symmetry

MDPI

Article

A Fast Approach for Generating Efficient Parsers on FPGAs

Zhuang Cao [1], Huiguo Zhang [2], Junnan Li [1], Mei Wen [1,*] and Chunyuan Zhang [1]

[1] College of Computer, National University of Defense Technology, Changsha 410073, China; caozhuang16@nudt.edu.cn (Z.C.); lijunnan@nudt.edu.cn (J.L.); cyzhang@nudt.edu.cn (C.Z.)
[2] School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, Singapore; Huiguo001@e.ntu.edu.sg
* Correspondence: meiwen@nudt.edu.cn; Tel.: +86-18692868268

Received: 9 September 2019; Accepted: 3 October 2019; Published: 10 October 2019

Abstract: The development of modern networking requires that high-performance network processors be designed quickly and efficiently to support new protocols. As a very important part of the processor, the parser parses the headers of the packets—this is the precondition for further processing and finally forwarding these packets. This paper presents a framework designed to transform P4 programs to VHDL and to generate parsers on Field Programmable Gate Arrays (FPGAs). The framework includes a pipeline-based hardware architecture and a back-end compiler. The hardware architecture comprises many components with varying functionality, each of which has its own optimized VHDL template. By using the output of a standard frontend P4 compiler, our proposed compiler extracts the parameters and relationships from within the used components, which can then be mapped to corresponding templates by configuring, optimizing, and instantiating them. Finally, these templates are connected to output VHDL code. When a prototype of this framework is implemented and evaluated, the results demonstrate that the throughputs of the generated parsers achieve nearly 320 Gbps at a clock rate of around 300 MHz. Compared with state-of-the-art solutions, our proposed parsers achieve an average of twice the throughput when similar amounts of resources are being used.

Keywords: packet parser; pipeline; P4; FPGA

1. Introduction

A network packet consists of a packet header and a payload, which are also known as the control information and user data, respectively. Protocol headers belonging to the different layers in the seven-layer Open System Interconnection (OSI) model are combined to form the packet header, which is used to decide how to deliver the payload. A parser parses a network packet by locating different protocol headers, extracting the required fields, and locating the payload. The results are used to forward, classify, and/or inspect the packets. The challenge here is that the packet headers are combinations of different protocol headers of various lengths, and it is not possible to determine the elements and their relationships in the header until all protocol headers have been parsed. A parser is a key function implemented in various network processors of different hardware platforms, including Application Specific Integrated Circuits (ASICs) [1,2], Network Processors (NPs) [3], Configurable Network Processors (CNPs) [4], and Central Processing Units (CPUs) [5,6]—each of these have their own advantages and disadvantages. For example, ASICs cannot be upgraded to support any new feature, meaning that a new one must be designed, which costs a lot in terms of time and money. CPUs are flexible but achieve low performance, while NPs achieve average performance and are flexible. Moreover, CNPs are high-performance and flexible, but are still limited by their

architecture—consider for example their limited capability to execute "external" functions, which are those that can execute more operations on the packets in addition to forwarding and are also important in router functions, such as packet inspection, decoding, etc.

Software-defined Networks (SDNs) [7–10] have been widely deployed on E-commerce platforms, in data centers, and for Internet Service Providers (ISPs), which are driven by the need for flexible high performance and online deployment. SDNs are useful in cases where traditional network processors cannot meet the new requirements. For example, traditional networks have a lack of flexibility, and they cannot allocate bandwidth based on real-time traffic or demands. On the other hand, thousands of protocols are used by their own manufacturers, so it is really hard to maintain and upgrade them. If SDN is introduced into the network, we can collect the traffic demand between data centers by deploying a unified controller, so as to perform unified calculation and scheduling, implement flexible on-demand allocation of bandwidth, conduct network optimization, and improve resource utilization.

Advances have been witnessed in Domain Specific Languages (DSLs) [11,12] and corresponding compilers [13,14] with various hardware platforms, including ASICs and FPGAs. For example, an ASIC chip named "Tofino", from Barefoot Networks, is a P4-programmable switch chip that runs up to 6.5 Tbit/s [4]. By using such technology, new IT demands can be applied in the existing devices, without replacing them. To develop network processors on FPGAs, Xilinx provides a toolkit, namely, SD-Net, which supports using high-level languages, including P4, and their purpose is to target the FPGA-based network processor market in the future. This tool has been widely used in universities and institutes for trial purposes. Many other solutions combine DSLs and FPGAs to develop the network parsers—some of these are introduced in Section 2.3. Investigating them reveals many disadvantages, as follows:

- Heavy resource cost and long pipeline stages are used, such as taking more than 250 nanoseconds to process one packet and using more than 10% Flip Flops (FFs) and Look Up Tables (LUTs) in the FPGAs.
- All the packets, including the payload, go through the parser, which wastes a lot of cycles to transport the payloads and reduces the packet parsing performance.
- Parsers are not thoroughly pipelined, and multiple cycles are used to process a packet in one pipeline stage, which stalls the pipeline and reduces the performance.

Accordingly, this paper proposes a framework comprising a pipeline-based hardware architecture and an approach that converts P4 programs to VHDL. This hardware architecture can be divided into many components, which are abstracted and implemented via Real Time Logic (RTL) templates with VHDL. These templates have been prewritten by skilled FPGA programmers, and are well-designed and thoroughly tested. A back-end compiler is also presented that parses the P4 programs by extracting the values of the parameters and mapping them to related templates of the components in the hardware architecture. Pipeline stages are scheduled to avoid conflicts and stalling during the compilation, while VHDL templates are instantiated and connected to form the expected parsers. Compared with existing architectures [15,16], the proposed framework presents an efficient method to extract the packet header from any supported packets. Thus, the processing speed of one packet per one cycle is sustained even on the longest frame length. In addition, the payload of each packet no longer goes through the parser, which increases the parsing efficiency.

The main contributions of this paper are as follows:

- We design a hardware architecture in a pipeline fashion for the parser. Templates for the components in the hardware architecture are created and thoroughly tested.
- We employ an approach to map the P4 programs to the proposed hardware architecture, in which the network programmers of nonhardware knowledge focus on the high-level software.
- We demonstrate a fast compiler to implement the P4 program to FPGA targets.

