

Efficient FPGA Architecture for DPDK Flow Rules Processing

Vo Thanh Van, Nguyen Phu Lam, Nguyen Nhat Truong, Nguyen Quang Phuong, Vu Tuan Duc, Le Ngoc Toan

Viettel High Technology Industries Corporation, Viettel Group, Vietnam

{vanvt21, lamnp8, truongnn18, phuongnq5, duevt, toanln3}@viettel.com.vn

Abstract— With the development of the Internet, the number of connected devices and the amount of transferred data increase annually leading to more complex network applications and limitations in data processing speed. The Data Plane Development Kit (DPDK) addresses these challenges by optimizing packet processing for high-speed network applications, including offloading processing tasks to Network Interface Cards (NICs) hardware using the `rte_flow` interface, which allows configuration of NICs offloads with specific flow rules. However, with the increasing complexity of network applications, the number of rules used is also growing rapidly, and more complexity. This presents a challenge for building and developing network systems capable of efficiently handling a large number of rules. This paper presents an architecture for processing the `rte_flow` interface using FPGA-based Smart NICs to leverage the reprogrammable capabilities of hardware accelerators. The architecture uses Ternary Content-Addressable Memory (TCAM) to reduce the key size in the rule tables, significantly increasing the number of rules and rule tables supported by the FPGA. Additionally, a custom DPDK driver (named VTL driver) is developed to manage and implement algorithms aimed at optimizing the number of rules in the hardware. To evaluate our proposed architecture, the Alveo U200 FPGA-based accelerator card is used to implement these features including 8 receive rule tables and 1 transmit rule table per ethernet port. Each table supports up to 2048 wildcard matches and 16384 exact match entries. The system achieves a network throughput of 100 Gbps per port and a rule insert rate of 204 million rules per second. These results significantly enhance hardware acceleration efficiency in networking.

Keywords— NIC, smartNICs, `rte_flow`, FPGA, DPDK, Offload, Wildcard match, Exact match, TCAM

I. INTRODUCTION

The rapid growth of data storage in modern networks necessitates advanced solutions for efficient network management. Smart NICs have emerged as an optimal solution to offload CPU tasks and enhance packet processing performance. Currently, there are two primary solutions for application-level flow control to SmartNICs: `tc_flow` API [1] in the Linux kernel and `rte_flow` API [2] from DPDK.

The `tc_flow` API is well-suited for general-purpose networking tasks in Linux environments, including traffic shaping, control, and management in servers, desktops, and other systems leveraging the Linux kernel. In contrast, the `rte_flow` API is ideal for environments requiring ultra-low

latency and high throughput, such as data centers, telecom networks, and other high-performance network applications. Utilizing `rte_flow` allows efficient offloading to NIC hardware for packet processing, reducing CPU load and optimizing network performance.

Existing studies like Corundum[3] focus on data flow processing on FPGA-based SmartNICs but lack integrated flow rule processing. Notably, Meng Sha et al.'s research [4] and FlexNIC [5] provides comprehensive studies on both data and control planes. However, their self-defined flow rules are not compatible with the `rte_flow` API of DPDK, limiting their integration with diverse network applications.

Thus, developing an architecture for processing flow rules on FPGA-based SmartNICs is essential. This will provide researchers and developers with a reference for developing their SmartNIC applications while adhering to the common `rte_flow` API standards of DPDK. The focus of this paper will be on an architecture capable of processing a large number of flow rules on FPGA with limited resources, optimizing driver performance through matching pattern size optimization, and evaluating its effectiveness when applied to current network products such as OVS, firewalls, and similar applications.

II. HARDWARE DESIGN

A. Hardware Architecture

This section details the FPGA design focused on processing configured flow rules, with additional insights into the packet processing flow for clarity. To simplify the architectural diagram, external communication interfaces such as Ethernet and PCIe have been omitted.

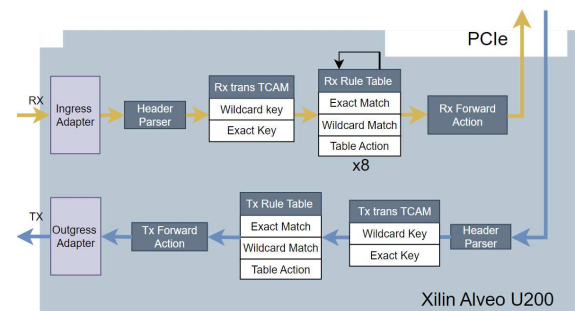


Figure 1. Hardware Diagram

Figure 1 presents the architecture for packet and flow rule processing. As a packet enters the RX path, it first passes through the Ingress Adapter module, where its length is

computed and its validity verified. The resulting metadata, which includes this essential information, accompanies the packet throughout its processing journey. The Header Parser module then extracts the required header fields from the packet, covering all necessary fields from layers 2, 3, and 4 according to the OSI model. This comprehensive extraction ensures compatibility with all matching patterns defined by the `rte_flow` API.

