electronics

MDPI

Article

AccelSDP: A Reconfigurable Accelerator for Software Data Plane Based on FPGA SmartNIC

Xiaoying Huang [1,2], Zhichuan Guo [1,2,*], Mangu Song [1] and Yunfei Guo [1,2]

[1] National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, No. 21, North Fourth Ring Road, Haidian District, Beijing 100190, China; huangxy@dsp.ac.cn (X.H.); songmg@dsp.ac.cn (M.S.); guoyf@dsp.ac.cn (Y.G.)

[2] School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, No. 19(A), Yuquan Road, Shijingshan District, Beijing 100049, China

* Correspondence: guozc@dsp.ac.cn

Abstract: Software-defined networking (SDN) has attracted much attention since it was proposed. The architecture of the SDN data plane is also evolving. To support the flexibility of the data plane, the software implementation approach is adopted. The software data plane of SDN is commonly implemented on a commercial off-the-shelf (COTS) server, executing an entire processing logic on a commodity CPU. With sharp increases in network capacity, CPU-based packet processing is overwhelmed. However, completely implementing the data plane on hardware weakens the flexibility. Therefore, hybrid implementation where a hardware device is adopted as the accelerator is proposed to balance the performance and flexibility. We propose an FPGA SmartNIC-based reconfigurable accelerator to offload some of the operation-intensive packet processing functions from the software data plane to reconfigurable hardware, thus improving the overall data plane performance while retaining flexibility. The accelerated software data plane has a powerful line-rate packet processing capability and flexible programmability at 100 Gbps and higher throughput. We offloaded a cached-rule table to the proposed accelerator and tested its performance with 100 GbE traffic. Compared with the software implementation, the evaluation result shows that the throughput can achieve a 600% improvement when processing small packets and a 100% increase in large packet processing, and the latency can be reduced by about 20× and 100×, respectively, when processing small packets and large packets.

Keywords: hardware acceleration; data plane; FPGA; SmartNIC; reconfigurable accelerator

Citation: Huang, X.; Guo, Z.; Song, M.; Guo, Y. AccelSDP: A Reconfigurable Accelerator for Software Data Plane Based on FPGA SmartNIC. *Electronics* **2021**, *10*, 1927. https://doi.org/10.3390/electronics10161927

Academic Editor: Luis Gomes

Received: 14 July 2021
Accepted: 7 August 2021
Published: 11 August 2021

1. Introduction

In recent years, software-defined networking (SDN) has attracted considerable attention and consists of application, control, and data planes. The data plane performs network packet processing, including parsing, classification, modification, deparsing, and forwarding, according to the instructions given by the controller. For better flexibility and more efficient communication with the control plane, many works have implemented a data plane based on software [1–5], with a commodity CPU bearing the burden of the intensive packet processing. These software implementations have made full use of user space I/O libraries (i.e., Intel DPDK), non-uniform memory access (NUMA) architecture, and some other emerging techniques linked to the current network interface card (NIC) Application Specific Integrated Circuits (ASICs) to improve processing efficiency. However, as current CPU development has encountered a bottleneck while network processing demands continue to rise, it will become increasingly difficult for software-based data planes to meet the high-throughput, low-latency metric of network processing. As SDN develops, researchers are attempting to create smarter data planes to ease the pressure of the centralized controller when scaling up, leaving the stable local functions alone with the data plane [6–9]. All these factors place a great strain on the software-based data plane, leading

it to be overloaded. In order to enhance the performance of a data plane, one approach is to turn to hardware. Many works have implemented a data plane in an ASIC [10] and an FPGA [11,12] based on P4 specification [13]. Undoubtedly, these platforms showed an improved performance benefiting from dedicated hardware design. However, when facing flexible operations in the data plane workloads (i.e., local computing [7]), these hardware platforms were overwhelmed. A hybrid software–hardware model is thus proposed to attempt to achieve a balance between performance and flexibility. Prior works used a graphics processing unit (GPU) as an accelerator in networking applications [14,15], utilizing parallelizable processing resources in packet processing. Nevertheless, the introduced latency and memory-access overheads weaken its advantages. In many works, ASICs are also adopted to assist software to process stable forward functions [16,17]. ASICs have an obvious disadvantage as well as their advantage, which is inflexibility, because the logic is hardened in the integrity circuits and unchangeable. An FPGA consists of integrated circuits that can be programmed as user-customized logic, which is commonly used in network systems design and implementation [18–22]. Although an FPGA can well balance performance and flexibility, being an accelerator for hybrid architecture means that the FPGA is also required to bear the original NIC functions. Therefore, NIC features inevitably need to be implemented in the FPGA, which may introduce plenty of challenges, including development difficulty, resource consumption, etc. A more applicable solution is to leverage the functionality of existing NICs rather than re-deploying them on FPGAs. The FPGA-based SmartNIC (FSNIC) [23,24], first introduced by Microsoft, is exactly such a converged architecture, and it is already used in accelerating network functions (NFs) [25,26]. Here, we propose a reconfigurable accelerator for the software implementation of an SDN data plane based on FSNIC, breaking through the performance bottleneck while providing excellent flexibility. The accelerator is located in a commercial off-the-shelf (COTS) server where the data plane is running, replacing the original NIC. Users can customize it to offload some packet processing functions from the software to the accelerator. Packets are processed by the offloaded functions (OFs) in the accelerator, saving precious CPU cycles so that the overall performance can be improved. To evaluate our accelerator, we designed and implemented a use case on the accelerator. In this use case, an SDN cached-rule table [1,17] was offloaded to the accelerator, delivering the processing of packets that match the cached rules to the hardware, and the result shows that the throughput can be increased by up to 6x and the latency can be reduced by up to 100×.

Our major contributions are as follows:

- We introduce an FSNIC-based reconfigurable accelerator for software implementation of an SDN data plane. Several packet processing tasks can be offloaded to the reconfigurable accelerator, improving the data plane's overall performance. The offloaded function units (OFUs) implemented in reconfigurable fabrics can be configured through an offered OFU API or even reprogrammed as new functions by users.
- The proposed accelerator is a general-purpose accelerator for software-implemented data planes running on a COTS server. It is easy to implement if replacing the original NIC with our accelerator because the accelerator has been developed based on the FSNIC, which is essentially an enhanced NIC with an FPGA.
- We implement our design with a use case for an Intel FSNIC. The use case involves performing a cached-rule table offloading function. A detailed workflow description and a full evaluation for this use case are given.

The rest of this paper is organized as follows. The background, motivation, and design goals are given in Sections 2 and 3. The system design with the use case is detailed in Section 4. In Sections 5 and 6, the implementation and evaluation are performed. Related works are discussed in Section 7. Finally, we conclude this paper in Section 8.

2. Background

This section explains the reasons why we choose an FSNIC as the target hardware to implement our accelerator, and the overall architecture of the adopted FSNIC device, Intel N3000, is introduced.

2.1. Evaluating FSNIC as Accelerator

The software data plane uses NIC ASIC to receive and transmit packets. Most of the current NIC products support the features including VXLAN offloading, Single Root I/O Virtualization (SR-IOV), kernel bypassing, and so on. The software data plane utilized these features to accelerate its processing in a limited scope. The features are fixed and logic is hardened in ASIC-based NIC, so the NIC cannot support more diversified acceleration requirements for software data plane. After all, the architecture of the SDN data plane keeps evolving. In order to improve the flexibility of NIC, many NIC vendors have embedded multiple CPU cores into an NIC ASIC or programmable fabrics to trade off performance for a better programmability [27–30]. However, since general-purpose processors are involved in packet processing, the throughput can be degraded by up to 8x and latency can increase by as much as 80× when performing memory-intensive operations [31]. Therefore, if the network is scaled up to a 100 GbE level, the Multicore-based SmartNIC (MSNIC) may hardly reach the deployment requirements. Above all, it can draw that the hardware accelerator for software data plane must be based on the two basic conditions: good scalability to high-throughput and certain flexibility. The FPGA can meet the two requirements. An FPGA contains a great many logic, arithmetic, and memory elements, which users can configure to implement custom circuits. Thus far, many works have designed an FPGA as an accelerator, used in various fields [32]. However, an FPGA first needs to function as an NIC to be an accelerator for a software data plane. Although the FPGA proved to be a basic NIC performing basic packet transmitting and receiving functions [33,34], many other features, such as supporting for SR-IOV and DPDK, that are commonly implemented for NICs are challenging to implement on FPGAs [35]. Therefore, the FSNIC [23,24] is the best hardware to implement the accelerator for a software data plane. FSNIC integrates commodity NIC ASIC and FPGA on the same board, fully taking advantage of the existing NIC implementation rather than re-implementing these features on an FPGA. The FSNIC has an architecture as shown in Figure 1, where an FPGA is placed between the network port and the NIC, forming a bump-in-the-wire architecture. In summary, with the flexibility and performance brought by the FPGA and the tractability from the NIC ASIC, an FSNIC is surely the most suitable hardware to implement our reconfigurable accelerator.

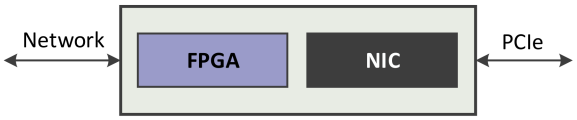


Figure 1. FSNIC architecture.

2.2. Intel N3000 Architecture

In this article, we will use Intel N3000 FSNIC to implement and evaluate our design. This section briefly introduces the overall architecture of the N3000, which is illustrated in Figure 2. The FPGA adopted was Intel Arria 10, which features industry-leading programmable logic built on 20 nm process technology and integrates a rich feature set of embedded peripherals, embedded high-speed transceivers, hard memory controllers, and so on [36]. The N3000 is equipped with two NIC ASICs, which are both an Intel Ethernet Controller XL710 (40G) [37], supporting DPDK kernel bypassing packet processing; PCI express (PCIe) virtualization, such as SR-IOV; and most advanced functionalities. The retimer is the Intel Ethernet Connection C827 Retimer, which provides tightly controlled network timing performance for the Ethernet. The Intel MAX 10 functions as the board management controller, responsible for controlling, monitoring, and giving low-level access

to board features. The communication link between the N3000 and a local host server is through a PCIe Gen3 x16 Edge Connector. The N3000 also contains a PEX8747 PCIe switch with the upstream port on the edge connector and the downstream ports connected to Intel Arria 10 FPGA and Intel Ethernet Controller XL710 devices. At the same time, the N3000 has two Quad Small Form-Factor Pluggable (QSFP) 28 cages, which support up to 200 Gbps link.

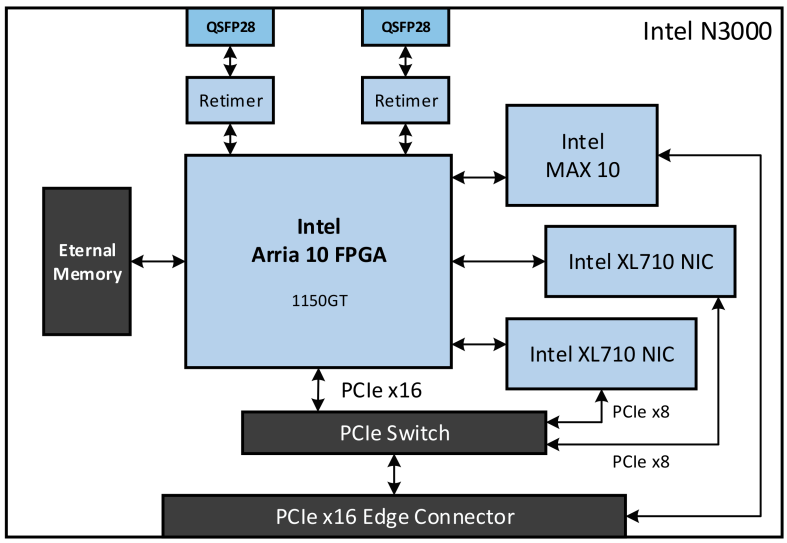


Figure 2. Overall architecture of N3000.