In addition, this paper presents two examples that are implemented based on the proposed framework. The results show that the packet rate achieved is nearly 300 Mbps, while the throughput

achieved is nearly 320 Gbps. Furthermore, the resource usage is less than 5% of the FPGA, and in the situation where similar resources are used, our proposed parsers achieve an average of twice the throughput compared to the state-of-the-art parsers.

2. Background and Related Work

Designing a network parser using traditional methods on a hardware platform is a difficult and costly process. For example, low-level languages are used, which requires the time-consuming work of describing hundreds of millions of gates—not even the smallest mistake is allowed, since the hardware is not reconfigured, so thorough tests will need to be repeated if even a small change occurs. Combining the P4 language and FPGAs may resolve this problem. Accordingly, this section briefly introduces FPGAs, the P4 language, and some related solutions.

2.1. Introduction to FPGAs

An FPGA is a type of integrated circuit that can be programmed at the hardware level after manufacture, and consists of FFs, LUTs, block RAM, a huge number of wires, etc. Such primitives are "pre-laid out" (already laid out) on the chip—their contents, as well as the connections among them, can be reconfigured based on requirements. Verilog or VHDL are types of language used to describe such requirements.

Due to their being programmable at the hardware level, FPGAs are suitable for parallel computing with high throughput, and are thus extensively used in big data processing, Artificial Intelligence (AI) computing, the network domain, etc. However, there are also many disadvantages to the use of FPGAs. For example, the fact that they are "pre-laid out" causes complex programs to require long critical paths, and they cannot be executed to high clock rate. Resource-consuming programs cannot even be implemented on FPGAs due to resource limitations.

2.2. P4 Language

P4 is a kind of DSL that describes the process for handling the packet in the data plane that can target a wide range of technologies including CPUs, FPGAs, and NPs. It aims to fundamentally change the way traditional network systems are designed. Two versions of P4 have been released so far. Limitations were quickly identified after the first version called $P4_{14}$ was released in 2015, so the new version called $P4_{16}$ with new features was released in 2017, and each of them is supported by the official compiler. The P4 program consists of the following five components:

- Protocol Headers: Each of these structures defines the header of a specific protocol, and includes the fields' names and lengths, as well as the order among all fields.
- Parser: This is a state machine that describes the transition among all supported protocol headers, which indicates the method of parsing one protocol header after another. For example, an IPv4 header can be confirmed and located by extracting the next type from the Ethernet header.
- Table: The first part of the "Match-action" implementation. It stores the mechanism for performing packet processing and defines how the extracted header fields are used for matching, including exact match, longest prefix match, or ternary match. For example, a destination IP address is used to find out which port the packet should be sent to.
- Action: The other part of "Match-action" implementation is built based on a series of predefined simple basic operations that are independent of the protocols. Complex customized actions are built with them and can be used in this part as well.
- Control flow: It is a simple imperative program that describes the flow of control, and it controls the process order within the "Match-action" architecture.

The "Protocol Header" and "Parser" components indicate how to parse the pending supported protocols. The "Table" and "Action" components explicitly or implicitly define the fields that need to

be extracted from the packet header. The P4 programs are compiled using the official P4 compiler [13], which outputs an intermediate file in the extensible mark-up language. This file can be easily parsed by various back-end compilers that find their own solution, such as generating executable instructions, or generating other high-level languages for different hardware platforms.

2.3. Packet Parser Solutions

Attig and Brebner proposed a high-performance pipeline-based online reconfigurable parser for FPGAs by using a simple high-level language in [17]. This parser has very high performance, as the throughput achieved is 400 Gbps. However, this parser runs in extreme conditions, such as having very long pipeline stages and taking up more than 10% of LUTs and FFs, which reduces the resource magnitude of other components of the network processors, such as reducing tables' size. In addition, long latencies cause a long processing time in the chip for the incoming packets.

Gibb et al. presented a methodology to design a fixed parser and reconfigurable parser for ASICs in [18]—these are two typical types of parser widely used in various network processors. The reconfigurable parser is a kind of Finite State Machine (FSM), it starts at the parsing of the first protocol header, and continues parsing other protocol headers based on the state transition diagram. It is designed based on Reconfigurable Match Tables (RMT) methodology [19], and relies heavily on the Content Addressable Memory (CAM), which costs a lot of memory resources on FPGAs and runs at a low clock rate. In addition, the parsing process cannot be pipelined, a packet needs to wait until its previous one is thoroughly parsed.

Benacek et al. presented an approach to converting P4 to VHDL by using the Parser Graph Representation (PGR) and their HFE M2 architecture in [15,20] and Jeferson et al. presented an approach of using templated C++ classes, which can be used to generate RTL code using Xilinx Vivado HLS following a series of graph transformation rounds. These two packet parsers are both organized in a pipeline fashion but of different hardware architectures, and the parsers based on their proposed hardware can process a fairly complex set of headers, as well as achieve 100 Gbps data rate. However, the comparison results show that the generated parser of [15] has roughly 100% overhead in terms of latency and consumes more resources than the handwritten VHDL implementation. Compared to [15], the comparison results of [16] show that it saves a lot of resources.

A new parser architecture is presented by Jakub et al. in [21], which is capable to currently scale up to a terabit throughput in a Xilinx UltraScale+ FPGA, and the overall processing speed is sustained even on the shortest frame lengths. Its main method is building a protocol process module pool, and multiple packets are processed in parallel in this pool. In addition, it combines multiple packets in one data frame once a packet cannot fill the data bus, which increases the bus utilization, so as to increase the throughput. This special architecture of multiple pipelines provides a good reference for our future work.

P4-SDNet is a high-level design tool from Xilinx, and it supports $P4_{16}$ to build new packet-processing data planes that target FPGA hardware. This tool has been continuously updated, and its license is available for academic people by the Xilinx University Program. To implement a P4 specification, the P4-SDNet compiler maps the control flow onto a custom data plane architecture of SDNet engines, and one of them is the parser. [22] is such an example. However, SDnet currently does not support variable-sized headers, which limits its scope of use.

