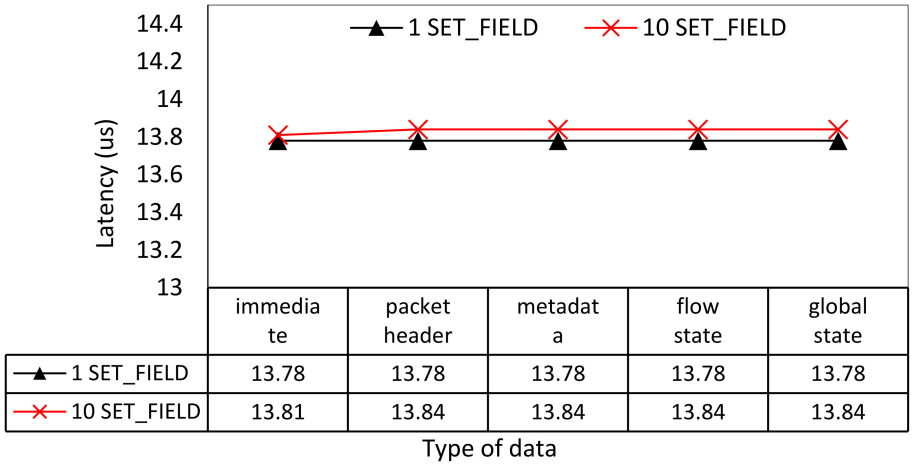
Packet In →

IPv4.Dest :
10.0.0.1/
10.0.0.2/
10.0.0.3/
10.0.0.4/
10.0.0.5

Index	Match: IPv4.Dest	Action
1	10.0.0.1	SET_FIELD (IPv4.dst, 11.0.0.1), Output (0)
2	10.0.0.2	SET_FIELD (IPv4.dst, IPv4.src), Output (0)
3	10.0.0.3	SET_FIELD (IPv4.dst, metadata), Output (0)
4	10.0.0.4	SET_FIELD (IPv4.dst, flow state), Output (0)
5	10.0.0.5	SET_FIELD (IPv4.dst, global state), Output (0)

Figure 8. Flow table used in experiment 1.

Figure 9 depicts the experimental results. When only one SET FIELD instruction is used, the forwarding latency of packets matching various entries remains constant at 13.78 us. This result does not meet our expectations since instructions that employ immediate data load one less time than other types of data, therefore packets matching the first entry should have a lower forwarding latency.



	immedia te	packet header	metadat a	flow state	global state
1 SET_FIELD	13.78	13.78	13.78	13.78	13.78
10 SET_FIELD	13.81	13.84	13.84	13.84	13.84

Figure 9. The impact of different types of data processing on packet forwarding latency.

We suspect that the cause is that the time required to load the data is too short to exceed the instrument's precision. Therefore, we modified the entry instructions to repeat the procedure of changing the destination IPv4 address 10 times. The forwarding latency of packets matching various entries differs after adjustment. Figure 9 shows that the packet forwarding latency for entries 2, 3, 4, and 5 is greater than that of the first entry. The reason for this is that these instructions need the loading of additional data, whereas the immediate data can be used instantly. According to the results in Figure 9, the time it takes to load 32 bits of data is about 3 ns. The packet forwarding latency is the same while loading different types of data (packet fields, metadata, flow state, global state). Because the procedure of loading different types of data is the same, the only difference is the addresses utilized.

In summary, in this paper, we suggested using data type and data location, which is type, offset, length, to represent data within the switch; it can accommodate many types of data without lowering forwarding performance, and it guarantees that the data type has no effect on forwarding performance.

4.2.2. The Performance of Loading and Storing Data

Experiment 1 shows that the time required to load different types of data is the same. This experiment investigates the time required to load/store data of varying lengths. Six different data lengths were evaluated in the experiment. The test data length consists of three types of byte aligned: 16 bits, 32 bits, and 64 bits, as well as three types of byte un-aligned: 7 bits, 21 bits, and 43 bits. Table 5 displays the experimental results. When the lengths are comparable, loading or storing byte-aligned data takes less time than byte-unaligned data. This is due to the fact that loading or saving byte-unaligned data requires additional bit shift operations, which lengthens processing time. To prevent the processing delay caused by non-aligned data, it is suggested that the application utilize aligned data to record the network status.

Table 5. Performance of loading and storing data of different lengths.

