IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 7, 2020, accepted January 25, 2020, date of publication January 30, 2020, date of current version February 6, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2970683

P4 to FPGA-A Fast Approach for Generating Efficient Network Processors

ZHUANG CAO, HUAYOU SU, (Member, IEEE), QIANMING YANG, JUNZHONG SHEN, MEI WEN, AND CHUNYUAN ZHANG, (Member, IEEE)
College of Computer, National University of Defense Technology, Changsha 410073, China

Corresponding author: Qianming Yang (yqm21249@nudt.edu.cn)

This work was supported in part by the National Key Research and Development Program under Grant 2016YFB1000400, in part by the National Key Program of China under Grant JZX2017-1585/Y479, and in part by the Youth Fund Project from the National Natural Science Foundation of China under Grant 61802420.

ABSTRACT This paper presents a framework for converting P4 programs to VHDL and then implementing them on Field-Programmable Gate Array (FPGA) platforms. In this framework, a match-action-based hardware architecture is introduced with clearly designed components, which correspond to the described functionalities in the P4 programs. A pre-built template library is used for the compilation that includes optimized VHDL templates corresponding to specific clearly designed components. From the output of a standard frontend P4 compiler, the proposed compiler extracts parameters and relationships within the functions being employed, maps them to corresponding templates by calling, configuring, optimizing and instantiating them, and finally generates the appropriate FPGA code. A pre-built evaluation library is also proposed that helps the compiler to optimize the implementation during the mapping phase. A prototype of this framework is also implemented and evaluated; in this process, it is found that the generated processors use few resources and have high throughput and low latency. Compared with a state-of-the-art solution, the packet processing time is halved. In addition, the generated processors are able to operate at a line rate of nearly 100 Gigabits per second for a basic layer-3 forwarding application.

INDEX TERMS P4, VHDL, FPGA, template, network processor.

I. INTRODUCTION

Network devices use various types of processors, including Central Processing Units (CPUs), Application-Specific Integrated Circuits (ASICs), and Network Processors (NPs), as the processing cores. The variety in the scope of the applications is not fully addressed by the inflexible ASIC designs, while high computation requirements and flexible demands make CPUs and NPs an inefficient alternative. These different types of processors are used in different application scenarios: for example, CPUs have the highest flexibility but lowest performance, and are usually used on software routers [1], [2], while ASICs have the highest performance but poor flexibility and are used in high-performance network devices [3], [4]. Moreover, NPs [5] have limited programmability and average performance, and are used in devices that require some programmability and do not present a performance challenge. If they are to meet different user requirements, E-commerce platforms, data centers and internet service providers (ISPs) require the ability to deploy flexible high-speed networks; this drives the need for flexible high-performance network processors.

Software Defined Networks (SDNs) [6], [15]–[17] have been widely deployed in various networks. To support this kind of network, devices should be Protocol-Independent, quickly developed/ deployed, and high-performance. To meet these requirements, Reconfigurable Match Tables (RMTs) [7] and disaggregated Reconfigurable Match-action Tables (dRMTs) [8] have been proposed to reconfigure ASICs. These two hardware architectures are designed based on the ''match-action'' architecture, which implements a rigid sequence of match-then-actions. Each ''match-action'' is clearly defined with different memories and various logic operations, such as ''and'', ''or'', ''shifter'', ''arithmetic'', etc.

When combined with high-level Domain Specific Languages (DSLs) [11], [12] and their corresponding compilers [13], [14], the use of ''match-action'' architecture

The associate editor coordinating the review of this manuscript and approving it for publication was Junaid Shuja.

can greatly reduce the design effort required. One such language is the P4 language [11], which was proposed in 2014 by Nick Mckeown et al. in order to improve the programmability of the data plane. The P4 language is reconfigurable, protocol-independent, platform-independent, and easy to use. It means that the developers can customize new protocols or process existing protocols on demand; it is also independent of any specific hardware platform, allowing the P4 programs to be quickly ported among different platforms such as hardware switches, FPGAs, SmartNICs, and software switches after compiling by hardware-related back-end compilers, thereby improving development efficiency.

Advances have been witnessed in new processors [9], [42], which combine ''match-action'' architecture with the P4 language. Reference [9] is the fastest programmable network processor and can achieve a line-rate of 6.5 Tbit/s, which far exceeds that of traditional switches. However, there are three main disadvantages of these new programmable processors.

- Devices containing processors of this kind are expensive [9] for small-scale application scenarios that use only a few ports, but are required to be high-performance and reconfigurable. For example, co-processors running special customized protocols are deployed to specific network nodes in data center or E-commerce platforms.
- The capability to execute ''external'' functions (defined as being capable of performing additional operations on the packets apart from forwarding) is also important for router operations, such as packet inspection, decoding, encoding, etc.; however, such processors lack the ability to perform these functions with high performance due to the hardware architecture.
- Although these processors can be programmed, their table sizes, bus widths, and resource types cannot be changed, and new instructions cannot be added; in other words, new features and specific resource-intensive applications cannot be supported.

Due to the ''match-action'' architecture which can be decomposed into various components of different functionalities, these components can be easily abstracted to templates. Due in part to the advantages provided by the ''match-action'' architecture and the P4 language, FPGAs provide a viable approach to addressing these problems. When combined with P4, this flexible hardware platform can greatly reduce the effort required to design a match-action-based processor. Accordingly, in this paper, we propose a framework to implement P4 programs on FPGAs: this framework includes a customizable hardware architecture based on ''match-action'', as well as a compiler that converts P4 programs to VHDL. The main contributions of this paper are as follows:

- The design of a match-action-based hardware architecture for FPGAs, which can be assembled from many independent hardware components for different functionalities. These components are clearly defined, and their corresponding VHDL templates have been well-designed and thoroughly tested by FPGA experts in order to reduce the processing latency and resource usage.
- The design and implementation of a framework that converts P4 programs to VHDL in order to produce an FPGA-based network processor. A simple mapping algorithm is proposed that extracts parameters from P4 programs, then maps them to VHDL templates; moreover, a simple back-end compiler is used in our framework. In addition, judicious pipeline scheduling for the parser, deparser, and match-action tables are also proposed.
- The creation of an evaluation library, which includes resource usage and timings of each template with typical configurations, and is used in the compilation process to optimize the design and fast estimate the performance of the designs before synthesizing.

Experiments were also carried out to evaluate the proposed framework. The results demonstrate that the experimental P4 programs are converted to VHDL efficiently, and the generated processors consume few resources and have low latencies. The processors can achieve a line rate of nearly 100 Gbps for a basic layer-3 forwarding application.

The remainder of this paper is organized as follows. Section II reviews the researches on designing efficient network processor with reconfigurable ASICs and FPGAs. Section III presents the micro-architecture of the hardware for the FPGA platform. Section IV describes the working flow of the compiler, which converts the P4 programs to VHDL by mapping the functions in the P4 programs to corresponding components in the hardware architecture. In Section V, the proposed framework is evaluated in terms of resource usage, clock rate, throughput and latency, which are extracted from the generated processors or components. Finally, future work and conclusions are discussed in Sections VI and VII.

II. RELATED WORKS

Many researchers have studied the problem of how to rapidly design an efficient network processor. Approaches include the use of reconfigurable chips and FPGAs.

The use of ''softly'' defined networks was first proposed in 2014 [18]. This method aims to convert Px++ [39] language programs to a low-level Hardware Description Language (HDL), then synthesize and implement it to bitstreams [19]. A network development tool ''SDNet'' based on an FPGA platform was designed in [20]. Native support for the P4 language by SDNet allows for the conversion of P4 to verilog [21]. This tool has been continuously updated, and its license is made available to people who work in academic fields supported by the Xilinx University Program.