The remainder of this paper is organized as follows. Section 3 presents the high-level hardware architecture of the parser. Section 4 describes the circuit details of the header processing module in the pipeline stage. Section 5 presents the working flow of the compiler, which converts the P4 programs to VHDL by mapping the parameter values in the P4 programs to their corresponding hardware modules. In Section 6, the proposed solution is evaluated by comparing the resource usage, clock rate, throughput, and latency with state-of-the-art solutions. Finally, future work and conclusions are discussed in Section 7.

3. Hardware Architecture of the Parser

3.1. Microarchitecture

The proposed parser consists of many pipeline stages. Each pipeline stage includes one or more header processing modules, and each of these modules is configured to handle a specific protocol header. The packet headers are input to the parser, and their inside protocol headers are processed by the corresponding processing module in the appropriate order. During processing, specified fields are extracted and updated to form the output, referred to as the header vector. In addition, status bytes such as error code and protocol types are also output to the downstream component. Figure 1 presents the abstracted microarchitecture of the parser.

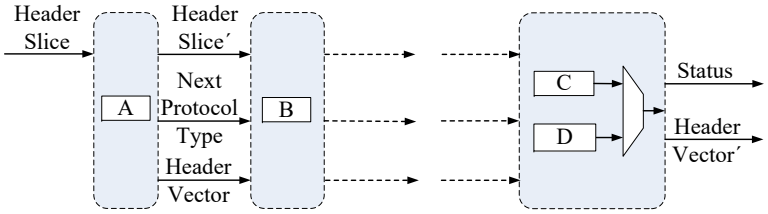

Figure 1. Microarchitecture of the parser. The dotted-line frames represent the pipeline stages; while A, B, C, and D represent different header processing modules.

The packet headers are input to all processing modules in the same pipeline stage simultaneously. The processing module decides how to handle the incoming data by comparing the header types. If the incoming header matches the processing module type, it will be processed during this cycle; otherwise, the header will be sent downstream without any changes being made. For pipeline stages with multiple processing modules, the extra circuit chooses a module for outputting the result of this stage; the last pipeline stage shows this situation.

In addition, errors such as protocol header validations, unrecognized protocols, etc., and customized information extracted during the parsing are also handled by the parser. Such information is sent out of the parser while maintaining alignment with the header vector.

3.2. Main Input and Output

To ensure that the proposed parser processes one packet header per cycle, we send the packet header slice into the parser, which is cut out from the packet and will include all bits of the packet header. In addition, all specified fields are stored in the header vector and output from the last pipeline stage in one cycle. These two ports are defined based on the application.

3.2.1. Header Slice

Since the packet header is a combination of multiple protocol headers of different lengths, the width of the header slice is determined by the supported protocols and will be equal to the longest packet header supported by the parser, regardless of the length of the incoming packet.

For illustration purposes, we assume that a designed parser only supports the protocols "Ethernet", "IPv4", "IPv4 with extensions", and "IPv6". There are three combinations of these four protocols, and the longest packet header is the combination of "Ethernet" + "IPv4 with extensions", the length of which is 592 bits. Thus, the width of the header slice should be 592 bits. For those combinations that have lengths less than 592, the remaining bits will be padded with '0'.

3.2.2. Header Vector

The vector width is determined by the longest total length of the extracted fields in all packets. These extracted fields have fixed positions in the vector during designing. For illustration purposes, we assume that a 5-tuple is extracted for the output. Figure 2 presents the header vector arrangement.

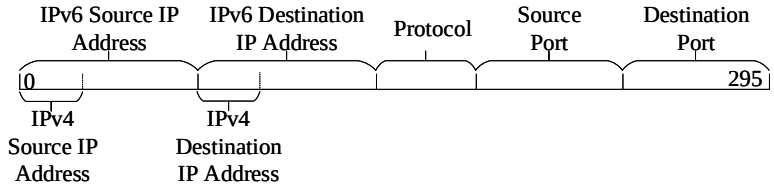

Figure 2. Header vector arrangement.

As the 5-tuple includes the IPv6 and IPv4 addresses, which are of the same type but have different lengths, the longer addresses should be used to calculate the width of the vector. Thus, the length of the header vector should be equal to the total lengths of IPv6 IP addresses, protocols, and ports—i.e., 296 bits. In addition, IPv4 addresses share the same space of the same type and remain aligned with the same starting position of the IPv6 addresses, as shown in Figure 2.

3.3. Processing Module Scheduling

Due to the hierarchical relationship between the protocols, a protocol header in the packet header remains unknown until its previous protocol has been parsed. Based on the hierarchical relationship described in P4 programs, directed acyclic graphs (DAGs)—i.e., parse graphs—can be established. Figure 3a presents a typical parse graph, where the edges represent the protocol transitions, and each node represents a pipeline stage to process a protocol header.

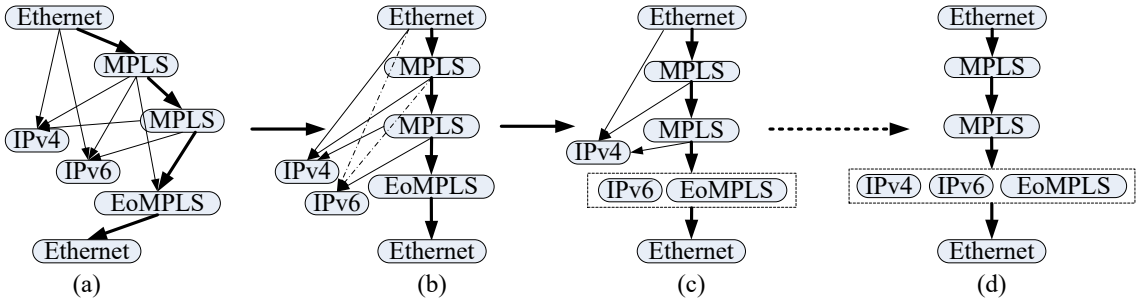

Figure 3. Pipeline Schedule. (**a**) The initially parsed graph from P4 program, and the nodes with thick arrow contracts the longest path in the graph; (**b**) shows when the "IPv6" are selected and two dependencies are deleted; (**c**) shows merging "IPv6" and "EoMPLS" to one node; and (**d**) shows the final result after scheduling.