The extracted header fields are then consolidated into a key by the Trans TCAM Rx module. This key is subsequently used to query one or more Rule Tables modules, determining the specific actions to be applied to the packet. The same processing logic is applied to the TX path during packet transmission. Within this architecture, the Trans TCAM module and the Rule Table module are the primary components responsible for flow rule processing on the FPGA. These modules are crucial for demonstrating the flexibility, scalability, and high performance of the architecture.

B. Trans TCAM

To manage the increasing complexity of modern applications, keys are constructed from multiple distinct header fields. The size of these keys, along with the number of rules in the Rule Table, has a significant impact on the FPGA's resource usage, including both logic gates [6] and memory [7] [8] [9] [10]. Reducing key size is crucial because it directly decreases resource consumption, allowing the NIC to support a greater number of wildcard rules, which is essential for accommodating large-scale applications.

This paper proposes a novel method for reducing key size by grouping header fields. The idea is to categorize common header field values into distinct groups, while consolidating less common values into a single group. This grouping process can be customized by users based on the specific needs of each application, enhancing the flexibility of the architecture while also reducing resource demands compared to maintaining each header field in its original state.

For example, consider the EtherType field [11] in the Layer 2 header of an Ethernet packet, which is 16 bits long and identifies the protocol encapsulated in the payload. In network applications like Open vSwitch (OvS), which mainly handles IPv4, IPv6, ARP, and VLAN protocols, only five groups are necessary: one group each for IPv4, IPv6, ARP, VLAN, and one for all other protocols. By grouping in this way, the EtherType field can be reduced from 16 bits to just 3 bits, which is sufficient to represent the five groups.

Figure 2 illustrates the transformation of header fields into keys. Before the fields are combined to create keys for the Rule Table, each field passes through the trans TCAM block, where the grouping process takes place. The trans TCAM module, responsible for translating header fields, takes advantage of TCAM's ability to perform one search operation per cycle, mapping each header field to its corresponding group. This approach not only meets high-performance processing requirements but also optimizes the use of FPGA resources.

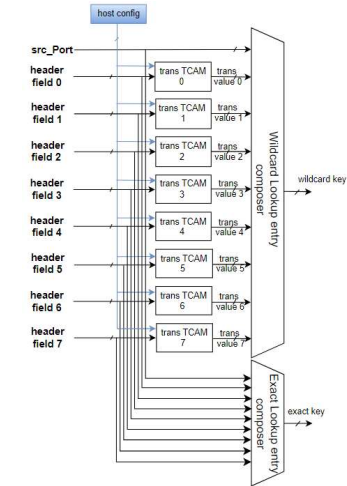


Figure 2. trans TCAM Architecture

C. Key Classification and Rule Table Design

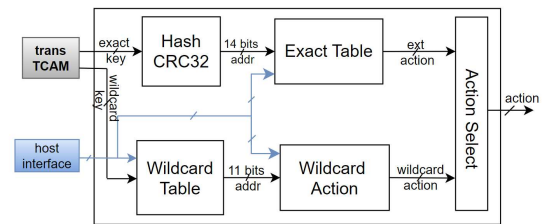


Figure 3. Rule Table Architecture

In the field of network packet processing, the classification of keys into exact keys and wildcard keys plays an important role in optimizing performance and managing hardware resources, especially on FPGA platforms. Exact_key requires a complete match of fields in the packet header, allowing for precise and efficient identification of data streams, thereby improving the accuracy and security of the network system. In contrast, wildcard_key provides the necessary flexibility for complex network systems, allowing a rule to be applied to many different packets without having to specifically define each field. The use of wildcard_key helps to reduce hardware resource consumption and increase the scalability of the system, which is especially useful when handling large network traffic while ensuring high performance. After classifying keys into exact_key and wildcard_key in the Trans TCAM module, the Rule Table module is designed with two separate tables to handle each type of key. The Exact Table uses a hash key to quickly generate a lookup address in the table, optimizing lookup time and managing memory efficiently. Converting the input key into a direct index helps the system reduce resource consumption and improve search performance, especially when storing key-action pairs in FPGA memory. Meanwhile, the Wildcard Table uses TCAM to look up wildcard keys. TCAM allows multiple bits in the key to be compared with table values simultaneously, even if some bits are masked. The look up results return the matching address with the highest priority [12]. Although TCAM requires more resources and may limit the number of rules that the Wildcard Table can handle, it provides the flexibility and robustness needed to handle complex rules in a network