NetFPGA [22]–[24] provides an open low-cost reusable platform that allows network researchers to build Gbps-level high-performance network system models on hardware for next-generation networks. This platform supports modular design, and complex design can be executed through the combination of various modules. In addition, the existing development resources (such as peripheral modules, drivers,

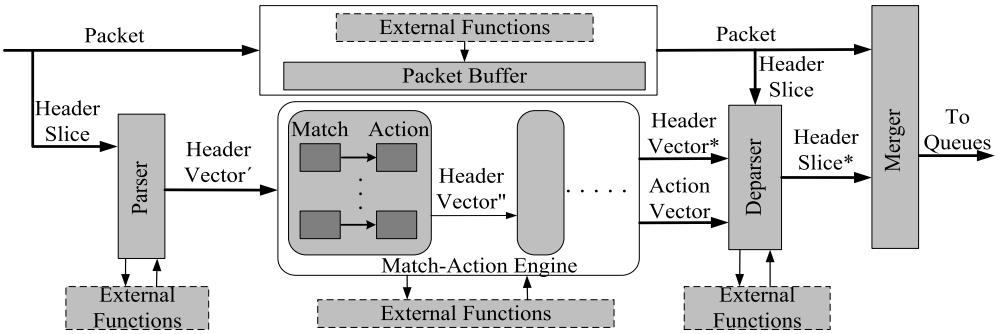


FIGURE 1. Hardware architecture. This figure illustrates the relationship between the different components and the packet processing within the processor. "External" function interfaces are reserved in the parser and deparser for customized applications.

GUIs, etc.) can be shared with developers, thus reducing the need for repetitive development and simplifying the task of network research. At present, this platform is used for experimental courses in many universities to implement networks and routers. However, compared with the state-of-the-art solutions, the performance of its generated network processors is insufficient.

P4FPGA [25] presents a solution in which P4 programs are first converted to Bluespec [40] programs. The Bluespec compiler is used to generate Verilog source code, which can be processed with standard EDA tools. In EMU [28], the Kiwi compiler [14] is used to convert C# programs to Verilog. In [31], moreover, a line-rate parser is used to convert P4 programs to a high-level language such as C++, which is followed by the use of a high-level synthesis (HLS) development tool kit.

The abovementioned methods mainly make use of the flexibility of high-level languages and existing compilers to reduce the development time required. This approach results in the performance of the generated circuits being dependent on third-party compilers, which are limited by the HLS technologies used for FPGAs; moreover, multi-step conversion results in greater performance losses. Furthermore, some intermediate results are opaque to users during the multi-step conversion process, making manual intervention difficult.

Benáček et al. present another solution, "P4-TO-VHDL" [26], [27]. As laid out in the structure of its hardware architecture, the compiler reads the P4 programs and then generates synthesizable VHDL codes for blocks of the network processor to various FPGAs; these generated processor blocks are able to reach high speeds of around 100 Gbps. However, the development of this solution is still in progress, and it can currently only generate the blocks of the network processors' parser and deparser. In addition, this solution only supports $P4_{14}$.

Compared with these state-of-art solutions, the advantages of our proposed framework are numerous. First, components are clearly defined in the hardware architecture, corresponding to the functionalities described in the P4 programs, and abstracted to templates with the help of FPGA experts. Implementations of this kinds can achieve high performance while using fewer resources. Second, a simple mapping method is introduced that implements conversion only once, so as to avoid any performance loss caused by a series of language conversions. Moreover, it also avoids designing complicated compilers; these compilers also help to decide the performance of the designs. Third, the evaluation library helps to optimize the design in the compilation process, thereby improving the design performance. In addition, the fast reporting helps programmers to estimate resource usage and clock frequency in advance and modify their programs accordingly, which saves synthesization time and improves the development efficiency.

III. FPGA HARDWARE ARCHITECTURE

The key advantages of designing an ASIC are that the resources and their locations can be arranged according to requirements, allowing resource and clock rate limitations to be avoided within the range allowed by the chip manufacturer. Compared to ASICs, FPGAs can also be customized, albeit with limitations. Firstly, programmable primitives of FPGA are pre-laid out in the chip; this fixed position may affect the wiring and clock rate. For example, due to the limited number of wires between adjacent blocks, blocks with farther distances will be involved in implementing some functions. This leads to an increase in wiring complexity and propagation delay, which reduces the clock rate. Secondly, limited resources are not always able to meet the requirements of various applications, since these applications may incur heavy resource costs in terms of Lookup Table (LUT), block RAM, or other resources. Due to these inherent characteristics of FPGA, it is not appropriate to copy the ASIC architectures to FPGA directly, as this may lead to a lack of particular resources or other unacceptable performance. However, components in the architectures can be reorganized and optimized for FPGAs, enabling the full use of limited resources and the exploitation of parallelism.

A. MICRO-ARCHITECTURE

As shown in Fig.1, this hardware architecture is primarily composed of a parser, match-action engine, deparser, packet buffer, merger and "external" functions; other memories and

accessorial components, such as FIFOs, Muxes, etc., can be added if necessary. These function modules can be further decomposed into many components. For example, the match-action engine is mainly composed of Content Addressable Memory (CAM), Random Access Memory (RAM), and various logic operations. By configuring the parameters of different components and then organizing them based on the designs, network processors with different functions and performance capabilities can be constructed.

Based on the hardware structure shown in Fig.1, when a packet is taken into the processor, a header slice is split from the packet and then sent to the parser. Some fields of the header slice are extracted and loaded into the header vector, which stores these fields along with some accessorial values during the parsing process. Some fields in the header vector are used as a key to facilitate matching, while others are edited in the match-action engine. The deparser updates the header slice with the new field values in the header vector. Finally, the merger merges the new header slice with its corresponding packet to create a new packet and then send it to the output queues. ''External'' functions are scheduled such that their functions are executed in parallel with the forwarding components and their results are also kept in alignment with the data stream. In contrast with the match-action architecture of ASICs, this architecture can be customized based on the P4 program and there is no need to reserve circuits for unused functions.

B. IMPLEMENTING ''EXTERNAL'' FUNCTIONS

As defined in Section I, ''external'' functions perform additional operations on the packets. Implementing such ''external'' functions in network processors helps in easily offloading the upper-layer applications to the hardware, which improves the forwarding performance of the switches. Although these ''external'' functions can be implemented in the programmable network processors (ASICs) on demand, they are executed through the execution of a large number of instructions by Arithmetic Logic Units (ALUs), a method consumes a large number of processing cycles. In addition, it is difficult to change their start execution times due to the fixed hardware structure; this may result in stalls if they are unable to generate results in time. However, FPGAs can be implemented with circuits that generate the results within far fewer cycles, and their start execution times can also be adjusted by scheduling the pipelines, thereby avoiding or reducing the stalls.

There is a simple example of implementing the ''external'' function. We assume that our switches need to certify the data for each incoming Secure Socket Layer (SSL) protocol packet using MD5 [43]. The function dictates that if the calculated hash code does not match that in the incoming packet, this packet should be abandoned. The SSL protocol is a kind of protocol that runs over the TCP/IP and allows the incoming packet to be forwarded even without MD5 certification. This certification function can be implemented on FPGAs via programming and generates the result in only a few cycles.

TABLE 1. Conflict and halt in header parsing.

Packet Header	Clock number			
	1	2	3	4
Ethernet→VLAN→IPv4	Ethernet	VLAN	IPv4	
Ethernet→IPv4		Ethernet	Stall	IPv4
Ethernet→			Stall	Ethernet

This function begins to execute once the TCP header is parsed and is executed with other independent functions in parallel. The generated result can be used without any stalls occurring through arranging the pipeline, thereby improving the performance.