In Figure 3a, the parser starts parsing the packet by "Ethernet", and three protocol headers may follow it, they are "IPv4", "MPLS", and "IPv6", this is decided by the indicating field in the "Ethernet" header. Similarly, other protocol headers would have many different successors, so there are many various paths from the start header to the end header in the parse graph. The goal of our approach lies in that instead of using duplicate protocol header process modules to prevent conflicts and stalls, we devise a unique pipeline for packet header processing for the same purpose. Specifically, we implement designated modules to process protocol headers in serial. Thanks to this architecture, our unified parser pipeline saves FPGA resources and increases the throughput, which are the main concerns of our design.

3.3.1. Potential Conflicts

Implementing a parse graph on an FPGA, as shown in Figure 3a, there are many branches when the packet is processed from one node to another, these branches bring about conflicts and stalling. For example, if three different packets are inputted sequentially to the parser, conflicts and halts will occur during parsing, as shown in Table 1.

Table 1. Conflict and Halt in Header Parsing.

Packet Header	Clock Number			
	1	**2**	**3**	**4**
Ethernet→MPLS→IPv4	Ethernet	MPLS	IPv4	
Ethernet→IPv4		Ethernet	Stall	IPv4
Ethernet→			Stall	Ethernet

From Table 1, it can be seen that when the first packet is inputted, its protocol "Ethernet" is processed by the first pipeline stage in the first cycle. The protocol "MPLS" is processed by the second pipeline stage in the second cycle, while the protocol of the second packet "Ethernet" is processed by the first pipeline stage. However, in the third cycle, two "IPv4" protocols from two packets need to be processed in the same pipeline stage. This conflict forces the second packet to halt—subsequently, the third packet cannot be accepted due to the halting of the second packet. Pipeline scheduling could be used to solve this problem.

3.3.2. Pipeline Scheduling

Pipeline scheduling is conducted to eliminate conflicts and stalling and to ensure that the parser processes one packet per cycle. For illustrative purposes, the parser graph shown in Figure 3a is chosen to demonstrate the scheduling, while Figure 3b–d show steps of the process flow.

- Find one of the longest paths in the parse graph. There is only one longest path among these five nodes, namely, "Ethernet → MPLS → MPLS → EoMPLS → Ethernet", and it is marked by the thick arrow in Figure 3a.
- Choose any one of the rest of the nontrunk nodes and identify all of its parents in the trunk. Then, reserve the dependency relationship with the last node in the trunk and delete the other dependencies. For example, if "IPv6" is selected, the reserved dependency should be that between "IPv6" and the second "MPLS" in the trunk, as shown in Figure 3b. The dependencies marked by dotted arrows will be deleted. This means that the packet cannot be directly transferred from the "Ethernet" stage and the first "MPLS" stage to the "IPv6" stage.
- Merge this chosen node with its brother in the trunk to form a new node, and further update the dependencies of its children. This process is shown in Figure 3c. After doing this, the packet can only flow in the trunk. We assume that there is a packet header which has only two protocol headers "Ethernet → IPv6" inputted, "Ethernet" will be processed at the first cycle; but the "IPv6" header will be processed in the fourth cycle.
- Go through the rest of non-trunk nodes one by one and add them to the trunk by repeating steps 2 and 3. The schedule is deemed completed when all nodes are in the trunk. Figure 3d shows the final result.

After scheduling, each input packet header passes through a pipeline stage by one cycle. Since the parser has a fixed number of pipelines and all packet headers go through them, the input packets are parsed within constant latency, which is decided by the pipeline stage number.

4. Hardware Architecture of the Processing Module

A common hardware architecture is configured to the different processing modules with various parameter values, each of which handles its corresponding specific protocol headers. This section introduces the common hardware architecture in detail.

4.1. Microarchitecture

There are four main components in the process module: Type Identification, Header Shifter, Field Extractor, and Next Type Generator. Ports include input/output header, types, header vectors, control signals for transporting data streams, etc. Figure 4 presents the microarchitecture.

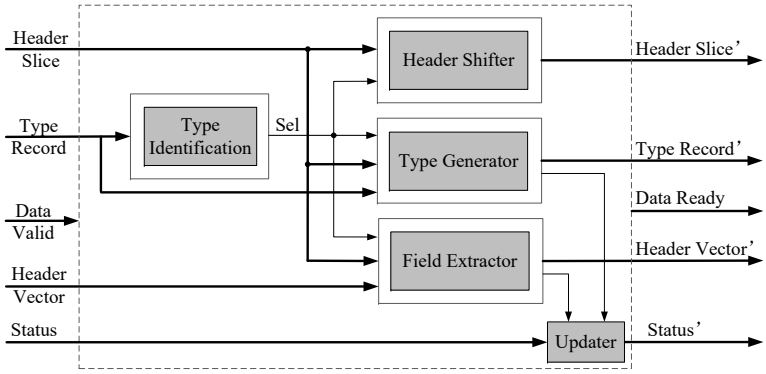

Figure 4. Microarchitecture of the process module.

All input and output data are kept aligned by means of the "Data Valid" and "Data Ready" signals, and these signals enable whether this module works as well. Header slice and header vector are transferred by their own ports. The "Type Record" is a vector that indicates what protocol headers in this packet were parsed and what is the next pending protocol type, it can be checked by the process module. The "Status" is also a vector that stores the error code and customized information, which is also updated by this module.

When the input data are valid, the protocol type is firstly compared by the "Type Identification"—the result of this process controls how the other three components will handle the input data. Examples include extracting specific fields (done by the "Field Extractor"), shifting the incoming header slice of specific bits (by the "Header Shifter"), and generating the next protocol type (by the "Type Generator"), as well as the error detection, etc. performed by other circuits.