Length (bit)	LOAD (ns)	STORE (ns)
7	3.4	5.82
16	2.2	1.92
21	3.99	7.85
32	2.62	2.25
43	5.52	9.3
64	2.85	2.91

4.2.3. The Performance Impact of Separate Data Loading and Storage Modules

The goal of this experiment is to see how isolating the data load and store operations from the instructions affects forwarding performance. The experiment contrasted two data loading methods: (1) utilizing the pipeline mentioned in this chapter to load and store data independently, and (2) using the OVS (OVS-DPDK v2.8.5 [34]), which has a tightly linked data type and instruction function. In the experiment, both methods performed the same operation of subtracting 1 from TTL and computing the packet's checksum. The flow table used in the pipeline and OVS has the same information. We measure packet forwarding delay to see if the independent data loading and storage function resulted in a significant performance difference between the pipeline and the presently popular software switch OVS.

Figure 10 depicts the experimental results. As can be observed, there is not much of a difference between the two methods in terms of packet forwarding latency. This demonstrates that the pipeline presented in this article has equivalent performance to the commonly used software switch OVS. As a result, separate data loading and storage modules have little effect on performance while allowing instruction functions to be atomized. Give innovative network applications a flexible way to actualize the space of complicated network operations by combining instructions.

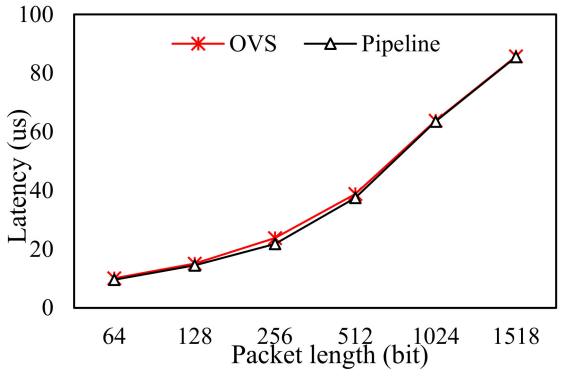


Figure 10. TTL modification and checksum computation are implemented via pipeline and OVS, respectively, with independent data loading and data loading linked with instructions. The difference in performance between the two data loading methods is reflected in packet forwarding latency.

Table 6 shows the CPU clock cycles spent by processing each instruction in the pipeline on the experimental platform. Two results of the instruction processing 64-bit immediate data and field are shown in the table. Knowing the performance allows the program to estimate how much time the instructions take. These instructions can assist network applications in responding rapidly to network events in the switch when they detect changes in network status, avoiding the delay caused by controller involvement.

Table 6. Instruction performance compatible with multiple data types.

Action	Field Type	Cycles	Action	Filed Type	Cycles	Action	Filed Type	Cycles
set_field	f, imm	7	sll	f, f	18	nor	f, f	16
set_field	f, f	13	srl	f, imm	12	calculate_checksum	f, f	67
add_field	f, imm	5	srl	f, f	18	not	f, imm_64	10
add_field	f, f	11	and	f, imm	10	not	f, f	16
del_field	f	14	and	f, f	16	compare	f, f	10
add	f, imm	10	or	f, imm	10	compare	f, f	16
add	f, f	16	or	f, f	16	add_entry	f, f	75,800
sub	f, imm	10	xor	f, imm	10	set_entry	f, f	76,410
sub	f, f	16	xor	f, f	16	del_entry	f, f	45,830
sll	f, imm	12	nor	f, imm	10			

1. f is a field, imm is an immediate data; both have a length of 64 bits.

It can be seen that loading a 64-bit field consumes 6 CPU clocks when comparing the overhead of the instruction processing immediate data and the field. According to the CPU frequency (2.1 Hz) calculation utilized by the experimental platform, loading 64 bits of data takes 2.85 ns, which is quite similar to the time (3 ns) achieved in experiment 1. Despite the fact that the two experimental procedures were different, similar findings were obtained.

4.2.4. The Performance Impact of Data Location Conversion

This experiment examines the pipeline overhead for data location conversion in the southbound agent. In the experiment, the application initially requested 1 K global space from the pipeline through the controller, and then utilized the controller to deliver the FLOW MOD message to the pipeline on a continual basis. The FLOW_MOD message comprises 16 type, offset, and length data that must be converted to convert the data location. Experiments measure the speed with which FLOW_MOD messages are processed when the southbound agent converts or does not convert the data location.

Table 7 displays the results. The number of FLOW_MOD messages handled by the southbound interface agent is the same in both situations. The reason for this is because the southbound interface (POF) utilizes fixed-length FLOW MOD messages (1448 bytes), and the speed of processing FLOW_MOD messages is primarily restricted by the speed of network transmission of FLOW_MOD messages (the experiment uses a 1 Gbits I3500 network card to connect the switch and the controller). The conversion of the data location is done automatically when the FLOW_MOD message is processed to validate the format, thus it does not take much extra time.