C. PARSER AND DEPARSER

In our proposed hardware architecture, the parser takes in packet headers and extracts specific fields to update the header vector. By contrast, the deparser accepts updated header vectors from the match-action engine and updates the original packet header. There are many parsers available for ASICs or FPGAs [29]–[32]. Some FPGA implementations can achieve very high performance but require very long pipeline stages or extensive resources. Moreover, they generally do not allow execution of the ''external'' functions while parsing the packet header, a feature that would work to improve parallelism and reduce processing latency. In addition, some non-pipelined parsers based on Finite State Machines (FSMs) begin to process the next packet only after parsing the previous one, which reduces the throughput.

We propose a pipelined parser that takes full advantage of the reconfigurable nature of FPGAs. This parser is designed and scheduled based on the supported network protocols, and each pipeline stage in the parser processes one protocol in each cycle. A previous version of this idea was presented in [33]; we will briefly review it and add new material regarding the ''external'' function to the pipeline scheduling in this section.

1) PIPELINE SCHEDULING

A parse graph is established based on parse dependencies among the supported protocols. Fig.2 (a) presents an example of such a graph. In the parse graph, each node represents a pipeline stage to process a protocol header or execute an ''external'' function, while the arrows denote the parsing sequence. Each path (no ''external'' function) in the graph indicates a protocol transition within the packet header, and all these paths are parsed by the parser.

Due to the presence of interlock branches in the parse graph, it would be unwise to implement such a parser in FPGA, as this would result in low performance due to conflicts and halts. For example, suppose that three different packets are inputted sequentially to the parser; Table 1 lists the conflicts and halts during the parsing.

The parser accepts the first packet and processes ''Ethernet'' in the first cycle. It then processes ''VLAN'' and

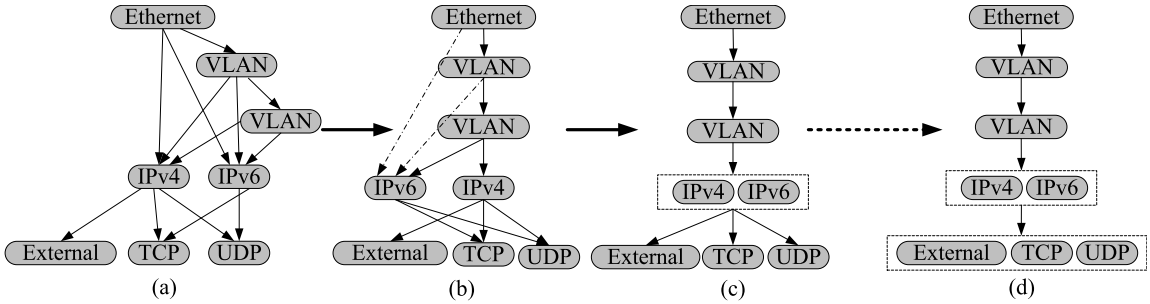


FIGURE 2. Parse graph and pipeline scheduling. (a) is the initial parse graph from the P4 program, (b) and (c) represent the intermediate process of the pipeline scheduling, and (d) is the final scheduling result.

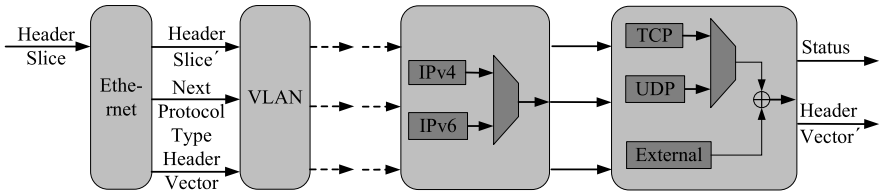


FIGURE 3. Hardware structure of the parser. Arrows indicate the direction of data flow.

accepts the second packet in the second cycle. However, in the third cycle, two "IPv4s" need to be processed in the same pipeline stage; this conflict forces the second packet to halt, meaning that the third packet cannot be accepted due to the halting of the second packet.

The proposed pipeline scheduling approach solves this problem by implementing the following steps. For illustrative purposes, Fig.2 (a) is taken as an example to help demonstrate this method, while the remaining graphs in Fig.2 presents the pipeline scheduling process. The corresponding pipeline scheduling steps are laid out below.

1) Find one of the longest paths in the parse graph. There are several longest paths containing five nodes, such as "Ethernet → VLAN → VLAN → IPv4 → TCP" and "Ethernet → VLAN → VLAN → IPv6 → UDP"; for illustration, we choose the first path and take it as a trunk;
2) Choose any one of the remaining non-trunk nodes and find all of its parents in the trunk. Then, reserve the dependency relationship with the last node in the trunk and delete all other dependencies. If "IPv6" is selected as an example, the reserved dependency should be that between "IPv6" and the second "VLAN" in the trunk (shown in Fig.2(b)), while dependencies marked by dotted arrows will be deleted.
3) Merge this chosen node with its brother in the trunk to form a new one and update the dependencies of its children (this process is outlined in Fig.2(c)).
4) Go through the remaining non-trunk nodes one by one and add them to the trunk by repeating steps 2 and 3. The schedule is completed when all nodes are in the trunk; Fig.2(d) shows the result.

Furthermore, in our example, we assume that there is an "external" function that depends on some fields of the "IPv4" header, and this "external" function can only start executing after the "IPv4" header has been parsed. This function can be treated as an independent protocol during the scheduling, and is scheduled to the same level as the "TCP" and "UDP" protocols. Since the parser has a fixed number of pipeline stages, and all packet headers go through these stages, the incoming packets are parsed within constant latency, which is decided by the pipeline stage number.

2) PIPELINE CONFIGURATION

After scheduling the pipeline, the parser can be configured and then instantiated, e.g. the width of the header slice and header vector, as well as the parameters for each pipeline stage. Fig.3 presents the hardware diagram of the illustrated case.

The width of the header slice is equal to the longest header supported by the processor, regardless of the length of the incoming packet, and the width of the header vector is decided by the number of extracted fields and the accessorial data. Each pipeline stage in the parser has the function of extracting fields, updating the header vector, and generating its next header type, etc. It works only the incoming header slice includes corresponding protocol header, otherwise, the header slice goes through without any processing.

3) PIPELINE CONFIGURATION FOR THE DEPARSER

The deparser has a similar pipeline architecture to the parser, but different functions. Firstly, the deparser pipeline is scheduled using the same method described in Section III-C1. Secondly, each pipeline stage has similar parameters to the pipeline stages in the parser. When the data are ready, the deparser takes in the header slice from the packet and the header vector, then edits the header slice by using the fields in the header vector (including adding / deleting / updating the

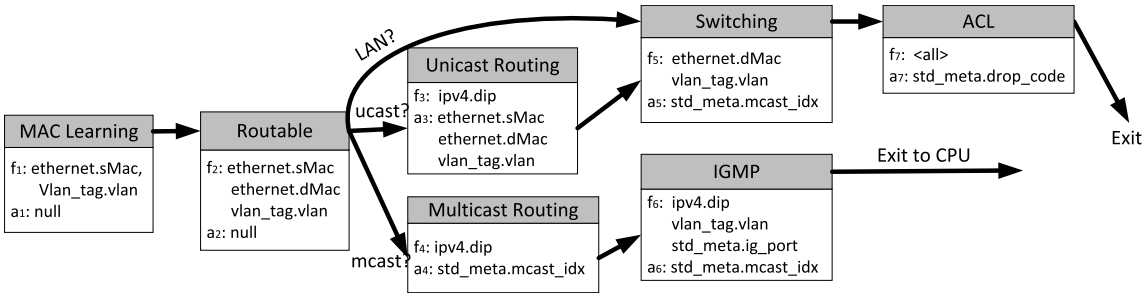


FIGURE 4. Match-action flow in the L2L3 program [41]. Here f_i indicates the match fields, while a_i indicates modified fields in the table.

field in the header slice). The new header slice will be sent out to the "Merger" in the hardware architecture, enabling the header slice in the packet to be replaced with this new one.