4.2. Function of Type Identification

The "Type Record" is a register vector that stores the code of all supported protocols—parsing the previous protocol header always marks its next header. The function of this component is to check whether its corresponding code has been marked by its previous pipeline stage, which means that the rest of the components in this module need to handle the current header in this cycle. Otherwise, processing is not required.

This component is a simple checker that is implemented by pure combinational logic. Code width and processing type configuration are required for the different processing modules.

4.3. Function of Header Shifter

Packet headers are combinations of different protocol headers with various lengths. It is difficult to determine the offsets of each protocol header before the previous one is parsed. In other words, the offsets of the same protocol header may be variable in different packets. A circuit that locates a random position in a vector may be complex, this not only uses a lot of resources, but also decreases the clock rate. Implementing this component reduces the complexity associated with computing the protocol header offsets. The method involves shifting out the current protocol header to make sure the offset of the next protocol header is 0, allowing us to configure the offsets in the processing module based on the offsets in the protocol header. Figure 5 presents the hardware architecture.

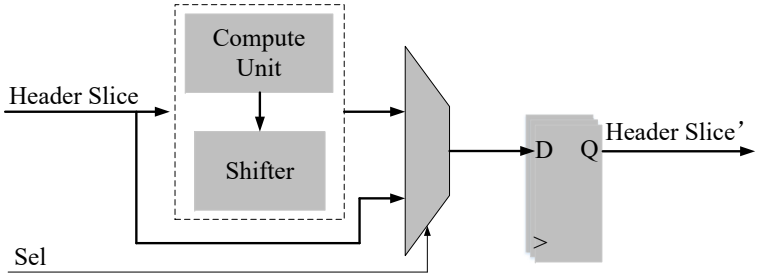

Figure 5. Block Diagram of Header Shifter.

This function consists of a "compute unit", which calculates how many bits should be shifted out; a shifter to implement the shifting; a MUX for choosing a source; and a register vector to store the header slice. The "compute unit" provides a constant shift number for protocol headers of fixed length, such as IPv6, ICMP, etc. The variable shift number for the protocols is also calculated using this function for the protocol headers of variable lengths. The selected signal "Sel" is drawn from the function of the "Type Identification" to select the new header or the original header.

For illustration, we choose the packet header of "Ethernet → IPv4 with option → TCP" as an example. After parsing the first protocol header of "Ethernet", the parser shifts it out from the header slice based on its fixed length. This makes the offset of the "IPv4 with option" to be 0. Thus, all current field offsets of the "IPv4" inside the header slice are equal to their relative offsets inside the protocol header, and no variable offsets are used. However, the length of the "IPv4 with option" is variable and shown in the field of "Header Length", and the shift number is calculated based on it. After the shifting, the offset of the "TCP" becomes 0. By implementing such a shifter, the complexity of other modules such as the "Field Extraction" and the "Type Generation", which locate fields in long vectors, would be reduced.

A dynamic shifter should be implemented to shift the protocol headers that are of variable length, which costs a lot of FPGA resources and also causes long critical paths that can reduce the clock rate. After investigating the protocol header format, we can conclude that the header lengths increase with limited times of 32 bits. For example, for the option part in the IPv4 header, the length is in the range of 0–10 times of 32 bits. If a dynamic shifter is implemented based on the limited step of 32 bits, this will reduce the resource usage and the lengths of critical paths; this is further discussed and proven in Section 6.1 below.

4.4. Function of Field Extraction

If this component is enabled, fields in the header slice are copied to specific positions in the header vector. The component consists of a "Field Indicator", an "Extractor", a MUX, and a register vector. Figure 6 presents the hardware architecture.

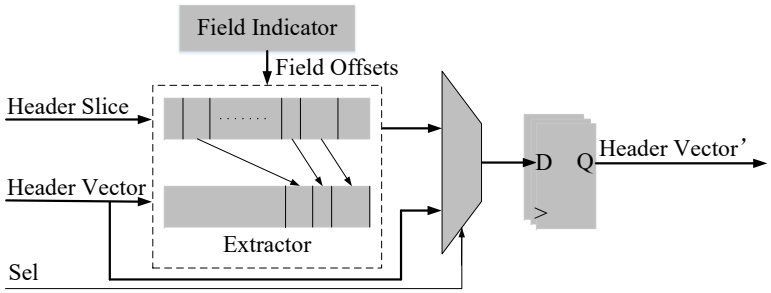

Figure 6. Block Diagram of Field Extracting.

The "Field Indicator" has many predefined arrays of addresses, due to the function of header shifter, relative addresses of extracted fields within the protocol header are used here. These addresses

indicate the begin offsets and end offsets of the specific fields in the header slice, along with the destination position for the extracted fields in the header vector. The source addresses are generated by parsing the P4 program, and the destination addresses are generated after arranging the extracted fields in the header vector during the compilation. The data copy is executed in the "Extractor" module, while the "Sel" signal from the "Type Identification" is used to select either the original header vector or the new one.

For example, we assume that a field of destination IP address in the "IPv4" protocol header will be used in the subsequent pipeline stages, so the relative start offset and end offset of this field should be stored in the "Field Indicator", and its offsets in the "Header Vector" should be stored as well. When an "IPv4" header is inputted to this module, the target field in the "Header Slice" can be copied to the corresponding position in the "Header Vector".

4.5. Function of Type Generation

The code or flag of the protocol header type is always stored in a field of its previous protocol header. The type of the next protocol is extracted or generated, and will then be updated in the "Type Record" vector in this component, which includes a "Type Indicator", "Type Generator", MUX, and register vector. Figure 7 presents the hardware architecture of the type generator.

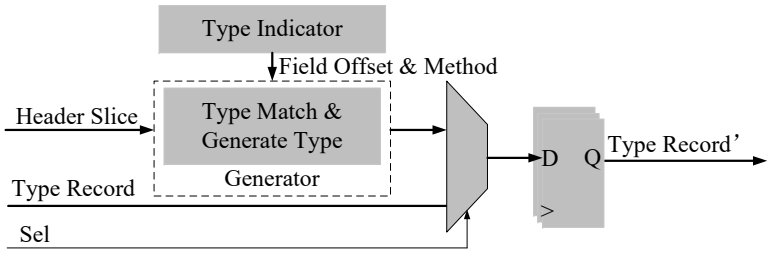

Figure 7. Block Diagram of Type Generator.