system. The combination of Exact Table and Wildcard Table creates a flexible architecture, capable of accurately and efficiently handling many types of network traffic, meeting the diverse requirements of modern network applications.

The Action Select module is used to select the corresponding action for each packet, depending on the results returned by `ext_action` and `wildcat_action` from the Exact Table and Wildcat Action modules. In case the packet matches both the exact table and the wildcard table, the exact table will take precedence. When there is an action for the packet, the actions will be performed in the next module (Rx/TX Forward Action) in the processing flow.

D. TCAM Implementation on FPGA

Currently, FPGA platforms lack dedicated TCAM resources, leading to the development of TCAM implementations using two main methods: logic-gate-based TCAM [6] and memory-based TCAM [7] [8] [9] [10]. In logic-gate-based TCAM, FPGA logic resources, such as flip-flops, are used to store key-mask pairs and compare them against input keys using logic circuits. While this method simplifies design and allows for high insertion throughput, it significantly consumes FPGA resources as the number of rules increases.

In contrast, memory-based TCAM utilizes memory resources like Block RAM (BRAM) or Lookup Table RAM (LUTRAM [13]) to store keys and masks. Each rule is represented by a one-hot bit in memory, with the input key serving as a pointer to the rule's location in memory (Figure 4). Although memory-based TCAM is more efficient in terms of memory usage, it suffers from lower insertion throughput due to the need to write to all memory addresses. This limitation can be mitigated by applying a "burst" insertion mode, which allows multiple rules to be inserted simultaneously, thereby increasing insertion throughput. Memory-based TCAM is particularly suitable for Wildcard Tables due to its lower resource cost and flexibility in managing complex rules, making it ideal for scalable network systems.

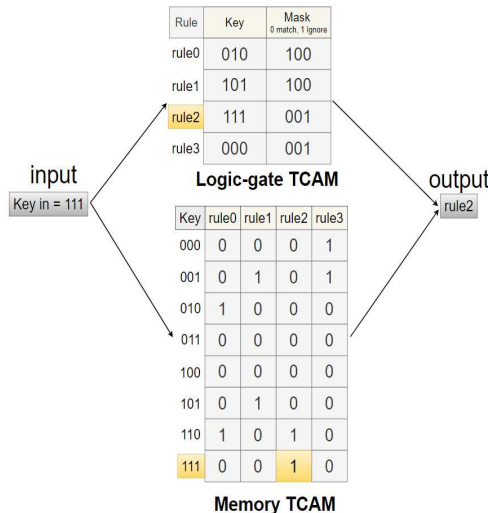


Figure 4. Logic-gate TCAM vs Memory TCAM.

The hardware architecture includes eight Rx rule tables and one Tx rule table per Ethernet port. Each rule table can

accommodate up to 2,048 wildcard entries and 16,384 exact entries. The configuration of eight Rx rule tables and one Tx rule table is selected based on the characteristics of the `rte_flow` format, where most actions target Rx packets (such as drop, RSS, mark, and queue), with only a few actions applied to Tx packets (such as count, checksum, and encapsulation). The choice of 2,048 wildcard rules and 16,384 exact rules is derived from a hyperscale rule value [14] to estimate FPGA resource requirements.

The memory-based TCAM, which stores the wildcard entries, is organized into multiple memory blocks, each sized at 2,048×16 and addressed using 4-bit keys. For instance, an 88-bit key, formed by combining all `trans_values` into a wildcard key, is segmented into 22 addresses, each 4 bits long, corresponding to 22 memory blocks. A rule is considered a match when all 22 memory blocks align, as demonstrated in Figure 5.

Wildcard TCAM Block memory

key	rule0	rule1	...	rule2048
0000	0	0	...	1
0001	0	1	...	1
...	...	...	...	...
1110	1	0	...	0
1111	0	0	...	0

x 22

Figure 5. Wildcard TCAM Block memory

Trans TCAM also employs memory-based TCAM. However, unlike wildcard TCAM, Trans TCAM only performs matching based on the longest prefix. This approach avoids conflicts (i.e., multiple matches) in Trans TCAM, thereby eliminating undefined behavior.