D. MATCH-ACTION ENGINE

Match-action engine is a control flow among a series of match-action tables. This flow decides how the incoming packets are forwarded, and decisions such as routing, dropping, calculating, statistics, etc. are made during this processing. When a header vector is inputted in, some fields of the vector are used for matching to find out their related information, for example, the destination IP addresses are always related to the output port, MAC address of the next-hop, etc., while some other fields are involved in the actions, they can be operands or to store processed results. Before describing our idea for this section, we first acknowledge the authors of [41], who presented methods for mapping logical match-action engine to reconfigurable switches; their work has inspired our ideas for FPGAs. In this section, we borrow their example program, namely L2L3: this program describes a simple L2/L3 IPv4 switch and its match-action engine, consisting of seven tables. Tables in this example are shown in Fig.4.

Firstly, the source MAC address and VLAN tag are matched in the tables "MAC Learning" and "Routable". The match result from "Routable" decides which table will be next in the data flow: "Switching", "Unicast Routing" or "Multicast Routing". Subsequently, the data flow will complete matching and action in the subsequent tables, such as table "ACL" and "IGMP".

1) SUB MATCH-ACTION MODULE DESIGN

Tables in the match-action engine consist of match tables, registers and action units, as shown in Fig.5. Part of the fields in the header vector are used to do matching in the table, with some others used to be handled in action units or "external" functions. Match results from the match table and some fields from the header vector are operated by the action units. Finally, results from the action units and "external" functions are stored in the action vector and sent to its subsequent tables.

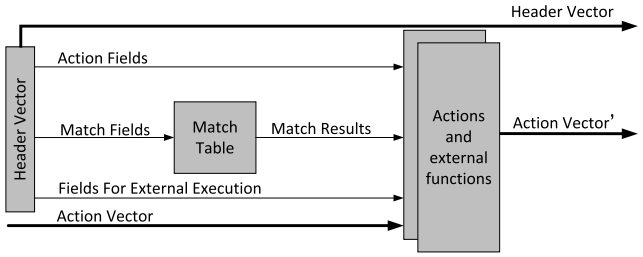


FIGURE 5. Micro-architecture of the table in the match-action engine.

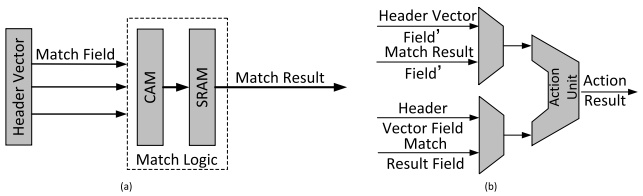


FIGURE 6. Match logic module and action module. (a) indicates the match logic and (b) indicates the action unit.

The key components in each table are the match logic and action units. The match logic is illustrated in Fig.6(a), while the action unit is shown in Fig.6(b).

Match logic comprises CAM and SRAM. CAM [34]–[36] is a kind of memory used in certain very-high-speed searching applications. It is used to compare an input key-value against a table of stored data and return the address of matching data. Since expensive CAM takes up a large number of LUTs or block RAMs in FPGAs, it is usually used to store values of match key type and output indexes in order to select big data stored in SRAM. One or more fields combined to form one key in the header vector can act as the match key for the CAM; the result from CAM selects the content in the SRAM, and this selected SRAM will be outputted to action units.

A set of logical operations in the action units edit the fields from SRAM, the header vector, and the action vector. The action vector stores the results of each action unit in all tables. "External" functions are executed in parallel with these operations. Taking the basic layer-3 forwarding as an example, the CAM stores the destination IP addresses while other forwarding information (e.g. output port, MAC address of the next hop, etc.) can be stored in SRAM. The action edits

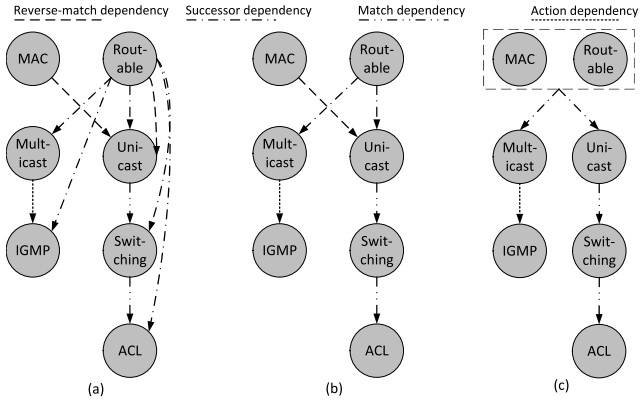


FIGURE 7. Table dependency; here, the nodes indicate the table, while the edges indicate the dependencies.

the TTL value and finally updates these forwarding data to the action vector.

2) DEPENDENCIES AMONG MATCH TABLES AND SIMPLIFICATION

There are four dependency types among tables in the match-action engine, as abstracted in [41]: ''Match dependency'', ''Action dependency'', ''Successor dependency'', and ''Reverse match dependency''. Based on the dependency types, we convert Fig.4 to a Table Dependency Graph (TDG), which is presented in Fig.7(a).

In this figure, the matching result of the table ''Routable'' determines where the packet data goes next—to the table ''Unicast'', ''Multicast'', or ''Switching''—and the successor dependencies. The fields ''ethernet.sMac'' and ''vlan_tag.vlan'' are modified in the table ''Unicast'', and will be the matching key in the table ''Switching'' for the next step; this is match dependency. The table ''Multicast'' and the table ''IGMP'' update the field of ''std_meata.mcast_idx'' one after another, which constitutes an action dependency between them. Finally, the matching executions of the tables ''MAC Learning'' and ''Routable'' need to be completed before the action of the table ''Unicast''; accordingly, this constitutes a reverse match dependency between the former and the latter.

Aspects of pipeline layout, such as pipeline branch number, pipeline stage, and table order in the pipeline, can be decided based on these dependencies. For example, match dependency decides that the latter table begins to execute matching after the completion of its former table. By contrast, action dependency decides that the action of the former table and the match operation of the latter table can be executed in parallel. Obviously, there may be multiple dependencies between nodes; these dependencies can be confusing when laying out the tables in the match-action engine, where only the strictest dependencies decide the final layout. For example, if both ''Successor dependency'' and ''Match dependency'' exist between two tables, the ''Match dependency'' determines the layout of the table, while the ''Successor dependency'' can be ignored.

Due to the existence of multiple dependencies, we need to establish multiple data paths among different match tables based on the dependencies during the layout. However, this layout solution brings about stalls due to the arising of potential conflicts; this situation is the same as that in Table 1. One typical solution used to avoid conflicts is to duplicate the match tables in order to establish all possible data paths. However, this solution incurs high resource-costs due to the fact that each match table takes up a lot of resources. For these reasons, both of them are unsuitable for the FPGAs. Instead, the multiple dependencies need to be simplified in order to enable a pipeline layout solution to be found; this solution will reduce the resource usage and process latency for the FPGAs. Thus, we simplify the TDG by going through all nodes in the TDG, and the method is as below:

- Precedence exists among these four dependencies, as follows: ''Match dependency'' > ''Action dependency'' > ''Successor dependency'' > ''Reverse match dependency'';
- If more than one dependency exists between two tables, reserve the highest-priority dependency;
- If a table has more than one predecessor, these predecessors can be constructed into a dependency chain, so the highest priority dependency is reserved and the remaining dependencies are ignored. If there is no dependency between any of the two predecessors, reserve all its dependencies.