The "Type Indicator" stores the offset for the field of type code or flag in the header slice, and also indicates whether we can use the field directly or whether further processing is required. For example, the protocol header of "MPLS" does not have such a field, it thus needs the field matching using this protocol header type. Type generation for the next protocol header is executed by the "Generator" module, which also updates the type record. The "Sel" then selects either the original record or the new record for the register vector.

4.6. Other Accessorial Circuits

In addition to the components above, the parser needs to handle errors and exceptions during the parsing of the header. For example, errors or exceptions such as unrecognized protocols, checksum errors, etc. should both occur and be recorded if the parser receives malformed headers or unrecognized protocols. In addition, customized information can be extracted or generated during the parsing, such as the length of each protocol header. These conditions should be marked and transferred to the downstream, so that they can be processed by the components outside of the parser.

5. Compilation

The use of a high-level language such as P4 to design a network parser obviates the need for network experts to familiarize themselves with the details of FPGA. Here, we propose a compiler that extracts parameters from P4 programs and maps them to our VHDL templates, then gathers them together to create a complete parser. In this section, the template design is firstly introduced, followed by the compilation workflow.

5.1. Template Design

Based on the hardware introduced in Sections 3 and 4, the proposed parser consists of many small function modules, including "Type Identification", "Header Shifter", and even the primitive MUX, etc. These are applied to various parsers to support different protocol headers of different performances, such as header slice width, header vector width, etc.

We implement all functions with VHDL and leave generics for reconfiguration purposes. Templates are instantiated by configuring their parameters based on the P4 programs. All templates have been created by a skillful RTL programmer and thoroughly tested to ensure their correctness—moreover, the run speed is preferred during the creation. Table 2 lists many of the main generics in our precreated templates, which are used to decide the performance and functionality of their corresponding processing modules.

Table 2. Main parameters of the templates.

Type	Generics	Description
Port	Header width, header vector width, protocol length	Decide the ports' width, as well as the primitives' width—selector, comparator, and so on.
Functionality	Addresses, extract indicators	Decide what will be done to the header slice
Control	Protocol code, module code, optional module position	Related to the supported protocols

5.2. Compiling Process

Several steps are needed to convert a P4 program to VHDL, namely—P4 program parsing, pipeline scheduling, mapping, VHDL code generation, synthesis, and implementation. Figure 8 below shows the workflow.

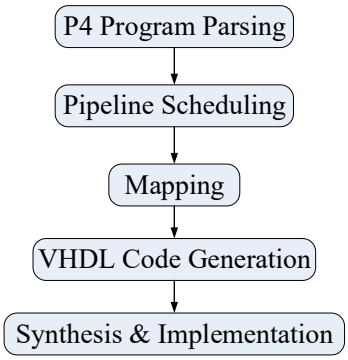

Figure 8. Workflow of the compilation.

P4 Program Parsing: By using the intermediate results from the official P4 compiler, our proposed compiler acts as a back-end compiler. In short, it extracts functions from the described parser associated with their parameter values, but for reconfiguring the VHDL templates.

Pipeline Scheduling: The parser pipeline stages are first scheduled for parsing the supported protocols without conflicts and stalling occurring. Moreover, all protocol headers can be processed sequentially by their corresponding processing modules, as described in Section 3.3.

Mapping and Generating HDL code: Once the parameters of the various functionalities in the parser are determined, the compiler maps them to their related VHDL templates. The compiler initializes the templates by configuring their parameters—examples of this include bus width configuration, updating the addresses of the fields for the templates, etc. Finally, the compiler generates VHDL code by instantiating all templates used along with their wrappers.

Synthesis and Implementation: The VHDL code can be synthesized and implemented using the standard EDA tools, such as Vivado from Xilinx or Quartus from Intel, for various FPGAs.

6. Evaluations

Two experiments were carried out in order to evaluate our proposed solution. To implement these experiments, a Xilinx Virtex-7 serial (the same as that in [16]) and a Xilinx Vivado 2015.4 are used.

6.1. Dynamic Shifter Evaluation

The dynamic shifter shifts a vector with variable number bits, and may thus use up a lot of resources in the FPGA. Based on the network protocols, the lengths of headers with options are always increased in steps of limited times of 32 bits. This experiment evaluates whether such a dynamic shifter saves resources and/or runs at a high clock rate. Three shifters are implemented with 1024-bit vector width to shift a 480-bit IPv4 header. Comparison results are shown in Table 3.

Table 3. Shifter Comparison.

Type	LUTs	FFs	Clock Rate (MHz)
Fixed	0	545	714.3
Proposed	1639	865	533.6
Fully Dynamic	5068	1025	306.6

From Table 3, the fixed shifter (which shifts a fixed number of bits for the vector) uses minimal resources and runs at a very high clock rate. The fully dynamic shifter has the maximum flexibility, as it can shift any number of bits for the vector, however, it also uses maximum resources and runs at the lowest clock rate. By contrast, our proposed solution uses an average amount of resources and an average clock rate, which is in line with our expectations.

6.2. Parser Performance Evaluations

To demonstrate and evaluate our proposed solution, we compare it with the same two types of parser used in [15,16]:

- **Simple parser**: Ethernet, IPv4/IPv6 (with 2 extensions), UDP, TCP, and ICMP/ICMPv6;
- **Full parser**: Same as the simple parser but also includes MPLS (with two nested headers) and VLAN (inner and outer).

To implement these two parsers, the values of key parameters are listed in Table 4.

Table 4. Key Parameter Values.