3. Motivation and Design Goals

3.1. Motivation

The role of the SDN data plane is to forward packets according to the instructions given by the controller. Although there are different implementations of SDN data planes due to the variation of SDN paradigms, the architectures are virtually the same. Taking the OpenFlow data plane as an example, it consists of fixed architecture switches. The working principle of the switch is to simply forward packets between ports according to the contents of several flow tables defined by the controller and modify the metadata of the packets to a certain extent. Each flow table includes multiple entries, and each entry specifies the actions to be performed when a packet matches the rule indexed by the entry. Since the flow tables and processing pipelines within SDN switches can be very different as the network application changes, deployment by pure software is a better way to achieve flexibility and programmability of the data plane. Pure software-based data plane implementations are characterized by excellent flexibility due to the high level of programmability and configurability of the forwarding functions [38]. However, with the rapid increase in the size of deployed in-network traffic, CPU-dependent software data planes experience bottlenecks in performance that are difficult to break through. Naturally, hardware acceleration is proposed to try to address the problem of limited performance of the software data plane. Pure hardware implementation is known to have a lack of flexibility and programmability, thus losing the benefits that SDN brings. Even a highly flexible programmable solution such as P4 specification and its hardware deployment has many limitations in its application. Thus, it can be seen that by using an FSNIC to offload some functions of the software data plane and adopting a hardware and software co-design, the programming flexibility of the data plane can still be retained, while improved performance of line-rate packet processing can also be achieved with the bump-in-the-wire design of the FSNIC. Additionally, software-based switches are often deployed on high-end multicore COTS servers to achieve high performance, which usually consume extremely large amounts of power. Offloading the pressure of line-rate processing to an FSNIC

saves the CPU cores of the server so that a better performance can be achieved in a more cost-efficient way.

3.2. Design Goals

Utilizing an FSNIC in the SDN data plane as an inline reconfigurable accelerator, we aimed to achieve the following goals:

- Improve throughput and reduce latency significantly: Even if kernel bypass technology (such as DPDK) is used, the packet processing performance is still unsatisfactory when the throughput increases above 40 GbE. The bump-in-the-wire FSNIC can achieve 100 GbE or higher throughput with ultra-low latency, and this is the reason why we adopted it as the accelerator for the current software implementation SDN data plane.
- Maintain data plane programmability: Another main reason for using an FSNIC is to maintain the programmability and flexibility of the data plane as much as possible. ASICs can be lightly configured based on reserved registers, but they function poorly when the data plane is updated with new matching rules and instructions. For the FSNIC, only slight reconfiguration of the hardware logic of the FPGA is required to add new functions.
- Be easy to implement: Since the FSNIC is an enhanced NIC that has the same interfacing method as the traditional NIC, software implementation does not need to modify existing architecture.
- Save CPU cores: Achieving a high throughput of line-rate packet processing often exhausts system resources, including a large number of CPU cores and memory. Offloading some operation-intensive packet processing on FSNIC can prevent the burning out of more CPU cores.

4. System Design

As shown in Figure 3, our system design consists of two main parts: software design and hardware design. The software SDN switch implemented in the COTS server is in charge of configuring and managing functions offloaded in the FSNIC accelerator. Drivers in the host interface with the FSNIC in the data path and control path. The hardware is responsible for handling the offloaded packet processing tasks. Multiple OFUs can be implemented in the FPGA, and the user can decide which OFU is activated through the OFU API supported by the FPGA driver. If a new OFU is planned to extend the existing design, the user just needs to add the new OFU logic to the existing hardware logic and offer the base address of the register space of the OFU to the driver for further configuration usage. The remainder of this section describes the design details of each component of our architecture, while the cached-rule process offloading is introduced as the use case of our design.

4.1. Software Design

The packet stream from or to the NIC is handled by a kernel bypass driver (generally DPDK driver) to ensure high performance and low CPU involvement. The management and configuration for FPGA are taken over by the FPGA driver. The FPGA driver offers access to the FPGA internal registers by using Memory Mapped I/O (MMIO) access, which is an efficient way of accessing the PCI device configuration space from the host user space. FPGA vendors currently all offer the implementation solution for internal register access, such as Advanced eXtensible Interface 4-Lite (AXI4-Lite) for Xilinx [39] and Core Cache Interface (CCI-P) with Avalon for Intel [40]. Moreover, we offer mutually isolated register spaces for different OFUs, and global management and monitoring registers for the FPGA are also supported. Based on the basic register operations, the API for OFU configuration can be implemented. For example, if an OFU is implemented as a reconfigurable match-action table (RMT) [41], the user can use the driver to add a new entry and modify or delete an existing entry in the match-action table (MAT).

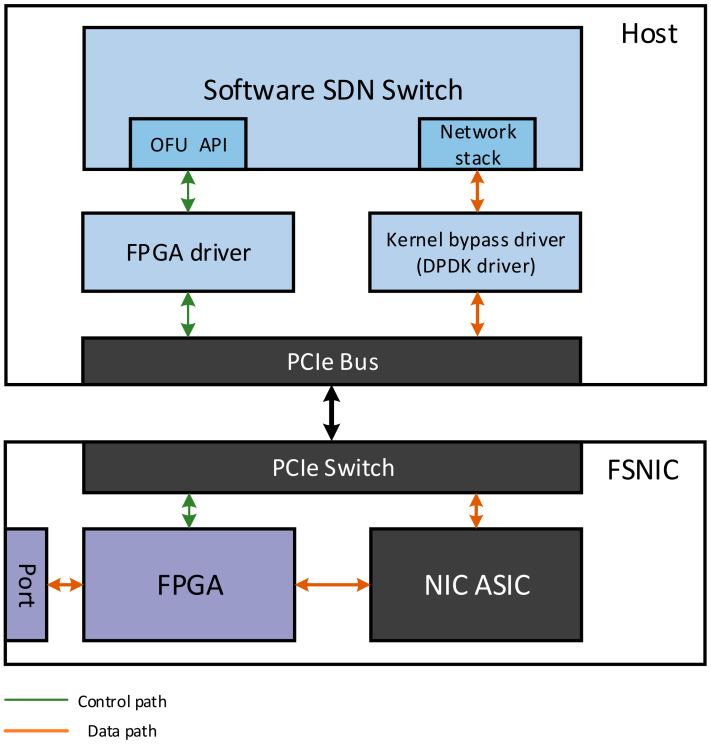


Figure 3. Overall architecture of our design.

4.2. Hardware Design

Figure 4 illustrates the hardware design architecture. The FPGA is located between the network port and the NIC ASIC as an inline reconfigurable fabric. The data path within the FPGA is distinguished into two directions: ingress and egress. The ingress stream is from the external port, vectoring to the software switch, while the egress stream targets the external port from the host. Packets from the ingress stream can be pre-processed with the OFU logic defined by the user. Similarly, the user can program a post-processing OFU for the egress packets. In most cases, the ingress packets that have completed offloaded function (OF) processing will converge with the egress packets and will be output to the corresponding port.

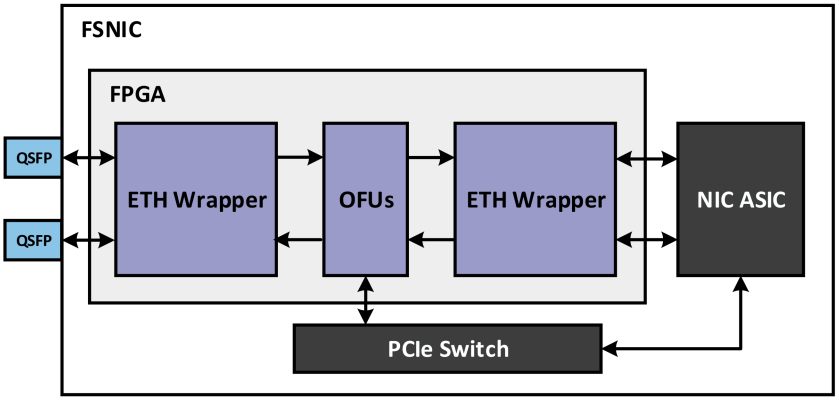


Figure 4. Hardware architecture design.

In order to ensure the efficiency and operability of data transfer between the various components in the FPGA, an Avalon interface is used for data transferring in the FPGA. The Avalon Streaming Interface is one of the Avalon interfaces that is specifically used for flow data transfer, including multiplexed streams, packets, and DSP data [42]. In our design, we only use the packet data transfer mode. The signals used to support this

mode include channel, data, error, ready, valid, empty, endofpacket, and startofpacket. Here, channel indicates which port (lane) this packet comes from or is about to output to, and valid indicates the validity of the data signal for the current cycle. The empty signal indicates the number of symbols that are empty during the current cycle. The endofpacket and startofpacket signals indicate the end and start of the packet, respectively. A timing waveform transferring a 317-byte packet using the Avalon Streaming Interface is illustrated in Figure 5. At t_0, the signal valid and startofpacket are both high, which means a packet transferring begins. Data transfer occurs on time point t_0, t_1, t_3, t_4, and t_5, when both ready and *valid* are asserted. During t_5, endofpacket is asserted, and empty has a value of 3. This value indicates that this is the end of the packet and that 3 of the 64 symbols are empty, which means the high-order byte, *data* [511:24], drives valid data at t_5.

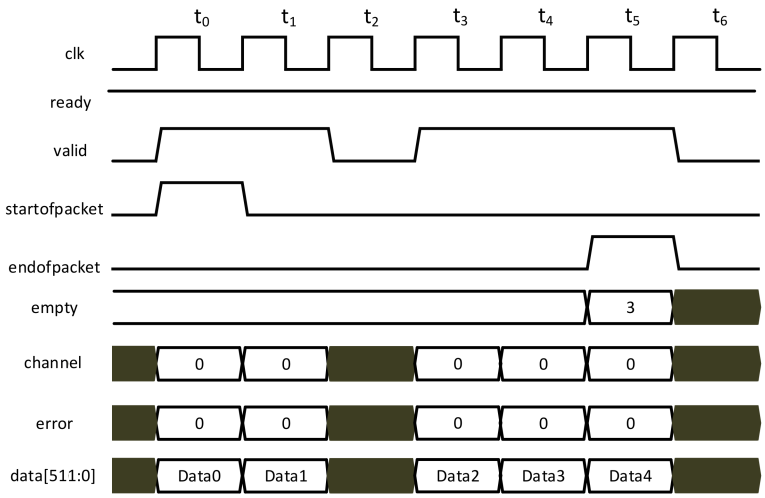


Figure 5. Timing waveform of Avalon interface working in packet transfer mode.

The data path within the FPGA is described in Figure 6. The data stream from QSFP28 (Quad SFP28) is recovered and re-clocked as four independent data lanes, which work in different clock domains. In the Eth Wrapper module, the CDC FIFOs are used to deal with the Clock Domain Crossing (CDC) issue, synchronizing the working frequency of the quads. As shown in Figure 7, the CDC FIFO is implemented by a Double Clock (DC) FIFO, whose input clock frequency is the operating frequency of the Ethernet interface and output clock frequency is a user-defined frequency. In order to recover all signals of the Avalon interface when outputting, the data widths of the FIFO input and output are both the sum of the data width of each signal in Avalon interface. During FIFO outputting, the output data are then reverted to the Avalon interface, keeping each signal of the Avalon interface in sync. Meanwhile, multiple lanes are aggregated into one stream by a configurable scheduler for more easily programming the data path in the user-defined OFU logic. The aggregated data stream has a 512-bit data width and is sufficient to carry a 100 GbE network data stream as long as it works at a clock frequency higher than 200MHz. In the opposite direction, once the data stream leaves the user-programmed logic, the arbiter will assign each packet to a specific Ethernet transmit channel according to the destination port previously decided in the OFU. The ingress OF pipeline can function as an RMT—which can cope with many packet processing functions but is a bit resource-intensive to deploy—or just implement some dedicated functions, such as intra-server load balancer (flow director), specific flow processing acceleration, etc. The egress OF pipeline can implement fast hardware packet duplication for multicast semantics support or utilize port-level Quality of Service (QoS).