A simplified TDG is shown in Fig.7(b). However, multiple dependencies may still exist in the graph, such as the table ''Unicast''. Two predecessors of this kind can be merged into one virtual table, and the dependencies between the virtual table and its successors are also updated. Thus, all tables have only one predecessor apart from the entry table; this result is shown in Fig.7(c), which will be used for the pipeline schedule.

3) PIPELINE SCHEDULE FOR THE MATCH-ACTION ENGINE

Based on the dependencies, the pipeline structure with dependency tables (including the table order, pipeline branches, etc.) can be determined by simplifying the TDG. For example, in Fig.7(c), the ''Successor dependency'' generates two branches for the pipeline, while the other dependencies decide the table order in the branches. Moreover, for the match-action engine pipeline schedule, the following works are included.

Reorganize Dependency Tables: As shown in Fig.8, different dependency types decide how the multiple tables run in parallel. For example, if two tables exist in a match dependency, the table in stage 2 can only begin matching after completing the action of the table in stage 1. If the dependency between them is an action, the table in stage 2 starts the match after completing the match of the table in stage 1. For the successor dependency, reverse match dependency, and no-dependency, tables in sequential stages can start running at the same time. Accordingly, tables in different stages can be

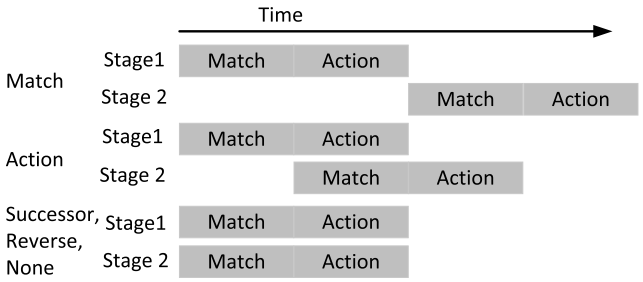


FIGURE 8. Dependency types and latency delays in FPGAs. Table in stage 2 depends on table in stage 1.

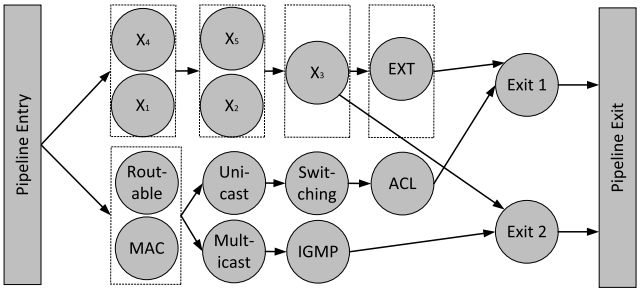


FIGURE 9. Pipeline scheduling for the match-action engine. The arrows indicate the dependencies between tables. Tables Xi represent the tables without dependency. Node "EXT" represents an extra pipeline stage for keeping data aligned. "Exit" Nodes represent the pipeline stages that combine the results from different pipeline branches.

reorganized based on their dependencies, matches, and the actions from different tables are run in parallel, which reduces the process latency in the match-action engine.

Add None-dependency Tables: In the match-action engine, no-dependency tables is the other aspect that needs to be planned out during the pipeline scheduling. All no-dependency tables can start executing at the same time, meaning that they can be arranged in one pipeline stage; however, it may cost extra resources to keep all fields aligned. We distribute these tables evenly over a pipeline branch, the stage number of which is equal to the minimum stage number in the existing pipeline branches.

Data Alignment: As tables can consist of match, action, or both, they process packet data with various latencies. In addition, different actions may take different numbers of cycles to complete. An extreme example is as follows: if we assume that our applications run at a clock rate of 500 MHz or higher, calculating $\sqrt{x}$ costs more cycles than x + 1. In addition, pipeline branches with different amounts of tables lead to further differences in latency. As a result, it is a challenge to keep all fields in one packet aligned to flow out of the match-action engine. To solve this data alignment problem, we need to first calculate the latencies for processing the fields in the tables, then insert pipeline stages to short pipeline branches in order to ensure that the results are outputted from associated pipeline branches at the same time.

For illustration, we assume that each table is one stage of the pipeline that costs a constant number of cycles. Fig.9 presents an example of scheduling the pipeline. In this

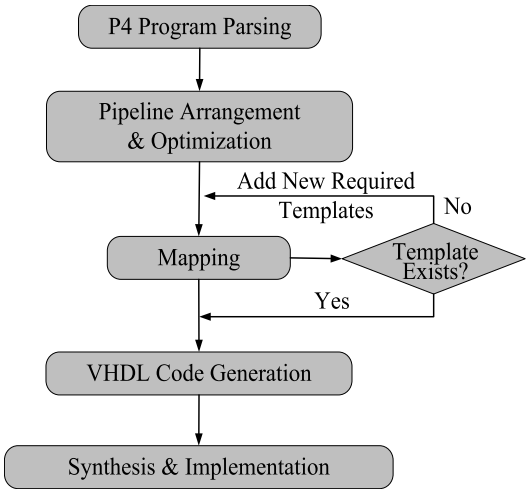


FIGURE 10. Workflow of the proposed compiler. Key steps in the compilation are shown in this figure, including calling the standard P4 compiler and the EDA tool.

diagram, there are three main pipeline branches: the first branch consists of table Xs, and the remaining branches are "Routable (MAC), Unicast, Switching, ACL" and "Routable (MAC), Multicast, IGMP", respectively. The first pipeline branch with tables X_i handles the fields without dependencies, while the other two branches handle the fields with dependencies. By adding the "EXT" stage, the results from the first pipeline branch are kept aligned with the results from the second one; moreover, the results from table "X3" are kept aligned with the results of the third one.

IV. CONVERSION OF P4 TO VHDL

The use of a high-level language such as P4 to design a network processor obviates the need for network programmers to be familiar with the details of the hardware. We now propose a compiler that converts P4 programs to VHDL, then uses standard EDA tools to generate bitstreams for the FPGA targets. This process can be used to reduce the effort required for development.

A. COMPILING PROCESS

Several steps are needed to convert a P4 program to VHDL: namely, P4 program parsing, pipeline scheduling, mapping, VHDL code generation, and synthesis and implementation. Fig.10 presents the workflow.

1) P4 PROGRAM PARSING

The official P4 compiler [37] is used to parse the P4 program. This generates an intermediate file in the extensible markup language that can be easily parsed by various backend compilers to facilitate their solution, such as generating executable instructions, or generating other high-level languages. Our proposed compiler is this type of backend compiler. This compiler extracts different functions of the described network processor associated with their parameter values, but for reconfiguring the VHDL templates.

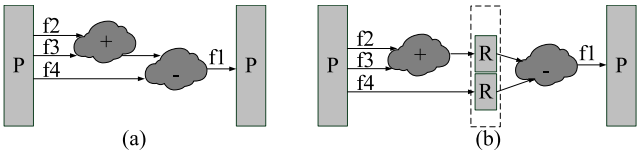


FIGURE 11. Optimization for increasing clock rate by inserting pipeline stages. "P" and "R" in this figure refer to "Pipeline Stage" and "Register Array", respectively.

2) PIPELINE SCHEDULING AND OPTIMIZATION

The parser pipeline stages of the processor are firstly scheduled for parsing the supported protocols without conflicts and halts, as described in Section III-C. Another function of this compiler is to schedule the pipeline in the match-action engine so as to reduce the processing latency, as described in Section III-D. To accomplish this, all related functions and their dependencies and latencies are firstly abstracted to form a Directed Acyclic Graph (DAG), which can be processed directly by the compiler.