Parameters	Extract Type	Simple Parser	Full Parser	Notes
Pipeline Stage Number	5 Tuple & All Fields	5	7	After the pipeline scheduling, it indicates how many cycles for parsing a packet
Header Slice Length	5 Tuple & All Fields	1072	1136	Calculated by the supported longest packet header
Header Vector Length	5 Tuple	296	296	Calculated by the lengths of all extracted fields
	All Fields	1072	1136	
Error Types	5 Tuple & All Fields	Unrecognized, IPv4 Valid, TCP Valid		
Individual Modules	5 Tuple & All Fields	Such as protocol header length, type code, field indicators, positions, and so on, they vary due to different protocols.		

Table 5 below presents a comparison of the results of our proposed solution and those obtained by the other two works presented in [15,16] in terms of throughput, latency, and resource usage. In this table, the parser labeled "Golden [15]" is a handwritten implementation in [15], while the other parsers labeled [15,16] are implemented using the methods set out in the corresponding literature. The two parsers with the same labels of the same type are distinguished by different fields.

Table 5. Comparison Results.

Work	Performance				Resources			Extracted Fields
	Data Bus [bits]	Frequency	Throughput	Latency	LUTs	FFs	Slice Logic (LUTs + FFs)	
Simple Parser								
Golden [15]	512	195.3	100	15	N/A	N/A	5000	TCP/IP 5-tuple
[15]	512	195.3	100	29	N/A	N/A	12,000	TCP/IP 5-tuple
[16]	320	312.5	100	19.2	4270	6163	10,433	TCP/IP 5-tuple
[16]	320	312.5	100	19.2	5888	10,448	16,336	All fields
Proposed	1072	346	370	14.45	4884	3135	8019	TCP/IP 5-tuple
Proposed	1072	334.4	358	14.55	10,495	6886	17,381	All fields
Full Parser								
Golden [15]	512	195.3	100	27	N/A	N/A	8000	TCP/IP 5-tuple
[15]	512	195.3	100	46.1	10,103	5537	15,640	TCP/IP 5-tuple
[16]	320	312.5	100	25.6	6046	8900	14,946	TCP/IP 5-tuple
[16]	320	312.5	100	25.6	7831	13,671	21,502	All fields
Proposed	1136	320.5	364	21.84	9515	6930	16,445	TCP/IP 5-tuple
Proposed	1136	279.3	317	25.06	16,888	12,033	28,921	All fields

Due to the hardware architecture, the bus widths of our proposed parsers are 1072 and 1136, which correspond to the longest header ("Ethernet → IPv4 with option → TCP with option") in the simple parser and the header of "Ethernet → VLAN (MPLS) → VLAN (MPLS) → IPv4 with option → TCP with option", respectively. All extracted fields include options in IPv4 and TCP headers.

Timing Comparison: Our proposed parsers and the parsers of [16] run at a clock rate of around 300 MHz. This can be compared to the parsers of [15], with a clock rate of only 195 MHz. The clock rates of our proposed parsers decrease slightly as the data bus width increases, however, the clock rate of the full parser that extracts all fields from the header decreases to nearly 280 MHz under these circumstances. Total latencies in the table show that a packet takes only 14.55 ns to go through our proposed parser, which means that our proposed parser has a minimal processing time. Pipeline stages are calculated based on the clock rate and latency presented in the table. For illustrative purposes, only the comparison results of the simple parsers are presented, as the same conclusions are found among the full parsers. The stage numbers in the simple parser type are 3, 6, 6, 6, 5, and 5, respectively. Moreover, while the total latency of the Golden parser is 15 ns, the clock period is more than 5 ns. While [15] has the same pipeline stages as [16], the critical paths between the two pipelines are still long, which causes a low clock rate. Compared with our proposed parsers, the parsers of [16] have one more pipeline stage due to the pipeline scheduling method adopted.

Throughput Comparison: When directly compared to the parsers of [15,16], our proposed parsers have higher throughput, achieving more than 300 Gbps—however, this comparison is based on situations involving different data bus widths and clock rates. We can assume that the clock rate will not decrease for double and triple, [15,16], respectively, and that their throughputs are still lower than ours. In addition, as our parser processes only the header slice, which is part of the packet, the throughput should be higher than that recorded in the above table.

Resource Usage Comparison: As these parsers have differing hardware architectures, they use different resources for their implementation and also run at different clock rates, thus, it would not be fair to compare their resource usage directly. However, it is possible to conduct a comparison based

on throughputs by counting the slice logic (LUT + FF) of the FPGAs. Since our proposed parsers (of both simple type and full type) have throughputs of 358 Gbps and 317 Gbps, respectively, we use a middle value of 320 Gbps as the coordinator on the X-axis. The resource usage comparison is shown in Figure 9.

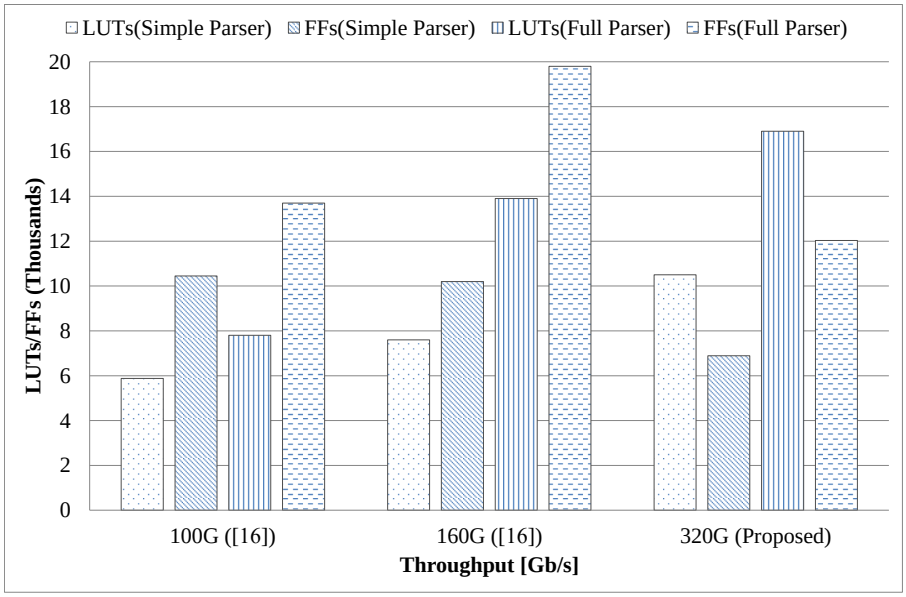

Figure 9. Resource usage comparison.