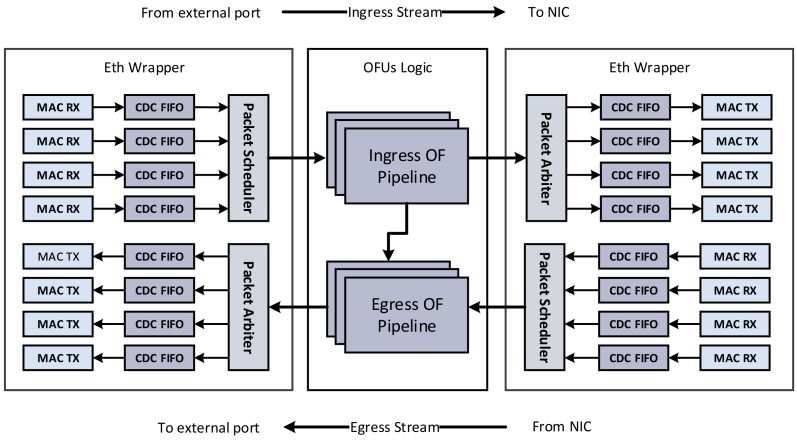


Figure 6. Data path in FPGA.

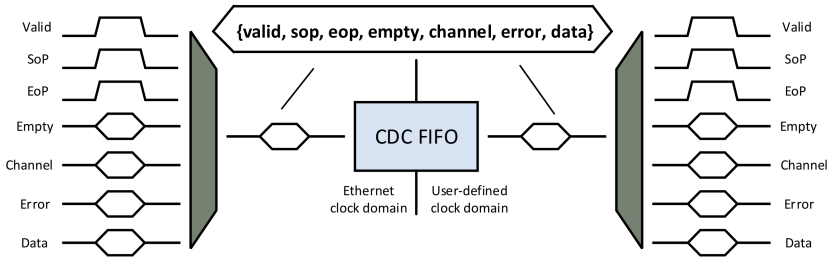


Figure 7. Design of CDC FIFO.

4.3. Use Cases

Current software SDN switches suffer from poor performance of multi-stage flow table lookups and local computing. Hence, rule caching is proposed to reduce the redundant lookup overheads. However, generating a rule for every processed packet leads to ultra-large match fields and a large increase in flow table entries. A more targeted caching policy can balance resource consumption and processing performance. Moreover, offloading the processing of packets that match the cached rules to the hardware accelerator can allow for making full use of parallel hardware and improve performance. Rule-caching schemes can vary according to different deployment scenarios [43]. The choice of caching policy is beyond the scope of this article; thus, we simply implement cached-rule tables with two different kinds of caching policies on the accelerator as use cases.

4.3.1. Offloading LPM-Based Cached-Rule Table for IP Routing Acceleration

This policy focuses on the IP routing relative processing, which caches the rules with only one match field—that is, the destination IP address. The rule table performs longest prefix matching (LPM), and the matched packets will complete a set of specified actions, including metadata modification and forwarding. To some extent, this rule table works like a transformed Forward Information Base (FIB). The establishment of the FIB-like rule table is usually completed by the cooperation of the control plane and the data plane, and it updates when a new rule is added. The packets from the ingress stream always look up the cached-rule table, and the packets matching the rule quickly perform the actions, while those that missed continue to perform multi-stage flow table lookups. However, the software switch needs to run the LPM algorithm for IP address lookup, which is also time-consuming in software, although many optimization methods have been proposed to improve lookup performance [44]. At the same time, packet forwarding in software is a dumb operation but highly CPU-intensive. It is widely accepted that hardware is capable of performing LPM and forwarding packets at line rate with ultra-low latency [35,45]. Therefore, it is meaningful to offload the LPM-based rule table on the FPGA as an OFU logic, while the software only needs to update the table using the OFU API. After offloading,

all packet processing related to the rule table is taken over by the FPGA, which can prevent overuse of the CPU cores to perform the IP searching and packet processing. Since the resource consumption linked to hardware implementation of the LPM table is extremely relevant to the width of the table [46], we do not implement the entire LPM table with the maximum IP width—that is, 128 bits for IPv6. Dividing the entire LPM table into two smaller tables with widths of 32 and 128 bits makes the implementation of LPM more resource-efficient. In this design, we use an RMT to implement this OFU. The RMT was first proposed in [41] and is now widely used in programmable switches [10]. The work flow of an RMT is described in Figure 8. The RMT pipeline first extracts the packet information of incoming packets and then processes them with a series of match action (MA) tables. Each MAT matches the specified packet header field and then performs a series of actions to the header. After that, the deparser combines the modified packet header with the original packet payload. Since the actions used in this OF are not very complicated, having too many MA stages seems slightly redundant. We therefore used a simplified version of the RMT with only two MA stages to implement the LPM rule table OFU logic.

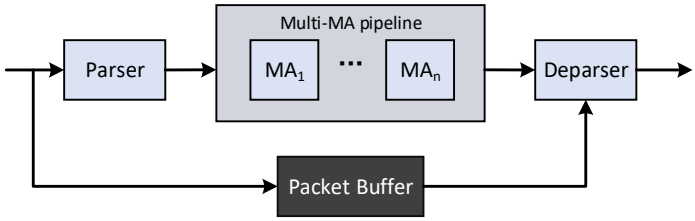


Figure 8. RMT pipeline.

Figure 9 shows the details of the hardware implementation of this OFU logic. The packets from the ingress stream are first parsed by the packet parser to generate a series of packet information whose data structure is introduced later, and then, the packet information and the packet data are fed into the FIFOs to prevent packet loss due to the MAT configuration process. Once the MATs are ready, one piece of packet information is read out and input into the MATs for processing. For the packets that match the rules existing in the rule table, the MATs recognize them and carry out a series of actions that are described in the rules, while mismatched ones simply pass through the module. For some actions, such as forward and drop, that cannot be performed immediately, the MA logic will modify the state field in the packet information for follow-up processing in the deparser module. After that, all packets' header information and original packets' data from Packet FIFO are combined in the deparser module. The output interface of the deparser is reverted to the Avalon streaming interface, and a specific signal is modified (if needed) according to the state field in the packet information that was previously modified in the MA logic. After deparsing, the mismatched packets are sent to the host for further processing, while the matched ones are fed into After MAT FIFO. Finally, the stream merger that simply performs a round-robin algorithm merges the packets from ingress and egress into one stream to the output port. A detailed description of each module is given later in this section.

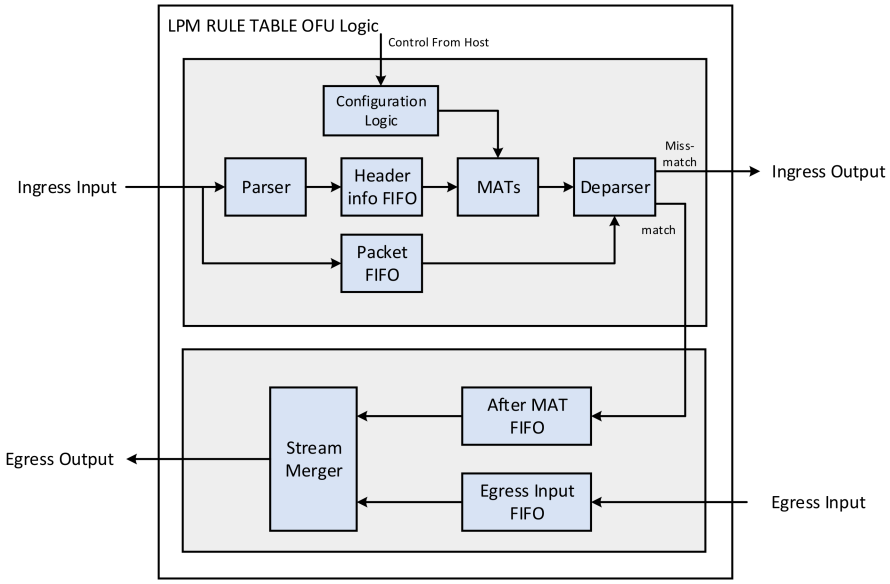


Figure 9. Logic design of the LPM-based rule table OFU.

Parser and Deparser

The parser is used to extract the packet information, including the MAC address, Ethernet type, IP address, etc. Since the packet processing mainly focuses on header information modification, extracting the header information from the packets to perform the follow-up MA logic while leaving the original packet data in buffer can significantly scale down the logic consumption.

Additionally, to better record the state of the packet during the MA pipeline and deparsing, a custom data structure is proposed. As Figure 10 illustrates, Hdr_info is the extracted packet information. The State_info field is used to store the packet state, including to-host, drop, and forward, which can be modified in the follow-up MA logic. In particular, in the case that the state is set to be forward, the output port is also attached to the State_info field. Hdr_len shows the current length of the header information. Generally, the last two fields are just for the deparser to better recover each signal of the Avalon interface based on the state of the packet.

Figure 10. Custom data structure of packet information.

The role of the deparser is to assemble the packets with the modified packet header and the packet payload, compatible with the Avalon interface, while the specific signal of the Avalon interface (e.g., channel) is modified according to the packet state labeled in the State_info field of the packet information. If the packet is tagged with drop, the valid signal is zeroed, whereas for a packet whose state is marked as forward, the deparser will modify the channel signal as the given port number. Furthermore, the deparser does not make any changes to those packets that need to be uploaded to the host. In addition to the State_info field, the Hdr_len field is used for accurate packet recovery to prevent data loss due to packet header length modification during the MA pipeline.

MAT Design

As shown in Figure 11, the MA logic includes two MA stages. The first stage matches the IP version of the packet, and the action decides which LPM table the destination IP address of this packet should look up. The second stage performs the LPM to the destination IP address. If it is a hit, the match result is used as the address to retrieve the

action set that describes the series of actions to be performed, stored in the action RAM. After that, the action set and the packet data are sent to the action units to execute the corresponding actions. The support actions include drop, set field, etc. If a packet does not match any rules in the existing table, it just passes through. A full introduction of each part of the MA logic is given as follows.

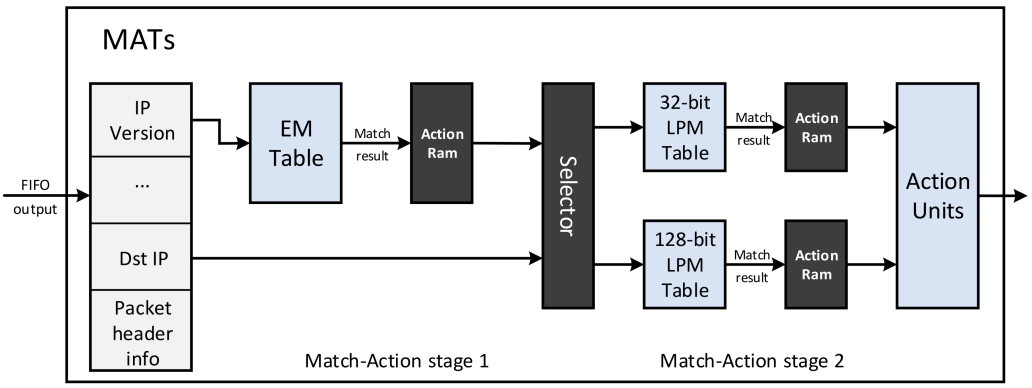


Figure 11. MA logic design.