In addition, the compiler needs to insert additional pipeline stages in order to keep all data aligned, as well as to ensure that the generated application can run at the specified frequency. As an example, consider a computation in the network processor: the processor calculates a result based on the formula of $pkt.f1 = pkt.f2 + pkt.f3 - pkt.f4$.[1] We can get a result in one cycle by implementing it with pure combinatorial logic; this implementation model is shown in Fig.11(a). However, the clock rate of the model in Fig.11(a) would be lower than the model with inserted registers (pipeline stage) shown in Fig.11(b). This is because the former has a longer propagation delay, while the latter achieves a shorter propagation delay by inserting registers (pipeline stages) to break the long path. More details of the optimization strategy are presented in Section IV-C.

3) MAPPING AND GENERATING HDL CODE

Once the pipeline consisting of the various functionalities, match tables, logic operations, connections, etc. has been determined, the compiler can map them to their related VHDL templates that are stored in the template library. The compiler initializes the templates by configuring their parameters; for example, CAM is initialized with width, depth, resource type, memory type, etc. The compilation is terminated if no corresponding template is found and resumes when the required template is added. More template details are provided in Section IV-B. Finally, the compiler generates VHDL code with instantiated templates and their wrappers.

B. TEMPLATE LIBRARY FOR CONVERTING

The availability of a modular template library for all basic functions used in the P4 program is a precondition for subsequent compilation to continue. We built up a VHDL template library, the templates of which are managed by a configuration file based on the XML format.

[1] Where $pkt.fi$ refers to the i field of the packet

1) TEMPLATES CREATION

A VHDL template for each basic function in P4 programs is created by experts in FPGA design. These functions may include the operations in the parser that extract fields, update vectors, or rams that stores the forwarding information, and operations of add, subtract, etc. Each template includes parameter values that control the final implementation, including resource usage, pipeline stages, performance, etc. Once these templates are instantiated by the compiler, they become basic components of the hardware architecture that can be reorganized to construct a network processor for different protocols. For example, different clock rates and table sizes for CAMs can be designed by setting the depth, width, and pipeline stages.

Table 2 presents a subset of our template library, which can be called and initialized by the compiler according to requirements. The templates in the first three categories are created based on the basic components of FPGA circuits, which enable the applications run on the FPGAs. The templates in the fourth category are created based on partial elements of our designed network processors, which themselves consist of basic components in the first three categories. Creating these element templates means that only a few parameters can be set, but complex functions can be achieved as a result; thus, the compiling process can be highly simplified.

2) ADD CUSTOMIZED COMPONENTS

Generally speaking, the existing templates in the library should be sufficient to enable the construction of a general network processor. However, customized functions or intellectual properties (IPs) should be used to support new features or improve performance. For illustration, we demonstrate how to add an "adder" template to the library. The XML text below describes the connection interface of this "adder", which includes function names from the P4 program and template names from the template library, as well as properties for hardware implementation.

```
<Component>
  <!—Sign or Name in P4~language—>
  <FunctionName> + </FunctionName>
  <TemplateName>Adder</TemplateName>
  <DefaultLatency>0</DefaultLatency >
  <Generics>
    <FrontPipelineStages>0</FrontPipelineStages>
    <AfterPipelineStages>0</AfterPipelineStages>
  </Generics>
  <Ports>
    <InPort>
      <Name>Clk</Name>
      <Type>std_logic</Type>
    </InPort>
    <InPort>
      <Name>Reset</Name>
      <Type>std_logic</Type>
    </InPort>
    <InPort>
      <Name>Input1</Name>
      <Type>std_logic_vector</Type>
      <Generic>Input_width</Generic>
    </InPort>
    <InPort>
```

TABLE 2. Partial template set.

Category	Description
Arithmetic	Adder, subtractor, multiply, inc, dec, comparison, counter, max, min,...
Vector logic	Shifter, selector, copy/ replace/ delete/ Field from Vector, vector index, rotate, merger, vector transpose,...
Memory	TCAM, BCAM, RAM, ROM, FIFO
Basic elements	Checksum, encrypt, decryption and certification(DES, MD5, SHA), parser pipeline stage, deparser pipeline stage, TCAM search frame, RAM search frame,...

```
      <Name>Input2</Name>
      <Type>std_logic_vector</Type>
      <Generic>Input_width</Generic>
    </InPort>
    <OutPort>
      <Name>Output</Name>
      <Type>std_logic_vector</Type>
      <Generic>Output_width</Generic>
    </OutPort>
  </Ports>
</Component>
```

There are five connection ports of this component, some of the type ''vector'' and some of the type ''bit''. Their properties, such as port width and registers (pipeline stages), can be changed by setting the related generics. By inserting the description into the configuration file, the compiler can find the template and initialize it using the values parsed from the P4 program. However, it is challenging for the compiler to connect its ports, as it is difficult to distinguish between ports of the same type. For example, if we initialize both of the two input ports to an 8-bit ''vector'' in the exponentiation template, the compiler does not know which port is the base and which is the exponent. Although the answer can sometimes be inferred through the name or the order of the ports in the description, sometimes human interaction is required.

C. BUILDING THE EVALUATION LIBRARY

Generally speaking, network programmers configure parameters based on need without considering hardware implementation issues. However, it may not be possible to implement some configurations in specific FPGAs in the desired way due to resource and/or timing limitations. Accordingly, this section presents an evaluation library to reduce such uncertainty and help the compiler to optimize the applications; the method adopted involves using the existing synthesized data of some configurations to estimate that of other configurations.

1) ESTIMATION MODEL AND SYNTHESIZED DATA

Network processors typically consist of many components of different functions, all of which are described by various HDLs. EDA tools compile the programs to circuits by using sequential primitives and combinatorial logics in the FPGAs. The circuit of each component can be abstracted, as shown in the middle frame in the below Fig.12: the combinatorial logics execute the computation, while the sequential

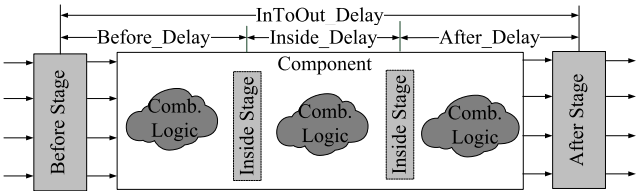


FIGURE 12. Estimation model. "Comb. Logic" denotes "Combinatorial Logic" and the pipeline stages can be constructed by any sequential primitives in FPGAs. The "Before Stage" and "After Stage" in this model simulate the last pipeline stages in its upstream and the first pipeline stage in its downstream, and there are no sequential primitives between these two stages and the component. The number of "Inside Stages" is theoretically within a range of natural numbers.

primitives (pipeline stages) store the medium results. Different components are connected to form the network processors, and each component in the processor can be abstracted to the model shown in Fig.12.

Based on the pipeline configurations of this model, four types of timing may exist. The ''InToOut_Delay'' represents the delay of the longest path between the ''Before Stage'' and the ''After Stage'', when the component is pure combinatorial logic. When there is at least one inside stage in the component, the ''Before_Delay'' represents the delay of the longest path between the ''Before Stage'' and the first inside stage, while the ''After_Delay'' represents the delay of the longest path between the last inside stage and the ''After Stage''. Moreover, the ''Inside_Delay'' represents the delay of the longest path between any inside stages of the component, and only exists when there are at least two inside stages.