For illustrative purposes, only a comparison of full parsers is presented here. Figure 9 shows that [16] uses about 22 thousand slice logics to achieve 100 Gbps, and uses about 34 thousand slice logics to achieve 160 Gbps; by contrast, our generated parser uses about 29 thousand slice logics to achieve nearly 320 Gbps; also, no resources of other types (such as block RAM) are used in our parsers. In addition, when the proportions of resource usage are calculated, our most complicated parser is found to use less than 5% LUTs and 2% FFs.

7. Conclusions

We have proposed a framework here that includes a pipeline-based hardware infrastructure for the parser, along with an approach that converts P4 programs to VHDL. The parsers can be generated and implemented on FPGAs automatically by using this framework. Our proposed method has the following features:

- A hardware structure is configured to different parsers of varying performances.
- The pipeline of the parser is scheduled by the compiler to avoid conflicts and stalling—the fully pipelined parser processes one packet per cycle.
- Prebuilt function templates are used to generate the final VHDL code. The templates are all well-designed and have been thoroughly tested.

The framework rapidly converts the P4 program to VHDL, which greatly reduces the effort involved in designing the parser for the FPGA platform. Experimental results demonstrate that the generated parser uses FPGA resources efficiently and can achieve a line rate of around 320 Gbps.

Author Contributions: Z.C. proposed the methodologies of this paper and wrote the original draft preparation. H.Z. and J.L. wrote part of the code and reviewed the manuscript. C.Z. and M.W. supervised the implementation of the methodology.

Funding: This research is supported by the National Key Research and Development Program under No.2016YFB1000400, National Key Program JZX2017-1585/Y479 of China.

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

References

1. Switches for Every Network. Available online: https://www.cisco.com/c/en/us/products/switches/index.html (accessed on 15 December 2018).
2. Network Switches. Available online: https://e.huawei.com/en/products/enterprise-networking/switches (accessed on 15 December 2018).
3. Morgan, J. Intel® IXP2XXX Network Processor Architecture Overview. 2004. Available online: https://www.slideserve.com/bikita/intel-ixp2xxx-network-processor-architecture-overview (accessed on 5 May 2019).
4. Networks, B. Tofino: World's Fastest P4-Programmable Ethernet Switch ASICs. Available online: https://barefootnetworks.com/products/brief-tofino/ (accessed on 2 December 2018).
5. Kohler, E.; Morris, R.; Chen, B.; Jannotti, J.; Kaashoek, M.F. The Click Modular Router. *ACM Trans. Comput. Syst.* **2000**, *18*, 263–297. [CrossRef]
6. Lovato, J. Data Plane Development Kit (DPDK) Further Accelerates Packet Processing Workloads, Issues Most Robust Platform Release to Date. Available online: https://www.dpdk.org/announcements/2018/06/21/data-plane-development-kit-dpdk-further-accelerates-packet-processing-workloads-issues-most-robust-platform-release-to-date/ (accessed on 10 December 2018).
7. Benzekki, K.; El Fergougui, A.; Elbelrhiti Elalaoui, A. Software-defined networking (SDN): A survey. *Secur. Commun. Netw.* **2016**, *9*, 5803–5833. [CrossRef]
8. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* **2008**, *38*, 69–74. [CrossRef]
9. Kreutz, D.; Ramos, F.M.; Verissimo, P. Towards Secure and Dependable Software-defined Networks. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, China, 16 August 2013; pp. 55–60. [CrossRef]
10. Wan, T.; Abdou, A.; van Oorschot, P.C. A Framework and Comparative Analysis of Control Plane Security of SDN and Conventional Networks. *arXiv* **2017**, arXiv:1703.06992.
11. 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. *SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 87–95. [CrossRef]
12. Li, B.; Tan, K.; Luo, L.L.; Peng, Y.; Luo, R.; Xu, N.; Xiong, Y.; Cheng, P.; Chen, E. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; pp. 1–14. [CrossRef]
13. P4 Language. Available online: https://github.com/p4lang/ (accessed on 5 October 2018).
14. Singh, S.; Greaves, D.J. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines, Palo Alto, CA, USA, 14–15 April 2008; pp. 3–12. [CrossRef]
15. Benácek, P.; Pu, V.; Kubátová, H. P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers. In Proceedings of the 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Washington, DC, USA, 1–3 May 2016; pp. 148–155. [CrossRef]
16. Santiago da Silva, J.; Boyer, F.R.; Langlois, J.P. P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 147–152. [CrossRef]
17. Attig, M.; Brebner, G. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, Brooklyn, NY, USA, 3–4 October 2011; pp. 12–23. [CrossRef]
18. Gibb, G.; Varghese, G.; Horowitz, M.; McKeown, N. Design Principles for Packet Parsers. In Proceedings of the Architectures for Networking and Communications Systems, San Jose, CA, USA, 21–22 October 2013; pp. 13–24. [CrossRef]

19. 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. *SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 99–110. [CrossRef]
20. Pus, V.; Kekely, L.; Korenek, J. Low-latency Modular Packet Header Parser for FPGA. In Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Austin, TX, USA, 29–30 October 2012; pp. 77–78. [CrossRef]
21. Cabal, J.; Benáček, P.; Kekely, L.; Kekely, M.; Puš, V.; Kořenek, J. Configurable FPGA packet parser for terabit networks with guaranteed wire-speed throughput. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 249–258. [CrossRef]
22. Yazdinejad, A.; Bohlooli, A.; Jamshidi, K. P4 to SDNet: Automatic Generation of an Efficient Protocol-Independent Packet Parser on Reconfigurable Hardware. In Proceedings of the 8th International Conference on Computer and Knowledge Engineering (ICCKE 2018), Mashhad, Iran, 25–26 October 2018; pp. 159–164. [CrossRef]