In this design, two kinds of match tables are used in MA logic. The Exact Match (EM) table is used to match the IP version information and direct the packet information to the right LPM table for looking up. Accordingly, a very small-sized Binary Content-Addressable Memory (BCAM) is sufficient to implement the EM table. The register-based method is a better way to implement a small-sized BCAM with high performance and low resource consumption. As for the implementation of the LPM table in the FPGA, Ternary Content-Addressable Memory (TCAM) is used. Hardware-implemented TCAMs are able to search the whole memory space for a specific value within a single clock cycle due to their ability to perform a massively parallel search. There are two main implementation approaches of TCAM, namely registers-based and RAM-based, and the RAM-based solution is more applicable for large-scale implementation [46]. RAM-based TCAM is always implemented using the transposition method. RAM is addressed by the data of each entry in the TCAM, and each bit of the RAM data indicates the presence of that entry. The position of the data bits corresponds to the address (also called the entry ID) of the TCAM. For example, if the address A of the RAM stores a binary number '0010', it means that the second entry of TCAM with a depth of 4 contains A. Next, we analyze the RAM resource consumption. Suppose there is TCAM with width W and depth D. Then, the RAM size required to implement this TCAM by transposition method is $(2^W \times D)$ bits. However, when the width of the TCAM becomes wider, the required RAM size also increases dramatically. To solve this dilemma, the concatenating approach is used. Dividing a wide TCAM into multiple narrower equal-sized TCAMs can efficiently reduce the RAM consumption. If a TCAM with a width of 8 bits and a depth of D is used as a concatenation cell, which consumes a $(2^8 \times D)$-bit RAM, the RAM size used to implement a $W \times D$ TCAM is reduced to $[W \div 8](2^8 \times D)$ bits.

To achieve LPM, a priority encoder (PE) is required to encode several match lines (MLs) into a specific entry index. Figure 12 illustrates the design overview of our TCAM-based LPM table. The input lookup data are divided into multiple bytes and input several TCAMs for parallel searching. The output MLs are sent to the LPM PE along with the corresponding priority stored in the priority RAM to find the only 'winner' with the highest priority. The priority in LPM is the length of the prefix.

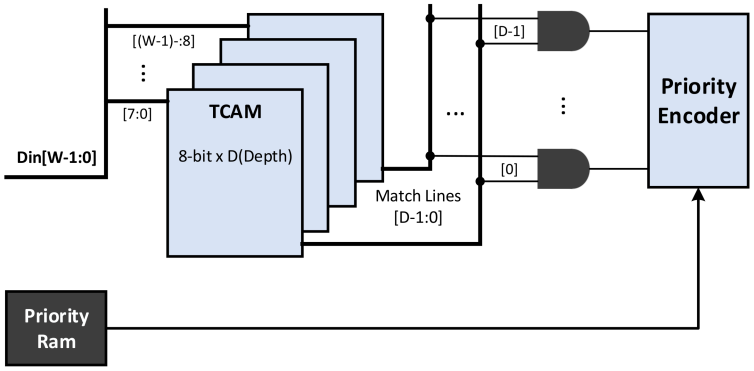
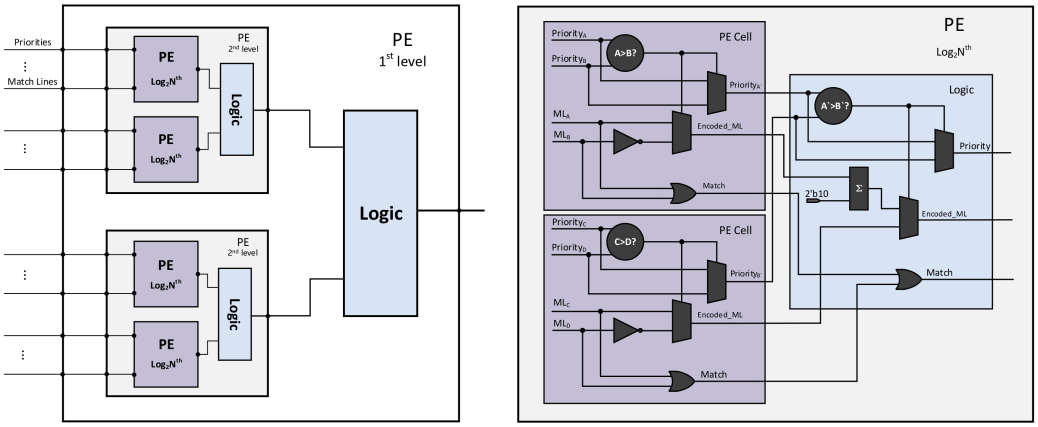


Figure 12. Design overview of our TCAM-based LPM table.

The LPM PE was designed with a recursive style to achieve high area efficiency and low latency. As illustrated in Figure 13a, the input MLs are partitioned into two equal-sized parts and then fed into the next-level PE, repeating the partition until the $\log_2 N^{\text{th}}$ level. At the $\log_2 N^{\text{th}}$ level, the inputs are only two MLs and the corresponding priorities, and the ML with a higher priority is elected and participates in the upper level encoding along with its priority as input. The recursion repeats until the 'winner' is crowned in the first level. Figure 13b shows the encoder logic in the $\log_2 N^{\text{th}}$ level. By introducing the priority RAM, the PE judges the winner depending on the input priority. Therefore, this PE does not require the TCAM to reprioritize all the entries when including a new entry, which massively reduces the cost of the TCAM.



(**a**) (**b**)

Figure 13. Design of the recursive-style PE. (**a**) Design overview of PE; (**b**) design of the $\log_2 N^{\text{th}}$ level PE.

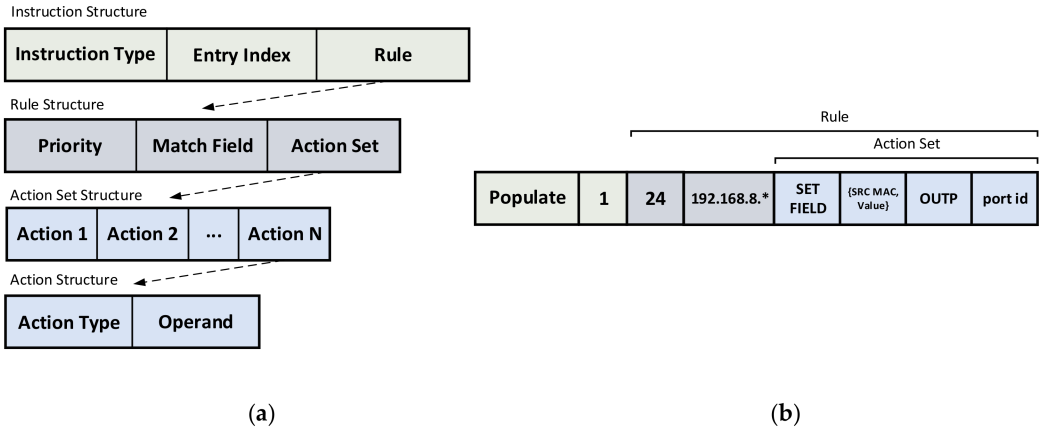
Each entry of the LPM table has a corresponding action set stored in the action RAM, and the action set describes the actions needing to be performed and the required operands of each action. The action types and the required operands are listed in Table 1.

Table 1. Support action types.

Action Type	Description	Required Operands
SUB	TTL decremental	Void
DROP	Drop the packet	Void
SFD	Set the specific field	{field name, value}
CAL	Calculate checksum	Void
INSERT	Insert a field	{field name, value}
DEL	Delete a field	field name
OUTP	Forward to a specific port	Port id

Configuration Logic

Configuration logic is used to configure the MATs, including initialization, adding rules, deleting rules, and updating rules. The instructions and rules are generated by the software and downloaded to the FPGA using the given API. The instruction structure is shown in Figure 14a. Instruction Type indicates the operation mode, including initialize, populate, delete, and update. The Rule structure consists of Priority, Match Field, and Action Set. The Action Set literally consists of several actions, and each action is composed of the action types and operands listed in Table 1. Figure 14b shows an example of an instruction. Once the instruction is given, the configuration logic will parse it into separate operations, including configuring the match tables that are equivalent to the TCAMs in this design and block RAMs (BRAMs) performed as action RAMs. Initialization will delete all the existing rules by zeroing the LPM table and RAMs. When adding a new rule to the MAT, the match field and its priority are written into the TCAM-based LPM table and the corresponding action set is stored in the action RAM. Deleting a rule clears the data in the TCAM and RAM with the address of the given entry index. If an update operation is needed, the new action set is written into the action RAM using the given entry index as the address.



(**a**) (**b**)

Figure 14. The structure of instruction. (**a**) Detailed structural composition of the instruction; (**b**) an example of the instruction.

The main components of the configuration logic include an instruction FIFO and a finite state machine (FSM). The instruction FIFO is used to cache the instructions to prevent instruction loss due to a mismatch between the write rate and execution rate of instructions in extreme cases. If the FIFO is full, new instructions fail to be written until a previous instruction is completed. The FSM is illustrated in Figure 15, containing five different states, namely Reset, Idle, Write, Delete, and Update. The Reset state is used to reset the TCAM and action RAM—that is, to assign a value of zero to all addresses. The *Idle* state is used to wait for a new instruction or for configuration process completion. The Write state is activated when adding a new rule. The match field is written into the TCAM according to the given index and mask, and the priority and action set are stored in the priority RAM and action RAM, respectively. As updating a rule will not change the match field information, the Update state only asserts the write enable of the action RAM to update the action set of the specific rule. Lastly, the Delete state is used to delete an existing rule by clearing the data stored in the TCAM and RAM with the address of a given entry index.

FIFOs

In this design, five synchronous FIFOs are used, and they all work in the same clock domain with a user-defined frequency. The Header Info FIFO and Packet Data FIFO are used to temporarily store data from the ingress when the MATs are unavailable due to the configuration process. The data width of the Header Info FIFO is equal to the packet information data structure, and the data width of the Packet Data FIFO is the sum of

each signal width defined in the Avalon interface, as shown in Figure 7. The After MAT FIFO and Egress Input FIFO buffer the data to the egress to avoid packet loss due to the competition of two data flows from different directions to the egress. The data width of these two FIFOs is the same as that of the Packet Data FIFO, and they all have a working mechanism similar to that of the CDC FIFO introduced above.

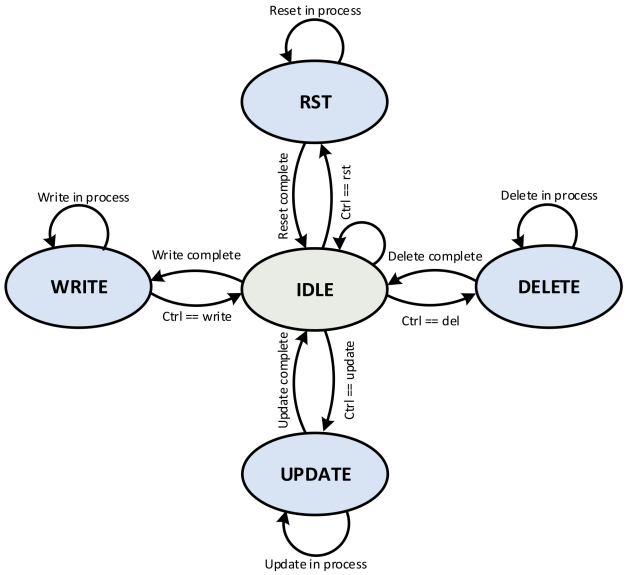


Figure 15. MAT configuration FSM.