Due to the consistency of the EDA tools, the same VHDL code always generates similar circuits for each of the different conditions by the same EDA tool, which means that the circuits have similar resource usage and timings. There is an inherent assumption here: namely, for a specific component, the results it synthesizes for a certain configuration can be estimated using the real synthesized results of a similar configuration. By extending this, the synthesized results of a circuit consisting of different components can also be estimated. For example, consider that we have the real synthesized timings and resource usage of adders with 32-bit, 36-bit and 40-bit input port width, respectively. If there is also an adder of 33-bit input port width, we can thus calculate suitable values for its synthesized timings and resource usage based on the existing adder data. Furthermore, if we have exact or estimated data for all components of an application, the synthesized timings and resource usage of this application can also be estimated. By using this model, we can thus generate the timings and resource usage for each component of any configuration with an EDA tool.

2) LIBRARY ESTABLISHMENT

To establish the evaluation library, the templates are firstly initialized under various configurations, then synthesized and implemented by EDA tools with a specific FPGA target. Here, we are unable to generate the timings and resource usage for each component of all possible configurations,

TABLE 3. Summary of parameters for VHDL templates.

Category	Parameters	Description
Functionality	Enable "EN",handshake type, reset type, signed/unsigned, round mode, overflow mode, ...	Change partial functionality of the component. The change has less impact on the final result. Each of these parameter has very few optional values.
Implementation	Resource type, register mode (pipelines), dual/single port, ...	Decide how to implement the component on FPGAs. These changes have a significant impact on timing and resource usage. Each of these parameter has very few optional values.
Normal	Port width, table depth, ...	Minor changes have less impact on resource usage and timings, and there is a relationship between the synthesis results and the parameters.

as this would result in a heavy workload and there is no need to enumerate all configurations. In practice, typical configurations are chosen for the estimation data generation, while other configurations can be estimated using the existing data. We record the generated data for each configuration in a database for reference, including the timings (the ''logic delay'' and ''propagation delay'' of each path delay are stored separately), resource usage (Flip Flop, block RAM, DSP, and LUTs), FPGA chips, and EDA tools.

The parameters of each component have different effects on resource usage and timings. Based on the influences they exert on the synthesized data, these parameters can be divided into three categories, as listed in Table 3. These parameters can be any of a number of data types, such as ''string'', ''boolean'', ''integer (natural, positive, and specific range)'', along with some customized data types that have different value ranges. Different configurations are chosen in order to establish the library; carefully choosing these configurations can not only reduce the scale of the database, but also help to improve our algorithm's estimation accuracy. More details are presented in Section IV-C4.

A simple rule is used to define the configuration set: the parameters ''Functionality'' and ''Implementation'' in Table 3 should enumerate all their values, while the parameters of the ''Normal'' should use sample values. The value ranges for the parameters of each template are formatted in an XML file, as this file type can be easily parsed by the library generator. Semicolons and colons are used to define the value ranges in this file; semicolons separate multiple independent values, while the format ''a:b:c'' defines a serial value whose start value is ''a'', step is ''b'', and end value is ''c''. For illustration, we take ''Shifter'' as an example to describe such a configuration set.

```
<ParameterRange>
  <Range name="SignedX">False</Range>
  <Range name="SignedN">False;True</Range>
  <Range name="BitWidthX">2:2:128</Range>
  <Range name="BitWidthN">2;4;8;16</Range>
  <Range name="Latency">1</Range>
  <Range name="RegisterMode">before;after</Range>
</ParameterRange>
```

''Shifter'' takes two inputs: one is the signal ''X'' to shift, and the other is the number of bits ''N'' to shift. ''SignedX'' and ''SignedN'' are both Boolean; the former distinguishes the shifts of ''Logical'' and ''Arithmetic'', while the latter indicates the shift direction. In addition, the parameter combination of ''Latency'' and ''RegisterMode'' decides the way in which the component pipeline is implemented. These parameters are of the types ''Functionality'' and ''Implementation'', and their values should be enumerated on demand. The bit widths of ''X'' and ''N'' are of the type ''Normal''; using the sample values is sufficient for the estimation. Based on the above settings, there will be 1024 configurations in the evaluation library (In practical terms, configurations in which ''BitWidthN'' is greater than ''BitWidthX'' are ignored).

So far, more than ten thousand configurations have been created for the ''Virtex-7'' series (one chip for one series; the chip of the smallest scale in the series is chosen to speed up the generation) with Vivado 2015.4.

3) THE APPLICATIONS OF ESTIMATION

Optimization: The compiler always tries its best to optimize the application in order to meet the users' requirements. The data in the evaluation library helps the compiler to do this. As an example, consider the function of the ''16-bit checksum'' with a 160-bit input width. By querying the library, the compiler discovers that the component can run at 210 MHz without any pipeline stage, but can run at 300 MHz if one pipeline stage is inserted. The compiler can then insert one register array and set the clock rate to a value between 210 MHz and 300 MHz. In our proposed solution, we choose the strategy of ''meet the minimum demands'' to select the configuration. Considering the example of checksum, if inserting one pipeline stage can meet the frequency requirement, we will not attempt to insert two pipeline stages. In addition, the compiler can automatically balance resource types: for example, memories can be implemented with LUTs or block RAMs in FPGA, and it implements some memories with LUTs once a few block RAMs are left.

Fast Reporting: Due to the inherent characteristics of FPGA, there may be situations in which the applications' resource usage exceeds the overall resources of the FPGA, or the clock rate is unable to meet design requirements. The users can only know the result after the programs are implemented, which is a time-consuming process. The evaluation library supports rapid reporting of the applications' resource usage and clock rates before implementation, thereby reducing the time costs. By querying the database, the compiler searches the estimation data for each component with the specified configuration during the compilation, and roughly calculates the resource usage and timings for reference. Accordingly, the users can decide their next step: changing

the hardware target, optimizing the code, modifying the required clock rate, etc.

4) ESTIMATION

For each specific component, there are only partial configurations in the evaluation library; their synthesized data is then used to estimate configurations that are not in the library, a technique referred to as static estimation. Based on the relationship between the parameters and the synthesized data, an equation can be established for the configurations, shown as (1).

$$f(x_1, x_2, \ldots x_m) = (TR_1, TR_2, \ldots TR_n)^2 \tag{1}$$

Below are the steps of the static estimation:

1) Find the N "closest" configurations in the evaluation library for the pending configuration, where the "closest" of these is defined as the one in which all parameters of the first two types described in Table 3 have the same values, and the parameters of "Normal" have the smallest errors. Taking the library of "Shifter" described in Section IV-C2 as the example, the configurations of {false, false, 6, 4, 1, before} and {false, false, 8, 4, 1, before} are the closest to the configuration of {false, false, 7, 4, 1, before}. The number of closest configurations "N" is decided by the demands of the estimation method. Currently, it is necessary to meet the requirements of linear/ polynomial interpolation calculation;
2) For the selected N configurations, the lowest-priority parameter (for "Normal" type, it is always the port width) is selected to arrange these configurations in ascending order. Based on these data, an equation set (2) can be established using (1);

$$\begin{cases} f_1(x_1, x_2, \ldots x_m) = (TR_1^1, TR_2^1, \ldots TR_n^1) \\ \qquad\qquad \ldots \\ f_p(x_1, x_2, \ldots x_m) = (TR_1^p, TR_2^p, \ldots TR_n^p) \\ \qquad\qquad \ldots \\ f_N(x_1, x_2, \ldots x_m) = (TR_1^N, TR_2^N, \ldots TR_n^N) \end{cases} \tag{2}$$

3) In this equation set, f_p is the pending estimation configuration, while $TR_1^p, TR_2^p, \ldots TR_n^p$ are the pending synthesized results. To calculate the resource usage and timings of each pending configuration, the equations can be simplified based on the synthesized result and its related parameters of the "Normal" type. For illustration purposes, we choose to calculate TR_1^p; the simplified equation set is shown as (3).

$$\begin{cases} f_1(x_1, x_2, \ldots x_l) = TR_1^1 \\ \qquad\qquad \ldots \\ f_p(x_1, x_2, \ldots x_l) = TR_1^p \\ \qquad\qquad \ldots \\ f_N(x_1, x_2, \ldots x_l) = TR_1^N \end{cases} \tag{3}$$

[2]where $x_i, (1 \le i \le m)$ indicate the parameters of the component and $TR_j, (1 \le i \le n)$ indicate the synthesized results.