Table 7. The impact of the conversion data location on the FLOW_MOD message loading speed.

Transfer the Data Location	Number of FLOW_MOD Processed/s
YES	8624
NO	8624

We also evaluated how long it takes to transform the data location. Converting the data location consists of two steps: (1) searching the hash table for the base address corresponding to the data type, and (2) using the base address plus the offset to determine the absolute address of the data. In our testing platform, completing these two processes takes 460 CPU clock cycles of roughly 230 ns. In the above experiment, 16 data locations in a FLOW MOD message should be transformed, and the total estimated time overhead is 3.68 us, which is insignificant when compared to the 0.115 ms required to transfer a FLOW MOD message.

In summary, data location conversion between the application and the switch transfers the process of getting the base address when forwarding packets to the table or the entry loading with very little time overhead. It avoids the issue of increased packet forwarding latency caused by locating the base address during packet forwarding.

5. Related Work

As the first data plane programming solution, OpenFlow [27] only has six instructions and 11 operations. It has minimal packet-processing capability. OpenFlow actions like copying TTL inwards and decreasing TTL are not universal and cannot be reused to process network state.

POF [29] represents packet fields with offset and length. The P4 switch [35] has a parser for matching custom protocol fields in the match-action table. The data plane can now allow arbitrary protocol matching thanks to POF and P4. However, with the exception of the packet field in the match-action table, none of them enable matching other forms of data. POF uses specific instructions to process the metadata and flow state. As a result, supporting new types of data necessitates expanded instructions, resulting in duplicated instruction functions and costly expansion costs. In the early P4 v1.1.0 language specification [36], only 19 instructions for packet processing (packet forwarding, dropping, header insert, deletion, and so on) were provided, and the language has limited capacity to process different types of data besides packets and metadata. The most recent P4 v1.2.0 [37] mostly defines the grammatical functions that P4 switches must offer, but does not describe how the switches implement these functions.

OpenState [7], FAST [30], and SDPA [25] offer additional tables to improve the data plane's ability to process network state. However, it is challenging to extend one type of table to record and update network states in the data plane [38]. To that end, OpenState provides a state table and an XFSM table. FAST introduces a state machine filter table, a state table, a state transition table, and an action table. SDPA defines three types of table: state tables, state transition tables, and action tables. FlowBlaze [31] has both a flow context table and an EFSM table. Such expansion meets specific demands but falls short of a complete examination of many types of data processing. Furthermore, introducing new types of tables necessitates expanding the southbound interface and upgrading the switch and controller protocol stacks. Furthermore, the presence of many distinct types of tables on the data plane complicates the southbound interface and makes network management more complex.

6. Conclusions

In this paper, we propose a method for adopting data type and data location to represent data in the SDN data plane, allowing the switch to handle different types of data with an unified match-action table, thereby improving data plane programmability. This data representation approach allows data loading and storage to be decoupled from data matching and instruction execution. The matching field and instruction function are no longer dependent on data type or data meaning after decoupling. The match-action table can be reused by network applications to handle user-defined data. The data plane not only supports the original type of data processing, but it can also easily support new types of data by expanding the data loading and storage module. Instructions can naturally

allow interoperability between different types of data by mixing the data types controlled by instructions.

Obtaining the data absolute address based on the data type in packet forwarding may increase forwarding latency. We proposed an application-to-switch data location conversion interaction method that stores the base addresses of various sorts of data beforehand. By calculating the required data address when switch adding the table or the entry, we solved the problem that getting the data address may increase the forwarding latency.

We built a pipeline that represents data with the data type and data location using Intel's DPDK framework. The pipeline's match-action table is independent of the matching protocol or data type. The pipeline also supports atomic instructions such as arithmetic operations, branch comparison, packet forwarding, and entry operation. The effect of data type (immediate data, packet field, metadata, flow state, global state) and data location conversion interaction on pipeline performance is investigated. The experimental findings demonstrate that the data type utilized to process the packet has no effect on packet forwarding latency. Furthermore, we transform the data location with very little time cost (3.68 us), avoiding the loss of forwarding performance caused by computing the data address in packet forwarding.

Future work will entail implementing the proposed data representation method on programmable hardware (such as an FPGA) and expanding the P4 language to express the match-action table compatible with multi-type data using the data representation approach described in this paper.

Author Contributions: Conceptualization, L.J., X.C. and J.W.; methodology, L.J. and X.C.; software, L.J. and X.C.; validation, L.J.; writing—original draft preparation, L.J.; writing—review and editing, L.J., X.C. and J.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Chinese Academy of Sciences through the SEANET Technology Standardization Research System Development grant number [No. XDC02070100].