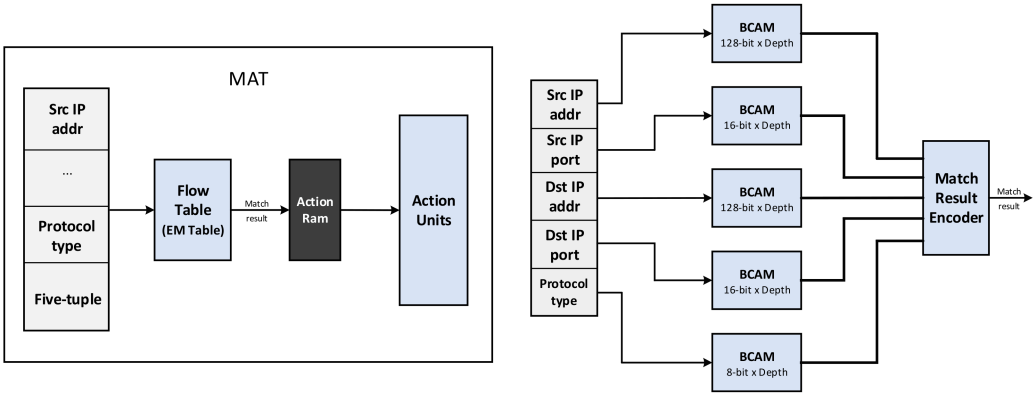
4.3.2. Offloading EM-Based Cached-Rule Table for Specific Flow Processing Acceleration

Section 4.3.1 introduces a use case of our acceleration paradigm, where an LPM-based rule table is offloaded to the FSNIC to achieve high-throughput and low-latency IP routing processing. In this context, EM-based rule tables are also adopted as an effective caching policy in many scenarios, such as public cloud [25]. The rule table caches a series of flows identified with EM and their actions after matching. The workflow of the rule table is introduced next.

Generally, every packet in a network can be recognized as a sub-part of a flow. Passing through the first packet of a flow to software and offloading the per-flow policy to hardware to MA rules is a quite typical flow-based acceleration solution for software data planes [1]. Hardware, such as an FSNIC, can process packets by matching the identified flow rules onboard without software involvement. The flow is identified by a series of packet information, including MAC address, IP address, VLAN ID, etc., and the parsed packet data are used as the match fields of the flow table to search for matching packets. The matched packet is processed according to the corresponding actions directly, and others are sent to the software for generating new rules. This method works well when an elephant flow is dominant in the network.

This function can be designed as an OFU in our FSNIC-based accelerator, which has a similar architecture to the LPM rule table OFU illustrated in Figure 9, except for the MAT. The MAT design is shown in Figure 16a. The parser firstly extracts a series of header information, and a five-tuple (source IP address, destination IP address, transport protocol number, source port number, and destination port number) is used in this design. After that, the five-tuple is used to look up a flow table to search for a matching flow entry cached in the flow table. The flow table is designed based on BCAMs, because the flow identification is an EM. Multiple BCAMs are concatenated to build a larger BCAM to support more match fields in a more area-efficient way, and each BCAM is responsible for the lookup of one match field, as shown in Figure 16b. If a packet does not match any entry in the flow table, it is vectored to the software to decide whether a new rule should be generated. If so, the rule is downloaded to the FPGA via configuration logic. The matched

packets will be processed according to the action set retrieved from the action RAM with the match result of the flow table as the address.



(**a**) (**b**)

Figure 16. Design of MAT and flow table in EM rule table OFU. (**a**) Design overview of MAT; (**b**) concatenating design of flow table based on multiple BCAMs.

4.3.3. Other Use Cases

Two use cases are introduced above. They both accelerate the processing of the software data plane by offloading the cached-rule table onto the FPGA, allowing the FPGA to take over the processing of packets that match the rules in the table. These two use cases are based on different rule-caching schemes, namely an LPM-based scheme and an EM-based scheme. However, other forms of rule table generated by various caching policies can also be deployed on our acceleration architecture.

Additionally, many other NFs on a data plane can be implemented on our accelerator, such as VXLAN encap/decap, receiving flow steering for better intra-server load balancing, egress-oriented port-level QoS policy offloading, or hardware packet replication for multicast semantics. Furthermore, some application-layer functions can even be abstracted as OFUs to implement on our accelerator.

5. Implementation

We implemented our accelerator with the two OFUs introduced above for the Intel N3000 FSNIC. Detailed descriptions of the software and hardware implementation are given in this section.

5.1. Software

The data path between the host and the board was driven by an NIC driver offered by Intel, supporting both a kernel driver and a DPDK driver. The control of OFUs was supported by a self-designed API including several functions, described in Table 2. These functions were all implemented based on the register read–write interface offered by the Intel FPGA driver [47]. Although the FPGA driver is offered as a Linux driver, the API can also be invoked from DPDK applications as long as the base code of the API and FPGA driver are compiled into the DPDK source code as a library.

Table 2. Offered API and description.

Function	Description
int ofu_init(uint16_t ofu_id)	Enable an OFU and initialize it
int rule_table_ops(uint16_t ofu_id, struct ofu_istr *istr)	Operation for rule table
int ofu_uninit(uint16_t ofu_id)	Disable an OFU

5.2. Hardware

The selected hardware implementation platform is Intel N3000, which is previously introduced in Section 2.2. The hardware design is developed by Verilog/SystemVerilog, synthesized and implemented with Intel Quartus 19.2.

In our hardware design, the Ethernet wrapper module integrates Intellectual Property (IP) cores and logic to support QSFP28 optical modules and interface with XL710 NICs. In our implementation, only one QSFP28 port was used due to the throughput limitation of sixteen-lane Gen3 PCIe interface, which provides a 100 GbE link that is composed of four lanes of 25 GbE stream. The Low Latency 25G Ethernet Intel FPGA IP was used to handle the data coding in the physical layer and data link layer, and the transmission interface was an Avalon streaming interface. Similarly, the Low Latency 40G Ethernet Intel FPGA IP was used to interface with the XL710 NIC. The Intel Arria 10/Cyclone 10 Hard IP for PCI Express was used to interface with PCIe hard logic using the Avalon interface. The CDC FIFO was implemented with the asynchronous mode of the FIFO Intel FPGA IP, and other FIFOs that do not have the CDC issue were used for the synchronous mode of the FIFO Intel FPGA IP. All the FIFOs had a BRAM-based architecture. The RAMs used to build action RAMs and RAM-based CAMs were all BRAMs, auto-generated by Intel FPGA Altsyncram. Furthermore, several other underlying functional IP cores were also used for clock generation and data interconnection.

The RAM-based CAM used in our design was developed based on an example from Altera Application Notes [48]. Although the RAM-based method has a better area efficiency, it still consumes a lot of logic and BRAM resources when scaling up. Figure 17 plots the relation between width, depth, and resource consumption when implementing RAM-based TCAM on Intel N3000. The main ordinate represents the consumption of adaptive logic modules (ALMs), which are the basic building block of supported Intel FPGA device families, and each ALM is composed of two adaptive look-up tables. The secondary ordinate indicates the consumption of BRAM in Kb.

We implemented two OFUs, introduced in Section 4.3, on the FPGA. They both operated at 200 MHz clock frequency. For the LPM rule table OFU, we selected 1K as the number of rules, so a (128×512)-bit LPM and a (32×512)-bit LPM should be implemented. Taking the (128×512)-bit LPM as an example, its implementation requires a (128×512)-bit RAM-based TCAM, which consumes a 2M-bit BRAM and a nine-stage PE logic. The width of instruction was set to be 128 bits; hence, a 128K-bit action RAM was needed. The Header Info FIFO was configured as (600×256) bits, and the other three were (522×256) bits. For the EM rule table OFU, we also chose 1K as the capacity of the rule table. This OFU shares a similar architecture with the LPM rule table OFU except for the match table. The EM table was built based on BCAMs, which only have a slight difference in write logic with the TCAMs. Furthermore, the PE logic was also excluded. Except for these, the implementation was roughly the same as with the LPM rule table OFU introduced above.

The resource utilization of our design is shown in Table 3. It can be seen from the utilization that the match tables always consume the most resources, both ALMs and BRAM, and it becomes more apparent when the number of rules increases. Replacing the LPM table and EM table with commercial LPM and EM IP cores [49,50] may offer better area efficiency.

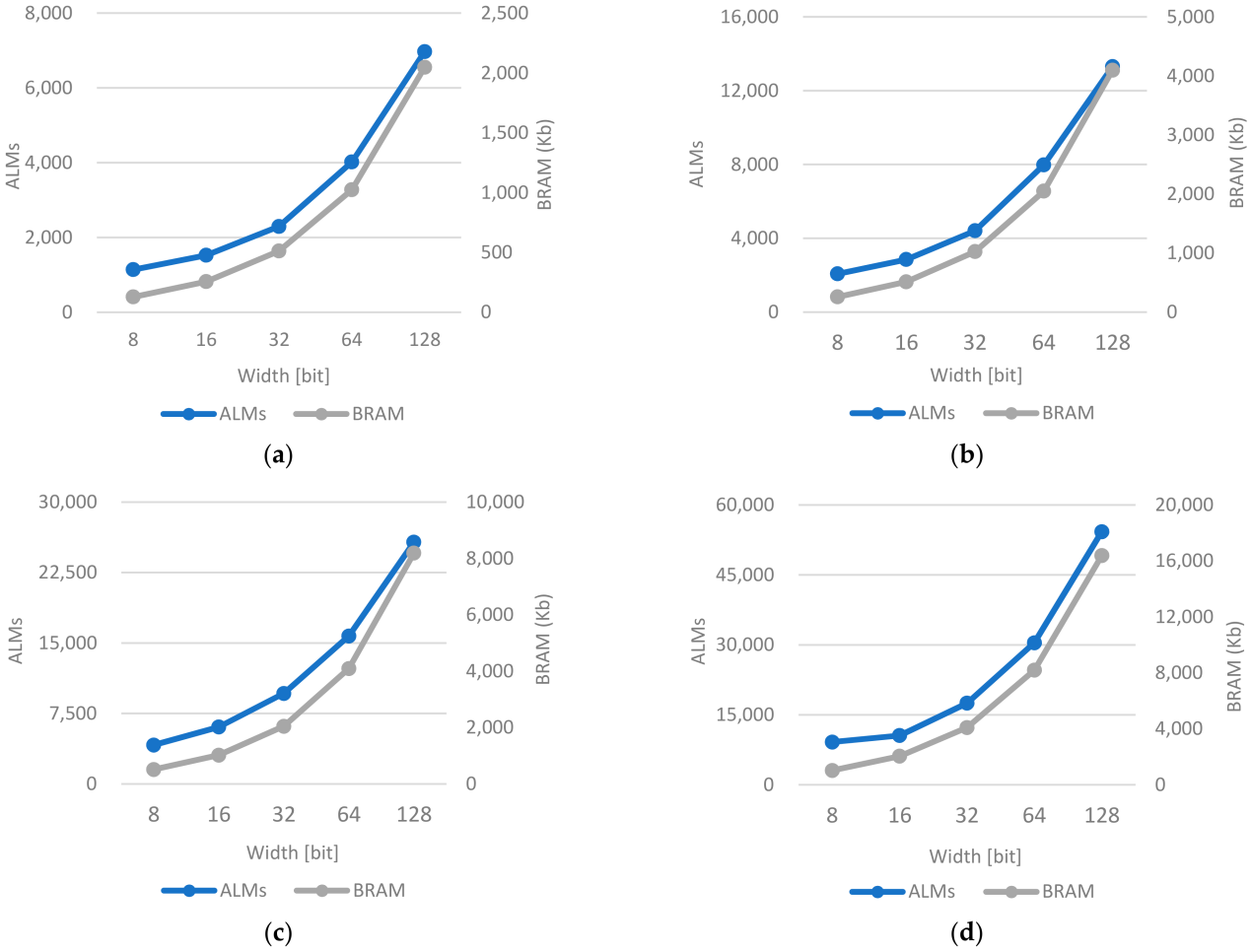


Figure 17. Resource consumption of RAM-based TCAM with different sizes. (**a**) Resource consumption with 512 depth; (**b**) resource consumption with 1024 depth; (**c**) resource consumption with 2048 depth; (**d**) resource consumption with 4096 depth.