The following section will detail the rule management process in firmware and the algorithms used to optimize rules on FPGA.

III. FIRMWARE DESIGN

Firmware allows the network application to communicate with the hardware of NIC through the API driver.

A. Firmware Architecture

The communication between software, firmware and hardware is illustrated in Figure 6.

DPDK applications use the `rte_flow` API to offload packet processing and classification to hardware. The VTL Driver translates the `rte_flow` API's parameters format into hardware configuration data, transmitting it via the Axi-Stream bus [15]. The Rule Parse block is responsible for analyzing and verifying flow rules, determining their type, and directing them to either the Wildcard Match Rule Handler or the Exact Match Rule Handler for processing. The hardware configuration information is then packaged into a data stream and sent to the NIC card via PCIe interface. The VFIO-PCI Driver is used to bypass the Linux kernel space.

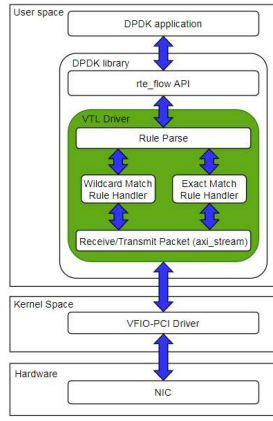


Figure 6. Firmware Architecture

B. Firmware flow

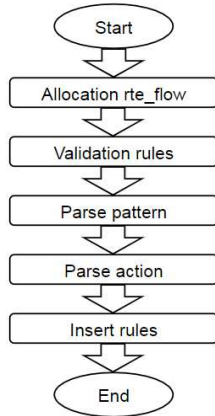


Figure 7. Firmware flow

The basic operation of the VTL Driver is illustrated in Figure 7. Initially, the flow is allocated a memory area, followed by the validation of flow information through the verification of attributes, patterns, and actions. Once the flow information is validated, the pattern and action are parsed and transmitted to the hardware as either an exact match or a wildcard match, based on the rule's attributes.

Wildcard match rule

The application interacts with the NIC hardware through the `rte_flow` API. The TCAM, located in the Programmable Logic (PL) section of the FPGA, is managed internally by the driver and is not visible to the DPDK application. Both the receive (Rx) trans TCAM and transmit (Tx) Trans TCAM are entirely managed by the driver. The driver creates or destroys rules in the trans TCAM based on the rules in the Wildcard Table.

The Figure 8 describes the Ethernet type trans TCAM structure. The `num_rule_depend` value indicates the number of wildcard rules currently using this Ethernet type value as a match field. The Ethernet type trans TCAM in the firmware domain and the NIC domain are synchronized. When an insertion of a wildcard rule into the Wildcard Table is needed, the driver updates the trans TCAM both in firmware and hardware. The `num_rule_depend` value is also adjusted accordingly. Conversely, when a wildcard rule is destroyed, the `num_rule_depend` value is decremented by 1. If the

`num_rule_depend` value reaches 0, the driver removes the corresponding rule from the trans TCAM.

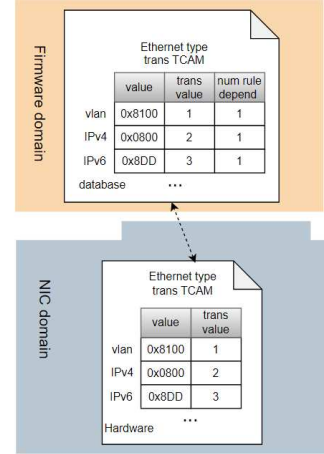


Figure 8. Trans TCAM management

As presented in the section III-C, the TCAM under the NIC uses FPGA memory resources. To manage wildcard rules, the DPDK driver also includes a wildcard TCAM that is mapped 1-to-1 with the TCAM under the FPGA. Thus, corresponding to 8 Rx Rule Tables and 1 Tx Rule Table, the driver maintains 8 Rx Wildcard Tables and 1 Tx Wildcard Table. When an insertion rule is required by the `rte_flow` API, the driver inserts it into the corresponding table. The data in the driver's table is then synchronized with the FPGA via a high-speed data transmission line using the Axi-Stream bus. Regardless of whether one rule or multiple rules are inserted, all the memory blocks in the Wildcard Table under the FPGA are loaded. Wildcard entries can be inserted in burst mode in order to optimize throughput.