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Masoudi, R.; Ghaffari, A. Software defined networks: A survey. *J. Netw. Comput. Appl.* **2016**, *67*, 1–25. [CrossRef]
2. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parelkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.* **2008**, *38*, 69–74. [CrossRef]
3. Tan, L.; Su, W.; Zhang, W.; Lv, J.; Zhang, Z.; Miao, J.; Liu, X.; Li, N. In-band Network Telemetry: A Survey. *Comput. Netw.* **2021**, *186*, 107763. [CrossRef]
4. Bera, S.; Misra, S.; Vasilakos, A.V. Software-Defined Networking for Internet of Things: A Survey. *IEEE Internet Things J.* **2017**, *4*, 1994–2008. [CrossRef]
5. Khan, W.Z.; Ahmed, E.; Hakak, S.; Yaqoob, I.; Ahmed, A. Edge computing: A survey. *Future Gener. Comput. Syst.* **2019**, *97*, 219–235. [CrossRef]
6. Eum, S.Y.; Jibiki, M.; Murata, M.; Asaeda, H.; Nishinaga, N. A design of an ICN architecture within the framework of SDN. In Proceedings of the 2015 Seventh International Conference on Ubiquitous and Future Networks, Sapporo, Japan, 7–10 July 2015; pp. 141–146. [CrossRef]
7. Bianchi, G.; Capone, A.; Bonola, M.; Cascone, C. Public Review for OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 44–51. [CrossRef]
8. Krongbaramee, P.; Somchit, Y. Implementation of SDN Stateful Firewall on Data Plane using Open vSwitch. In Proceedings of the 15th International Joint Conference on Computer Science and Software Engineering (JCSSE 2018), Nakhonpathom, Thailand, 11–13 July 2018. [CrossRef]
9. Alizadeh, M.; Edsall, T.; Dharmapurikar, S.; Vaidyanathan, R.; Chu, K.; Fingerhut, A.; Lam, V.T.; Matus, F.; Pan, R.; Yadav, N.; et al. CONGA: Distributed congestion-aware load balancing for datacenters. *Comput. Commun. Rev.* **2015**, *44*, 503–514. [CrossRef]
10. Lotfimahyari, I.; Sviridov, G.; Giaccone, P.; Bianco, A. Data-Plane-Assisted State Replication With Network Function Virtualization. *IEEE Syst. J.* **2021**, 1–12. [CrossRef]
11. Yang, G.; Yoo, Y.; Kang, M.; Jin, H.; Yoo, C. Bandwidth Isolation Guarantee for SDN Virtual Networks. In Proceedings of the IEEE Annual Joint Conference: INFOCOM, IEEE Computer and Communications Societies, Vancouver, BC, Canada, 10–13 May 2021; pp. 1–10.