Table 3. Utilization.

Module			Area (% of Total)	
			ALMs	**BRAM**
System	Eth Wrapper		19.7%	9.2%
	Vendor Shell		3.7%	0.6%
OFUs	IP Routing Acceleration	LPM Table	3.9%	5%
		Others	0.6%	0.8%
	Flow Processing Acceleration	EM Table	6.9%	17.7%
		Others	0.6%	0.9%

6. Evaluation

Our evaluation was performed under the assumption that the cached-rule table had already been generated, because assessing which caching policy is used to generate which kind of cached-rule table is beyond the scope of this paper. We evaluated the performance of the two OFUs introduced in Section 4.3 to see how much improvement can be achieved by offloading specific packet processing to the FSNIC. The evaluation is organized as follows: first, we evaluate the hardware implementation of the matching methods involved in the OFUs including LPM and EM, then some hardware-implemented actions, and finally

the overall performance. The software implementation used for comparison run on a high-end server to avoid unbalanced evaluations.

6.1. Hardware Setup

The Intel N3000 was deployed in a Dell R740 commercial server with two Intel Xeon Gold 5218 2.30 GHz CPUs and 128 GB RAM, interfaced with a sixteen-lane Gen3 PCIe that provides a nominal duplex data rate of approximately 128 Gbps. The QSFP28 port of the N3000 was connected with the Ixia XGS12 network test platform, and only one port was activated, providing a 100 GbE traffic link. The hardware setup is shown in Figure 18.

Figure 18. Hardware setup.

6.2. Match Tables

To evaluate the performance of the LPM table introduced in Section 4.3.1, we developed a simple test module that parses the destination IP address from pure IPv4 traffic and uses it as a key to look up the LPM table. The implemented LPM table had a size of (32×1024) bits. After programming the N3000 with the test binary file, four entries were written into the LPM table via software, which is depicted in Table 4; then, the Ixia tester generated traffic with a customized frame structure, shown in Figure 19.

Table 4. Example of an LPM table.

Entry ID	IP address	Mask
0	34.34.0.0	FFFF0000
1	17.17.17.17	FFFFFFFF
2	17.17.17.0	FFFFFF00
3	34.34.34.34	FFFFFFFF

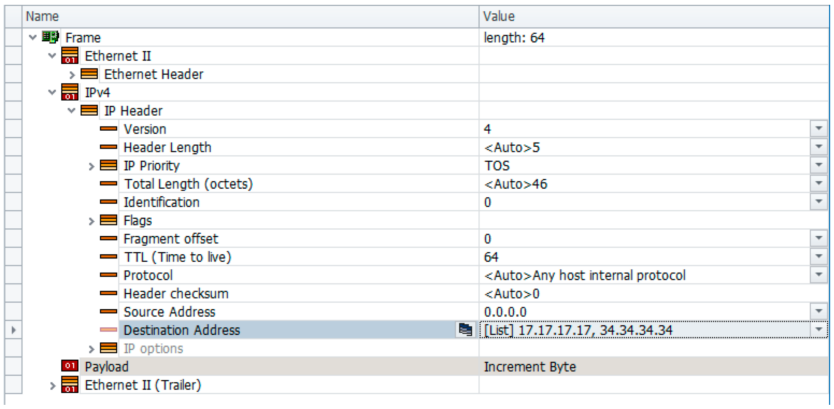


Figure 19. Ixia packet editing page.

We used the Quartus Signal Tap Logic Analyzer to capture real-time signals in FPGA to verify the LPM table. Figure 20 shows the real-time-captured operating waveform of the LPM table. The lookup_data is the parsed destination IP address of packets, using as a key to lookup the LPM table shown in Table 4. The match_lines indicates the matched entry ID for the lookup_data. The *match* indicates whether there is a match for the input lookup_data. The match_addr is the encoded value from match_lines by LPM PE. Since the searching latency is constant at two clock cycles due to the delay caused by BRAM access, the match information appears after two cycles since the lookup_data entered. For example, the bit 1 and bit 2 of match_lines are asserted at t_2, which means the lookup_data at t_0 matches the entry 1 and 2. Finally, the LPM PE selects the winner with longer prefix, which is entry 1. It can obviously draw from the waveform that the LPM table performed a search each clock cycle, which could continuously operate under a clock speed of 200 MHz. The search latency was constant at two clock cycles due to the delay caused by BRAM access. Consequently, the throughput of the LPM table was equal to its operating clock frequency, which was 200 million searches (MS) per second. Since the basic cell of the LPM table was a BRAM-based TCAM with a width of 8 bits, the write latency turned out to be $(2^8 + 2)$ clock cycles. Similarly, the EM table could achieve the same performance as the LPM table because they were all built based on the same CAM architecture. Unlike LPM, the EM table had a shorter write latency of two cycles because the EM table was built with BCAMs.

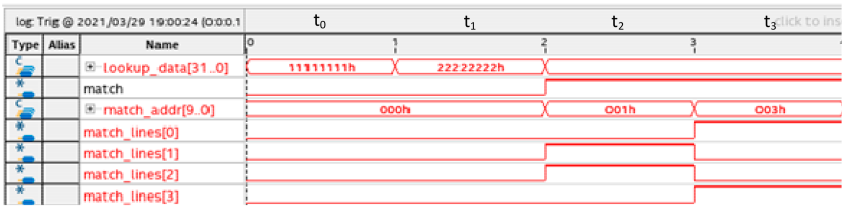


Figure 20. Real-time-captured operating waveform.

In order to evaluate the performance improvement of matching algorithms gained from the hardware implementation, we compared the performances of hardware and software implementations with the match tables of the same size. The software implementation of LPM and EM was borrowed from the DPDK library, whose matching performance has been approved extensively, and the test was performed using only one logic core in the server introduced in Section 6.1 with the Centos operating system. Figure 21 depicts the performance comparison between the hardware and software implementations of different matching methods. The number of rules was selected as 1K. For LPM, the match field was selected as the IPv4 destination address and the IPv6 destination address, and the match field of EM was always the five-tuple. For software implementation, the matching performance decreased as the match field increased, whereas the performance of hardware implementation was not affected by the size of the rule table, which is area-sensitive.

6.3. Action Units

The implemented actions in our design listed in Table 1 can be roughly divided into three categories: arithmetic and logic operations, such as SUB and CAL; field duplication, such as SFD, INS, and DEL; and condition operations, such as DROP and OUTP. They all show a very high performance and area efficiency when implemented on hardware. Most arithmetic logic and duplication-related actions can be completed in one clock cycle. However, these operations are also very efficiently executed on a CPU; hence, the improvement is not obvious. Actions such as packet forwarding are extremely memory-intensive and CPU-intensive when performed with software, yet quite efficient if offloaded onto hardware. We used basic forwarding as a test case to evaluate how much the forwarding performance of the FPGA can improve compared to the CPU. For the accelerator, a basic forward OFU was used to conduct the test, which forwards packets to where they are

received on the FPGA. We used a DPDK example application called basicfwd to test the forwarding performance of the CPU, which also transmits packets as soon as they are received from XL710 NICs. Meanwhile, the single queue and logic core were used for the DPDK test. When testing the forwarding performance of the CPU, the N3000 was configured to function as a normal NIC by programming the FPGA as a simple coupler to connect the external ports and the XL710 ASICs. The test traffic was generated by Ixia, which was traffic with a constant bit rate (CBR) of 100 Gbps and fixed packet sizes from 64 to 1024 bytes. Figure 22 compares the throughput and latency of the DPDK and FPGA for the basic forwarding test, and it is clear from the result that the FPGA has a large advantage over the CPU for fast packet forwarding.

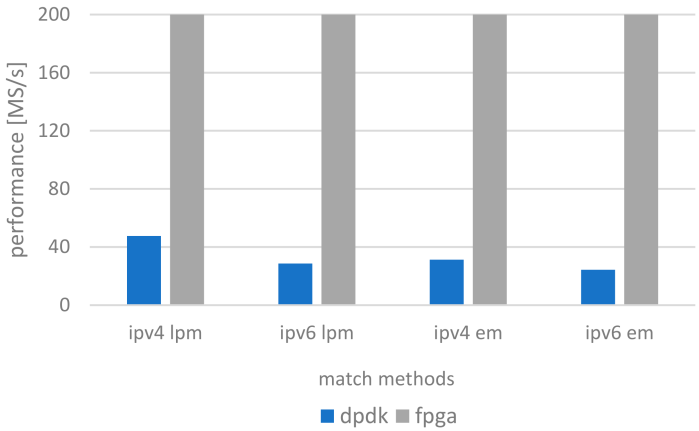


Figure 21. Comparison of the performance of software and hardware implementations of different matching algorithms.

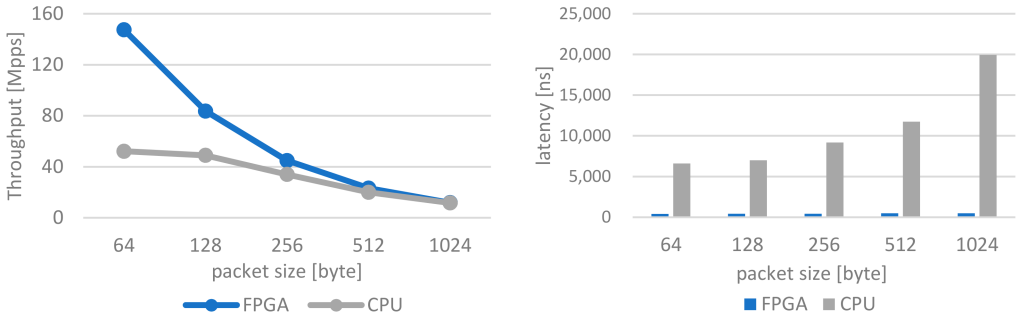


Figure 22. Packet forwarding performance comparison between FPGA and CPU. (**a**) The respectively achieved highest throughputs; (**b**) the generated latency of packet forwarding performed by FPGA and CPU.

6.4. System Performance

In this section, we evaluate the overall performance of the two OFUs. The software implementation used for comparison was a DPDK application that performs the function of fast path of DPDK accelerated Open Virtual Switch (OVS-DPDK) [51]. Briefly, the application parses the received packets and looks up the cached-rule table with corresponding match fields, performing the specified actions for the matching ones. The performed actions included setting up MAC addresses, TTL decrement, inserting VLAN headers, IP header checksum (only for IPv4), and output to specific ports. In addition, the N3000 was configured to function as an NIC when carrying out the above test. We evaluated the system performance of software and hardware implementations of different rule tables, respectively, and the measurement was performed under the assumption that every packet for the test hit the cached rules. Ixia generated the test traffic, which was traffic with a

CBR of 100 Gbps and fixed packet sizes from 66 to 1024 bytes. Figure 23 compares the throughput and latency of different implementations. The result shows that offloading the rule table onto the FSNIC could significantly improve the performance. The achieved throughput reached 100 Gbps with full-sized packets, which is nearly seven times better than that of software when processing small packets, and the latency reduced over 100-fold.