For wildcard rules with match fields in the form of (min, max), these rules are translated into individual entries configured in the wildcard rule table on the NIC. This paper presents an algorithm that significantly reduces the number of rule sets in the wildcard rule table without altering the application logic of the rules. The examples to perform algorithms show in Figure 9. The algorithms pseudo code is show below:

Algorithms pseudo code

Input: `header_field` range (*min*, *max*)

Output: TCAM rule in range (*min*, *max*)

Begin

1 `max_bit_idx = 8 * size of (header_filed) - 1`

2 **do**

3 **if** *min* == *max* **then**

4 `set_rule(key = min, mask = 0)`

5 **increase** *min*

6 **else then**

7 **for** *i = max_bit_idx* **downto** 1

8 **if** *min* % 2^i == 0 **then**

9 **if** (*min* + 2^i - 1) <= *max* **then**

10 `set_rule( key = min, mask =  $2^i$  - 1)`

11 **increase** *min* by 2^i

12 **break**

13 **else continue**

```

14      end-if
15      else then
16          if i == 1 then
17              set_rule( key = min, mask = 0)
18              increase min
19          else continue
20          end-if
21      end-for
22  end-if
23  while (min <= max)

```

Figure 9 provides a visual example of the algorithm in action. Consider a flow rule where the ipv4_src field falls within the range 192.168.0.84 to 192.168.0.96. To cover this range, 13 exact match rules would typically be required. However, by applying the algorithm, only three rules are needed, demonstrating its efficiency.

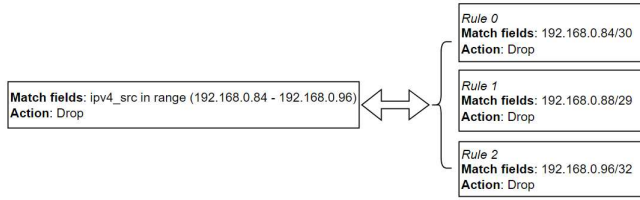


Figure 9. Set range (min, max) example

Here's the breakdown:

- The ipv4_src range converts to hexadecimal as 0xC0A80054 to 0xC0A80060.
- The binary equivalent of the range 0x54 to 0x60 is 01010100 to 01100000.

The algorithm's execution proceeds as follows:

- *First iteration:* The minimum value is 01010100 (0x54), satisfying the condition in step 8 of the pseudocode. Therefore, a rule 010101xx is set, with $i = 2$ (represented as Rule 0 in Figure 9).
- *Second iteration:* The minimum value now is 01011000, again meeting the condition in step 8. Thus, rule 01011xxx is set, with $i = 3$ (Rule 1 in Figure 9).
- *Third iteration:* The minimum value is 01100000, satisfying the condition in step 3. Consequently, an exact match rule 01100000 is set (Rule 2 in Figure 9).

In summary, the ipv4_src range is effectively covered using only two wildcard rules and one exact match rule, significantly reducing the number of rules required compared to the original 13 exact match rules.

Exact match rule

Managing the Exact Table is simpler than managing the Wildcard Table. The Exact Table functions as a cache to increase bandwidth for packets requiring Quality of service (QoS). All keys and actions are stored in memory under the FPGA at specified addresses. The DPDK application handles conflicts, selects rules to discard, and adds entries to the Exact Table based on application requirements.

IV. RESULT

A. FPGA resource

Figure 10 illustrates that the majority of FPGA resources utilized in the design are BRAM. However, BRAM can be substituted with URAM or LUT RAM.

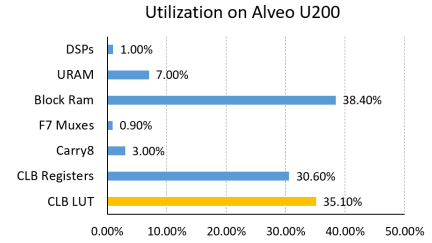


Figure 10. FPGA resources

With the hardware architecture outlined in Section III-A and the rule count used for resource evaluation in Section III-D, a comparison with the study by Meng Sha et al. [4], which utilizes TCAM based on Distributed RAM and custom rule processing, reveals that while both approaches achieve the same 100 Gbps throughput, our approach demonstrates significantly better resource efficiency. Specifically, our implementation consumes only 38.4% of the BRAM resources and 35.1% of the LUT on the AMD Alveo U200 board while supporting up to 165,888 rules (across 9 Rule Tables, each comprising 2048 wildcard rules and 16,384 exact rules).