12. Yang, G.; Yoo, Y.; Kang, M.; Jin, H.; Yoo, C. Accurate and Efficient Monitoring for Virtualized SDN in Clouds. *IEEE Trans. Cloud Comput.* **2021**. [CrossRef]
13. Yoo, Y.; Yang, G.; Kang, M.; Yoo, C. Adaptive Control Channel Traffic Shaping for Virtualized SDN in Clouds. In Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), Beijing, China, 19–23 October 2020; pp. 22–24.
14. Muqaddas, A.S.; Sviridov, G.; Giaccone, P.; Bianco, A. Optimal State Replication in Stateful Data Planes. *IEEE J. Sel. Areas Commun.* **2020**, *38*, 1388–1400. [CrossRef]
15. Zhang, M.; Li, G.; Wang, S.; Liu, C.; Chen, A.; Hu, H.; Gu, G.; Li, Q.; Xu, M.; Wu, J. Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches. In Proceedings of the 27th Network and Distributed System Security Symposium (NDSS 2020), San Diego, CA, USA, 23–26 February 2020. [CrossRef]
16. Mahrach, S.; Haqiq, A. DDoS flooding attack mitigation in software defined networks. *Int. J. Adv. Comput. Sci. Appl.* **2020**, *11*, 693–700. [CrossRef]
17. Wang, H.; Xu, H.; Qian, C.; Ge, J.; Liu, J.; Huang, H. PrePass: Load balancing with data plane resource constraints using commodity SDN switches. *Comput. Netw.* **2020**, *178*, 107339. [CrossRef]
18. Wang, H.; Xu, H.; Huang, L.; Wang, J.; Yang, X. Load-balancing routing in software defined networks with multiple controllers. *Comput. Netw.* **2018**, *141*, 82–91. [CrossRef]
19. Benet, C.H.; Kassler, A.J.; Benson, T.; Pongracz, G. MP-HULA: Multipath Transport Aware Load Balancing Using Programmable Data Planes. In Proceedings of the 2018 Morning Workshop on In-Network Computing (NetCompute '18), New York, NY, USA, 20 August 2018. [CrossRef]
20. Olteanu, V.; Agache, A.; Voinescu, A.; Raiciu, C. Stateless Datacenter Load-balancing with Beamer. In Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation, Renton, WA, USA, 9–11 April 2018.
21. Turkovic, B.; Oostenbrink, J.; Kuipers, J. Detecting Heavy Hitters in the Data-Plane. 2019. Available online: http://arxiv.org/abs/1902.06993 (accessed on 23 August 2021).
22. Caprolu, M.; Raponi, S.; Pietro, R.D. FORTRESS: An efficient and distributed firewall for stateful data plane SDN. *Secur. Commun. Netw.* **2019**, *2019*, 1–16. [CrossRef]
23. Cascone, C.; Sanvito, D.; Pollini, L.; Capone, A.; Sansò, B. Fast failure detection and recovery in SDN with stateful data plane. *Int. J. Netw. Manag.* **2017**, *27*, 1–32. [CrossRef]
24. Liao, Y.; Tsai, S.C. Fast failover with hierarchical disjoint paths in SDN. In Proceedings of the 2018 IEEE Global Communications Conference (GLOBECOM), Abu Dhabi, United Arab Emirates, 9–13 December 2018; pp. 1–7.
25. Zhu, S.; Bi, J.; Sun, C.; Wu, C.X.; Hu, H. SDPA: Enhancing stateful forwarding for software-defined networking. In Proceedings of the IEEE 23rd International Conference on Network Protocols (ICNP), San Francisco, CA, USA, 10–13 November 2015; pp. 323–333. [CrossRef]
26. Bonola, M.; Bifulco, R.; Petrucci, L.; Pontarelli, S.; Tulumello, A.; Bianchi, G. Implementing advanced network functions with stateful programmable data planes. In Proceedings of the 2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), Osaka, Japan, 12–14 June 2017; pp. 1–6. [CrossRef]
27. OpenFlow Switch Specifications Version 1.0, Open Network Fundation. Available online: https://www.opennetworking.org/wpcontent/uploads/2013/04/openflow-spec-v1.0.0.pdf (accessed on 23 August 2021).
28. DPDK. Available online: https://www.dpdk.org/ (accessed on 23 August 2021).
29. Song, H. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, China, 16 August 2013; pp. 127–132. [CrossRef]
30. Moshref, M.; Bhargava, A.; Gupta, A.; Yu, M.; Govindan, R. Flow-level state transition as a new switch primitive for SDN. In Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, Chicago, IL, USA, 22 August 2014; pp. 61–66. [CrossRef]
31. Pontarelli, S.; Bifulco, R.; Bonola, M.; Cascone, C.; Spaziani, M.; Bruschi, V. Flowblaze: Stateful packet processing in hardware. In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, Boston, MA, USA, 26–28 February 2019; pp. 531–547.
32. Bosshart, P.; Daly, D.; Gibb, G.; Izzard, M.; McKeown, N.; Rexford, J.; Schlesinger, C.; Talayco, D.; Vahdat, A.; Varghese, G.; et al. P4: Programming protocol-independent packet processors. *Comput. Commun. Rev.* **2014**, *44*, 87–95. [CrossRef]
33. Amin, R.; Shah, N.; Mehmood, W. Enforcing optimal ACL policies using K-partite graph in hybrid SDN. *Electronics* **2019**, *8*, 604. [CrossRef]
34. Open vSwitch v2.8.5. Available online: https://www.openvswitch.org/releases/openvswitch-2.8.5.tar.gz (accessed on 23 August 2021).
35. Bosshart, P.; Gibb, G.; Kim, H.; Varghese, G.; McKeown, N.; Izzard, M.; Mujica, F.; Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *Comput. Commun. Rev.* **2013**, *43*, 99–110. [CrossRef]
36. P4 v1.0. Available online: https://p4lang.github.io/p4-spec/p4--14/v1.0.4/tex/p4.pdf (accessed on 23 August 2021).
37. P4 v1.2.2. Available online: https://p4lang.github.io/p4-spec/docs/P4--16-working-spec.html (accessed on 23 August 2021).
38. Zhang, X.; Cui, L.; Wei, K.; Tso, F.; Ji, Y.; Jia, W. A survey on stateful data plane in software defined networks. *Comput. Netw.* **2021**, *184*, 107597. [CrossRef]