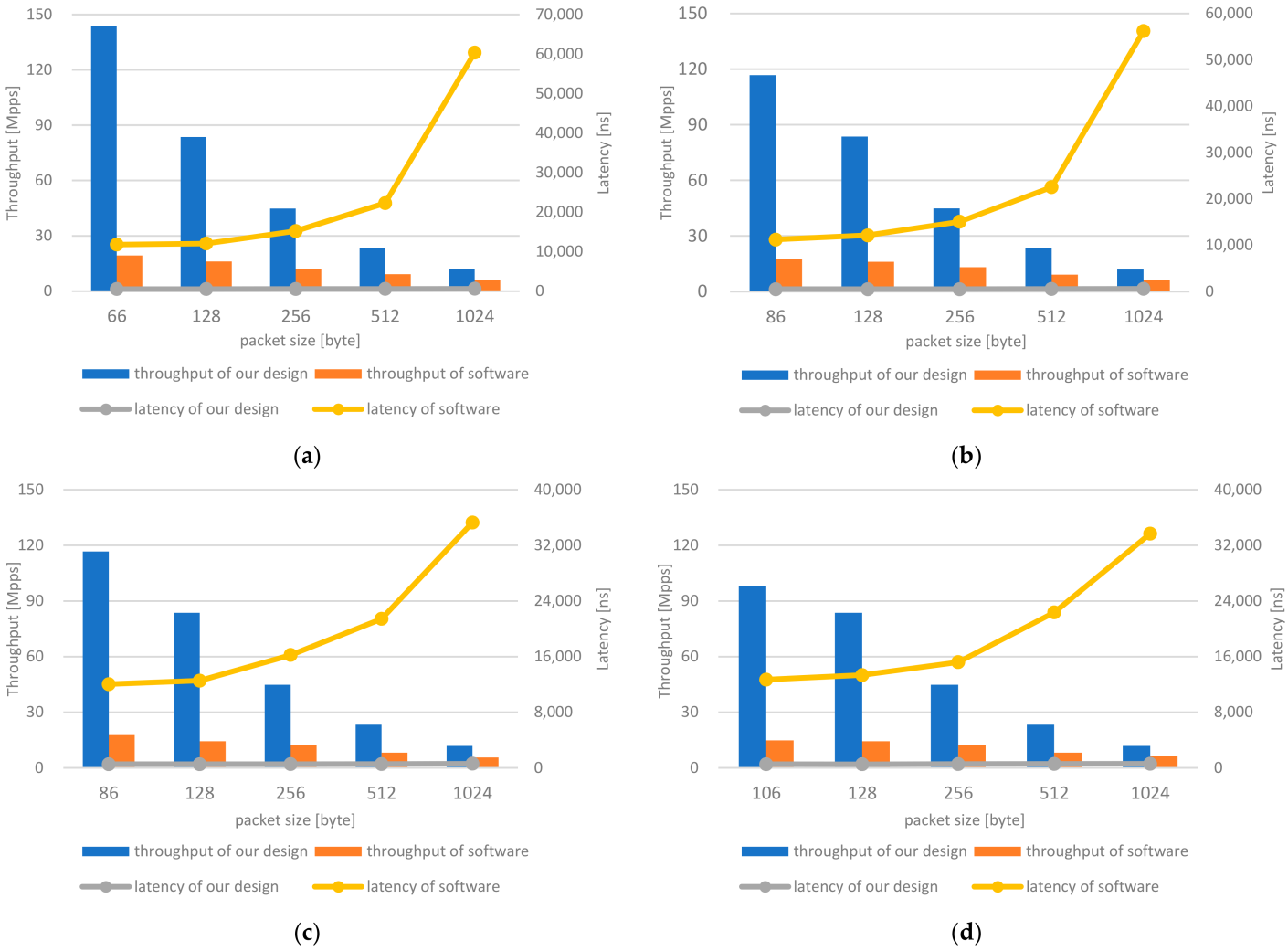


Figure 23. Performance comparison of different functions between our architecture and OVS-DPDK. (**a**) Result of LPM rule table with pure IPv4 traffic; (**b**) result of LPM rule table with pure IPv6 traffic; (**c**) result of EM rule table with pure IPv4 traffic; (**d**) result of EM rule table with pure IPv6 traffic.

6.5. Rule Table Configuration

The configuration efficiency of the rule table is also an important factor that affects the overall system performance due to the rapid change of SDN rules in the actual network. The update of the hardware rule table is performed in two main parts. The host gives instructions to the hardware via the API, and then the hardware parses the instructions and executes them. Consequently, the update efficiency of the hardware rule table is related to the rate of writing MMIO registers and the configuration rate of the hardware MAT, and the configuration rate of the hardware MAT is related to the configuration rate of the CAM-based match tables, including the TCAM-based LPM table and the BCAM-based EM table. Figure 24 depicts the writing rate for the three elements related to the update efficiency, with the ordinate showing the number of writes per second and the abscissa indicating the size of data per write. The results in the figure show that the writing speed of the BCAM is not the bottleneck of the update performance of the EM table, so its performance is limited

by the writing rate of MMIO registers. The instruction lengths of the three operations for the EM table were 456, 328, and 160 bits, and thus, the configuration performance of the EM table is the corresponding access speed of MMIO registers, which is shown in Table 5. For the LPM table, the writing speed of the TCAM limited the update efficiency when the instruction length did not exceed 512 bits, and the speed of accessing MMIO registers could have been a bottleneck when the instruction length exceeded 512 bits. Table 5 shows the limiting performance of the three operations under stress testing. The populate and delete operations are related to the configuration of the LPM table, and the required instruction length is less than 448 bits, so the ultimate performance of these two operations is limited by the writing efficiency of the TCAM. The update operation is only related to the action RAM, so its performance is only restrained by the writing speed of MMIO registers.

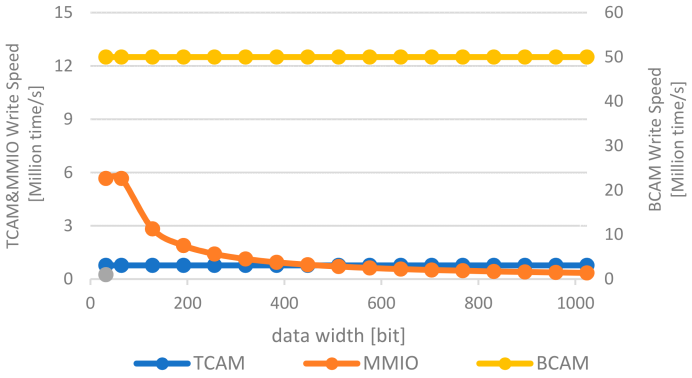


Figure 24. The write speed of TCAM, MMIO, and BCAM.

Table 5. Limiting performance of three operations of rule tables.

Operation Type	Speed (Million Times/s)	
	LPM Table	EM Table
Populate	0.78	0.71
Delete	0.78	0.78
Update	1.89	1.89

7. Related Work

The software implementation of a data plane stands out because of its competitive advantages of agility and flexibility in service creation and deployment; it is, however, subjected to a downgraded performance due to its softwarized architecture. Many works have leveraged hardware devices to improve the performance of the software data plane. Table 6 compares our work with the related works in terms of flexibility, implemented throughput, and so on.

FlexNIC [16] implements a fixed-scale RMT on an NIC so that applications can install packet processing rules into the NIC. The FlexNIC programming model can improve packet processing performance while reducing memory system pressure at fast network speeds. However, as we discussed in Section 2.1, an ASIC-based accelerator cannot cope well with the rapidly evolving SDN, suffering from the shortcomings of a lack of flexibility and adaptability. Our accelerator can not only implement an RMT to achieve the same design goals as FlexNIC but is also provided with the scalability for more acceleration functions due to the reconfigurability offered by the FPGA.

The study [52] introduces a strategy for OVS to detect an elephant flow and offload its processing to an NPU so that the overall performance of OVS can be improved. However, as we discussed in Section 2.1, processor-based accelerators hardly meet the scalability requirements for implementation beyond 100 Gbps due to their packet processing architecture. Nevertheless, the fast path of OVS can be offloaded to our accelerator, and the total throughput can reach 100 Gbps.

Table 6. Comparison to related work.

Design	Hardware Device	Accelerated Function (s)	Flexibility	Implemented Throughput
[16]	ASIC	RMT-Based NFs	Low	20 Gbps
[52]	Network Processing Unit (NPU)	Elephant Flow Detection and Offloading	Low	20 Gbps
[53]	FPGA	VXLAN Process	High	20 Gbps
[54]	FPGA	Data Path for Edge-to-Core Communications	High	40 Gbps
[25]	FSNIC	VM-VM Communications in Data Center	High	40 Gbps
[55]	SoC FPGA	Security Functions	High	8 Gbps
[56]	SoC FPGA	Multiple NFs	High	4 Gbps
Our Work	FSNIC	Multiple NFs	High	100 Gbps

The study [53] leverages an FPGA to accelerate the VXLAN performance of OVS and [54] accelerates the data path for a virtualized and softwarized 5G network with an FPGA. Both of them were implemented and evaluated under an Ethernet speed of 10 Gbps, and they were both designed to accelerate specific functions. These functions can be implemented on our accelerator as multiple OFUs, and the achieved throughput can be 100 Gbps.

AccelNet [25] accelerates VM–VM communications based on an FSNIC. It only focuses on offloading for the EM-based generic flow tables of the virtual switch platform in a public cloud. Our work proposes a more general acceleration architecture for packet processing in a software data plane that is not only able to offload the EM rule table but can also perform many other functions.

FAS [55] uses an SoC FPGA to accelerate and secure SDN software switches. DrawerPipe [56] proposes a reconfigurable pipeline for network processing based on the MSNIC architecture. Both designs were implemented and evaluated with an Ethernet speed of 1 Gbps. As we discussed in Section 2, the SoC-based accelerator failed to support 100 Gbps or higher throughput implementation. Our design is proposed to accelerate network processing in a software data plane at Ethernet speeds of 100 Gbps or higher. Our design was implemented on an FSNIC with 100 Gbps Ethernet, and the tests were run on actual hardware instead of a simulation, with 100 Gbps test traffic generated by the Ixia XGS12 network test platform.

8. Conclusions

We proposed an FSNIC-based reconfigurable accelerator for software-based SDN data plane architecture to break through the packet processing performance bottleneck caused by the CPU while retaining a certain degree of flexibility. The software data plane can offload some of the packet processing tasks to the accelerator as needed so that the overall performance of the data plane can be improved significantly. OFUs can be configured through the software API, and modifications to the OFUs can be made by reprogramming the FPGA logic. Cached-rule table offloading is deployed as a use case in our accelerator, which supports different lookup algorithms and can be configured through software (DPDK application) in the run time. We performed a comparative evaluation of the implementations, and the result showed that offloading the rule table into the FSNIC-based accelerator significantly improves the performance of packet match-action processing, with about a 600% improvement in throughput and over 20× reduction in latency when processing small packets as well as an approximately 100% improvement in throughput and over 100× reduction in latency when processing large packets.

Author Contributions: Conceptualization, X.H. and Z.G.; methodology, X.H. and M.S.; software, X.H. and Y.G.; hardware, X.H., Y.G. and M.S.; validation, X.H., Z.G. and M.S.; writing—original draft preparation, X.H.; writing—review and editing, Z.G., M.S. and Y.G.; supervision, Z.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by Strategic Leadership Project of Chinese Academy of Sciences: SEANET Technology Standardization Research System Development (Project No. XDC02070100).

Acknowledgments: The authors would like to thank the reviewers for their valuable feedback. The authors also would like to thank Ke Wang, Meng Sha, and Xiaoyong Song for their helpful comments.