B. Experiment scheme

To demonstrate the effectiveness of our design, we employed third-party applications to ensure transparency and reliability. Specifically, we utilized DPDK's l3fwd application for the application layer and implemented the rule table structure as outlined in the Section 5 Case Study of Meng Sha et al. [4]. The rule processing on the FPGA and driver adhered to the methodology proposed in this study. We modified the l3fwd application to support the configuration of rte_flow rules via command line, accommodating both single and burst modes for wildcard and exact matches. The match-action table, as described in [4], consists of five tables, each configured to check a specific packet field (e.g., source MAC, Ethernet type) and execute corresponding actions (e.g., drop, remove VLAN header, calculate checksum). Furthermore, we incorporated a substantial number of rule test cases with diverse rule ranges to assess the insertion rate.

Figure 11 depicts the test system topology:

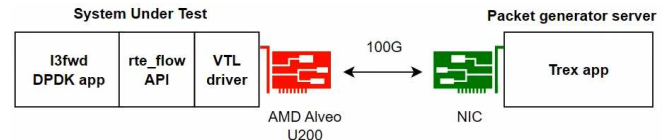


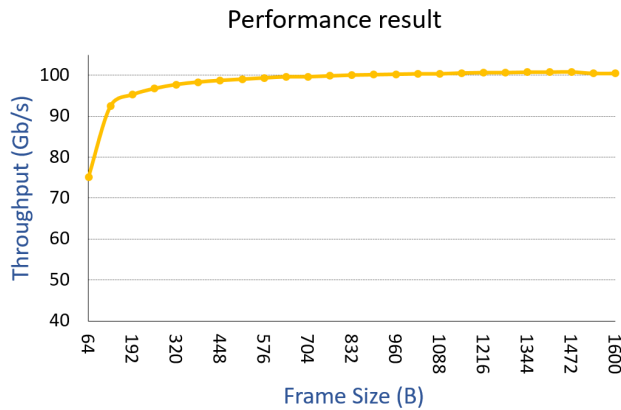
Figure 11. Test topology

The System Under Test (SUT) is made up of the Supermicro server and the AMD Alveo U200 NIC card. The SUT is connected to the packet generator server which generates traffic towards the Alveo U200. The Alveo U200 data traffic is passed through DPDK to the DPDK test application l3fwd and is redirected to the opposite direction on

the same port. Packet generator server measures throughput and packet loss.

C. Result

To evaluate the performance of the proposed architecture, we conducted tests to measure the following parameters: capacity, throughput, and insert rate.





Vo Thanh Van received the Engineer degree in electronics and telecommunication from Da Nang University of Technology (DUT), Vietnam, in 2014, and the M.S. degrees from the Electronic and Telecommunication Department, DUT, Vietnam in 2017. He is currently a FPGA Engineer at Viettel High Technology Industries Corporation, Viettel Group, Vietnam. His current research interests include hardware accelerators, high speed communication, high frequency signal processing.



Nguyen Phu Lam received the Engineer degree in electronics and telecommunication from Da Nang University of Technology (DUT), Vietnam, in 2014. He is currently a FPGA Engineer at Viettel High Technology Industries Corporation, Viettel Group, Vietnam. His current research interests include hardware accelerators, SDN.



Nguyen Nhat Truong received the Engineer degree in electronics and telecommunication from Da Nang University of Technology (DUT), Vietnam, in 2014. He is currently a Firmware Engineer at Viettel High Technology Industries Corporation, Viettel Group, Vietnam. His current research interests include hardware accelerators, network monitoring.



Nguyen Quang Phuong received the Engineer degree in embedded system from Da Nang University of Technology (DUT), Vietnam, in 2023. He is currently a FPGA Engineer at Viettel High Technology Industries Corporation, Viettel Group, Vietnam. His current research interests include hardware accelerators.



Vu Tuan Duc received the Engineer degree in information technology from HaNoi University of Science and Technology, Vietnam, in 2004. He currently serves as the Director of Technology Engineering Center at at Viettel High Technology Industries Corporation, Viettel Group, Vietnam.



Le Ngoc Toan received the Engineer degree in electronics and telecommunication from HaNoi University of Science and Technology, Vietnam, in 2009. He is currently the head of DPI Department at Viettel High Technology Industries Corporation, Viettel Group, Vietnam. His current research interests include fast-path processing, DPDK, networking.