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

References

1. Pfaff, B.; Pettit, J.; Koponen, T.; Jackson, E.; Zhou, A.; Rajahalme, J.; Gross, J.; Wang, A.; Stringer, J.; Shelar, P. The design and implementation of open vswitch. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, USA, 4–6 May 2015; pp. 117–130.
2. Firestone, D. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA, USA, 27–29 March 2017; pp. 315–328.
3. Honda, M.; Huici, F.; Lettieri, G.; Rizzo, L. mSwitch: A highly-scalable, modular software switch. In Proceedings of the SOSR 2015: The Symposium on SDN Research (SOSR 15), Santa Clara, CA, USA, 17–18 June 2015; pp. 1–13.
4. Molnár, L.; Pongrácz, G.; Enyedi, G.; Kis, Z.L.; Csikor, L.; Juhász, F.; Kőrösi, A.; Rétvári, G. Dataplane specialization for high-performance OpenFlow software switching. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; pp. 539–552.
5. Shahbaz, M.; Choi, S.; Pfaff, B.; Kim, C.; Feamster, N.; McKeown, N.; Rexford, J. Pisces: A programmable, protocol-independent software switch. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; pp. 525–538.
6. Bianchi, G.; Bonola, M.; Capone, A.; Cascone, C. Openstate: Programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 44–51. [CrossRef]
7. Kohler, T.; Durr, F.; Rothermel, K. ZeroSDN: A Highly Flexible and Modular Architecture for Full-Range Distribution of Event-Based Network Control. *IEEE Trans. Netw. Serv. Manag.* **2018**, *15*, 1207–1221. [CrossRef]
8. Pontarelli, S.; Bifulco, R.; Bonola, M.; Cascone, C.; Spaziani, M.; Bruschi, V.; Sanvito, D.; Siracusano, G.; Capone, A.; Honda, M.; et al. Flowblaze: Stateful packet processing in hardware. In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), Boston, MA, USA, 26–28 February 2019; pp. 531–548.
9. Yang, J.; Yang, X.; Zhou, Z.; Wu, X.; Benson, T.; Hu, C. FOCUS: Function offloading from a controller to utilize switch power. In Proceedings of the 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Palo Alto, CA, USA, 7–9 November 2016; pp. 199–205.
10. Barefoot Tofino. Available online: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html (accessed on 13 July 2021).
11. Ibanez, S.; Brebner, G.; McKeown, N.; Zilberman, N. The p4-> netfpga workflow for line-rate packet processing. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 19), Seaside, CA, USA, 24–26 February 2019; pp. 1–9.
12. Wang, H.; Soulé, R.; Dang, H.T.; Lee, K.S.; Shrivastav, V.; Foster, N.; Weatherspoon, H. P4fpga: A rapid prototyping framework for p4. In Proceedings of the SOSR 2017: The Symposium on SDN Research (SOSR 17), Santa Clara, CA, USA, 3–4 April 2017; pp. 122–135.
13. 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. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 87–95. [CrossRef]
14. Go, Y.; Jamshed, M.A.; Moon, Y.; Hwang, C.; Park, K. Apunet: Revitalizing GPU as packet processing accelerator. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA, USA, 27–29 March 2017; pp. 83–96.
15. Kalia, A.; Zhou, D.; Kaminsky, M.; Andersen, D.G. Raising the bar for using GPUs in software packet processing. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, USA, 4–6 May 2015; pp. 409–423.
16. Kaufmann, A.; Peter, S.; Sharma, N.K.; Anderson, T.; Krishnamurthy, A. High performance packet processing with flexnic. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 16), Atlanta, GA, USA, 2–6 April 2016; pp. 67–81.
17. Katta, N.; Alipourfard, O.; Rexford, J.; Walker, D. Cacheflow: Dependency-aware rule-caching for software-defined networks. In Proceedings of the SOSR 2016: The Symposium on SDN Research (SOSR 16), Santa Clara, CA, USA, 14–15 March 2016; pp. 1–12.
18. Li, Y.; Miao, R.; Liu, H.H.; Zhuang, Y.; Feng, F.; Tang, L.; Cao, Z.; Zhang, M.; Kelly, F.; Alizadeh, M.; et al. HPCC: High precision congestion control. In Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM 19), Beijing, China, 19–23 August 2019; pp. 44–58.
19. Huang, M.; Wu, D.; Yu, C.H.; Fang, Z.; Interlandi, M.; Condie, T.; Cong, J. Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale. In Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC 16), Santa Clara, CA, USA, 5–7 October 2016; pp. 456–469.

20. Byma, S.; Steffan, J.G.; Bannazadeh, H.; Garcia, A.L.; Chow, P. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 14), Boston, MA, USA, 11–13 May 20214; pp. 109–116.
21. Fahmy, S.A.; Vipin, K.; Shreejith, S. Virtualized FPGA accelerators for efficient cloud computing. In Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), Vancouver, BC, Canada, 30 November–3 December 2015; pp. 430–435.
22. Putnam, A.; Caulfield, A.M.; Chung, E.S.; Chiou, D.; Constantinides, K.; Demme, J.; Esmaeilzadeh, H.; Fowers, J.; Gopal, G.P.; Gray, J.; et al. A reconfigurable fabric for accelerating large-scale datacenter services. In Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, Minnesota, USA, 14–18 June 2014; pp. 13–24.
23. NVIDIA Mellanox Innova-2 Flex Open Programmable SmartNIC. Available online: https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf (accessed on 13 July 2021).
24. Intel FPGA Programmable Acceleration Card N3000. Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/intel-fpga-programmable-acceleration-card-n3000-for-networking.pdf (accessed on 13 July 2021).
25. Firestone, D.; Putnam, A.; Mundkur, S.; Chiou, D.; Dabagh, A.; Andrewartha, M.; Angepat, H.; Bhanu, V.; Caulfield, A.; Chung, E. Azure accelerated networking: Smartnics in the public cloud. In Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Rention, WA, USA, 9–11 April 2018; pp. 51–66.
26. Eran, H.; Zeno, L.; Tork, M.; Malka, G.; Silberstein, M. NICA: An infrastructure for inline acceleration of network applications. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19), San Diego, CA, USA, 10–12 July 2019; pp. 345–362.
27. Marvell LiquidIO III. Available online: https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf (accessed on 13 July 2021).
28. Netronome Agilio FX. Available online: https://www.netronome.com/media/documents/PB_Agilio-FX-7-20.pdf (accessed on 13 July 2021).
29. Xilinx Alveo U25. Available online: https://www.xilinx.com/publications/product-briefs/alveo-u25-product-brief.pdf (accessed on 13 July 2021).
30. Mellanox Bluefield SmartNIC. Available online: https://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf (accessed on 13 July 2021).
31. Viegas, B.P.; de Castro, A.G.; Lorenzon, A.; Rossi, F.; Luizelli, M.C. The Actual Cost of Programmable SmartNICs: Diving into the Existing Limits. In Proceedings of the 35th International Conference on Advanced Information Networking and Applications (AINA 2021), Toronto, ON, Canada, 12–14 May 2021; pp. 181–194.
32. Gandhare, S.; Karthikeyan, B. Survey on FPGA Architecture and Recent Applications. In Proceedings of the 2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN), Vellore, India, 30–31 March 2019; pp. 1–4.
33. Zilberman, N.; Audzevich, Y.; Covington, G.A.; Moore, A.W. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro.* **2014**, *34*, 32–41. [CrossRef]
34. Forencich, A.; Snoeren, A.C.; Porter, G.; Papen, G. Corundum: An open-source 100-Gbps NIC. In Proceedings of the 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Fayetteville, AR, USA, 3–5 May 2020; pp. 38–46.
35. Michel, O.; Bifulco, R.; Rétvári, G.; Schmid, S. The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications. *ACM Comput. Surv.* **2021**, *54*, 1–36. [CrossRef]
36. Intel FPGA Programmable Acceleration Card N3000 Data Sheet. Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ds/ds-pac-n3000.pdf (accessed on 13 July 2021).
37. Intel Ethernet Controller XL710-BM2. Available online: https://ark.intel.com/content/www/us/en/ark/products/93099/intel-ethernet-controller-xl710-bm2.html (accessed on 13 July 2021).
38. Kaljic, E.; Maric, A.; Njemcevic, P.; Hadzialic, M. A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking. *IEEE Access* **2019**, *7*, 47804–47840. [CrossRef]
39. AXI DMA IP. Available online: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf (accessed on 13 July 2021).
40. Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface. Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl-ias-ccip.pdf (accessed on 13 July 2021).
41. Bosshart, P.; Gibb, G.; Kim, H.-S.; Varghese, G.; McKeown, N.; Izzard, M.; Mujica, F.; Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 99–110. [CrossRef]
42. Avalon Interface Specifications. Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf (accessed on 13 July 2021).
43. Yang, J.; Li, T.; Yan, J.; Li, J.; Li, C.; Wang, B. PipeCache: High Hit Rate Rule-Caching Scheme Based on Multi-Stage Cache Tables. *Electronics* **2020**, *9*, 999. [CrossRef]

44. Wang, Y.; Qi, Z.; Dai, H.; Wu, H.; Lei, K.; Liu, B. Statistical optimal hash-based longest prefix match. In Proceedings of the 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Beijing, China, 18–19 May 2017; pp. 153–164.
45. Ray, S.S.; Singh, S.; Sengupta, C.; Ghosh, S.; Sardar, B. A Fine-grained Integrated IP Lookup Engine for Multigigabit IP Processing. In Proceedings of the 2018 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), Radisson Blu, Indore, India, 16–19 December 2018; pp. 1–6.
46. Abdelhadi, A.M.; Lemieux, G.G.; Shannon, L. Modular block-RAM-based longest-prefix match ternary content-addressable memories. In Proceedings of the 2018 28th International Conference on Field Programmable Logic and Applications (FPL), Dublin, Ireland, 27–30 August 2018; pp. 243–2437.
47. Open Programmable Acceleration Engine. Available online: https://opae.github.io/ (accessed on 13 July 2021).
48. Altera Advanced Synthesis Cookbook. Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/stx_cookbook.pdf (accessed on 13 July 2021).
49. Longest Prefix Match (LPM) Engine. Available online: https://www.axonerve.com/download/img/LPMIPv4.pdf (accessed on 13 July 2021).
50. Ultra-Low-Latency Exact Match Search Engine. Available online: https://www.algo-logic.com/sites/default/files/Algo-Logic_ULL_EMSE_0.pdf (accessed on 13 July 2021).
51. Hu, W.; Chen, J. Implement of OVS Based on DPDK Technology. *J. Network New Media* **2018**, *7*, 47–53.
52. Hamadi, S.; Snaiki, I.; Cherkaoui, O. Fast path acceleration for open vSwitch in overlay networks. In Proceedings of the 2014 Global Information Infrastructure and Networking Symposium (GIIS), Montreal, QC, Canada, 1–7 July 2014; pp. 1–5.
53. Yan, Y.; Wang, H. Open vSwitch Vxlan performance acceleration in cloud computing data center. In Proceedings of the 2016 5th International Conference on Computer Science and Network Technology (ICCSNT), Changchun, China, 10–11 December 2016; pp. 567–571.
54. Ricart-Sanchez, R.; Malagon, P.; Salva-Garcia, P.; Perez, E.C.; Wang, Q.; Calero, J.M.A. Towards an FPGA-Accelerated programmable data path for edge-to-core communications in 5G networks. *J. Netw. Comput. Appl.* **2018**, *124*, 80–93. [CrossRef]
55. Fu, W.; Li, T.; Sun, Z. FAS: Using FPGA to Accelerate and Secure SDN Software Switches. *Secur. Commun. Netw.* **2018**, *2018*, 1–13. [CrossRef]
56. Li, J.; Sun, Z.; Yan, J.; Yang, X.; Jiang, Y.; Quan, W. DrawerPipe: A reconfigurable pipeline for network processing on FPGA-based SmartNIC. *Electronics* **2020**, *9*, 59. [CrossRef]