Subsequently, a linear/ polynomial (as well multivariate) interpolation algorithm is used to estimate the value of "TR_1^p". The polynomial interpolation $f(x) = a * x^2 + b * x + c$ (where x is the bus width) is used in our design, as it is able to achieve high estimation accuracy. Other pending data are also calculated using the same method.

Estimating the applications that consist of various basic components is another task of the compiler's estimation module, which is called dynamic estimation. Essentially, estimating an application is similar to estimating a component, but is also the combination of various components. Compared with the static estimation, our experiments show that the dynamic estimation is not only affected by configurations of the components, but are also related to many other factors, including their upstream and downstream components, the circuits' scales, FPGA series, constraints, etc. Many of these factors are tuned by studying a large number of real applications, including applications in network and other domain in order to explore the relationship among the component combination, circuits' scales, constraints, etc.

The qualitative analysis method is used in the exploration. The main operation involves maintaining all variables unchanged except for the tested variable. This enables us to find out how this variable affects the synthesized results when it is changed to different values. For example, if we maintain the component (combination) and the constraints unchanged, but change the circuits' scales, we can determine out how the synthesized results change as a result. Based on the method, the more examples are studied, the higher the estimation accuracy that can be achieved. So far, about 100 different applications have been used for the training; the chip occupation rate ranges from 1% to 40%. More applications should be involved in the future.

The compiler estimates the applications based on the following steps:

1) Decompose the application to known and unknown components;
2) Query the database for each known component of their configurations and get their timings and resource usage. For those configurations that are not in the database, component estimation will be executed to get their estimation data, while for those components not in the evaluation library, the lowest clock rate and the highest resource usage of the known component are assigned;
3) The sum of the resource usage is used to evaluate the circuits' scales. The constraints are calculated based on the lowest clock rate of the component in the design. These are references for tuning the resource usage and timings for each component or component combination. Subsequently, the resource usage and timings of the used components can be updated based on the relationship among the circuits' scales, constraints, and components' orders;

TABLE 4. Resource consumption and clock frequency of different functions with typical parameter values.

Component	Width (bits)	Depth	Pipeline Stages	LUTs	FFs	BRAMs	Clock Rate (MHz)
Checksum	160	N/A	0	78	0	0	212
(16-bit)	160	N/A	1	165	39	0	298.5
BCAM	32	256	0	944	301	64	212.9
	48	256	0	1022	317	80	196
	48	128	0	581	188	40	220.5
TCAM	32	128	0	2548	87	0	184.2
	48	128	0	3648	119	0	177.4
	48	64	0	1948	117	0	205.1
SRAM	32	1024	0	1	0	2	543
(Block RAM)	64	2048	0	1	0	4	543
Add	64	N/A	0	64	0	0	512
	64	N/A	2	64	128	0	524.9
	64	N/A	4	159	131	0	424.8

4) New values of the resource usage and timings used in the design are calculated based on the results from the previous steps, and will be outputted as the estimation report. This report can be used to automatically optimize the design when the estimation results can not meet expectations. For example, by adding pipeline stages for the components that have a high delay. Steps 2 to 4 may need to be repeated many times until either the estimation results meet the requirements or we give up.

For the customized templates and the third-party IPs, users can generate synthesized data based on the model described in Fig.12 and add this data to the database.

V. EVALUATION

We have carried out a number of experiments in order to evaluate the performance of our proposed framework. In Section V-A, some typical components of the processor are synthesized and implemented in different configurations to examine how the parameter values affect the performance. In Section V-B, the processing latency of the parser and match-action engine are evaluated. In Section V-D, the effectiveness of the proposed framework is evaluated using a set of P4 applications. In order to ensure a fair comparison with the results in [25], we use the Xilinx XC7VX690T as the target FPGA and the Xilinx Vivado 2015.4 EDA tool to generate the FPGA bit-steam.

A. EFFECT OF PARAMETER VALUES

Many parameters can affect resource usage and timings, including the pipeline stage number, bus width and table depth. To meet our requirement of achieving the line rate of 100 Gbps by processing the smallest size packet (64 bytes), the clock rate should be about 195 MHz, while the clock constraints should be set to 5 ns. If the clock rate reaches 400 MHz, the constraints will be set to 2 ns in order to force the EDA tools to do more optimization.

1) PERFORMANCE WITH DIFFERENT PARAMETER VALUES

The goal of this experiment is to determine the relationship between configurations and their related performance, such as resource usage and clock rate. The implementation results of some typical components, along with their configurations, have been extracted from the evaluation library for illustrative purposes and are presented in Table 4.

From Table 4, different values of the parameters (such as ''width'', ''depth'' and ''pipeline stage number'') can be seen to have a positive or negative effect on performance. For example, compared with the instance without any pipeline stages, adding the extra pipeline stage in ''Checksum'' increases the clock rate from 212 MHz to 298 MHz while costing only an extra 87 (0.01%) LUTs and 39 (0.01%) FFs. However, for ''Add'', inserting four pipeline stages decreases the clock rate relative to inserting only two pipeline stages, and it also doubles the LUT cost.

Based on such estimation data, the compiler reports the performance before synthesis by the EDA tools and attempts to optimize the application. For illustration, let us consider ''Checksum'' as the example: if the user sets the required clock rate to be higher than 212 MHz, one pipeline stage should be inserted; otherwise, pure combinatorial logic implementation is chosen to reduce the resource usage.

TABLE 5. Clock rates of two sub-table organizations.

Type	Matrixes	In Parallel		Cascaded	
TCAM	Sub-table(s)	1	2	2	4
(48 bits * 64)	Clock Rate (MHz)	205.1	183.6	205.5	206.6

2) CAM IN PARALLEL VS. CAM IN PIPELINE

TCAM with low depth can only run at clock rates lower than 200 MHz. In practical applications, however, the depth should be far higher than 128, which would result in a bottleneck in the process chain. This experiment tries to solve this problem.

A large-size CAM can be divided into many small sub-tables capable of running fast. These small tables can be combined by using one of two types of processes: searching the same key in parallel sub-tables or in cascaded ones. Table 5 presents the results.

The results show that searching a key in two parallel sub-tables costs more time than searching a key in only one sub-table; by contrast, there is no significant change in

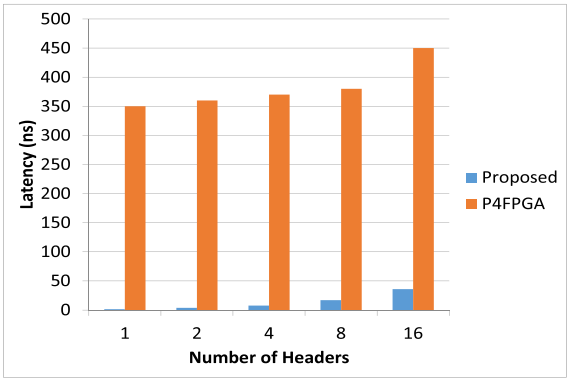


FIGURE 13. Parser latency v.s. number of headers parsed.

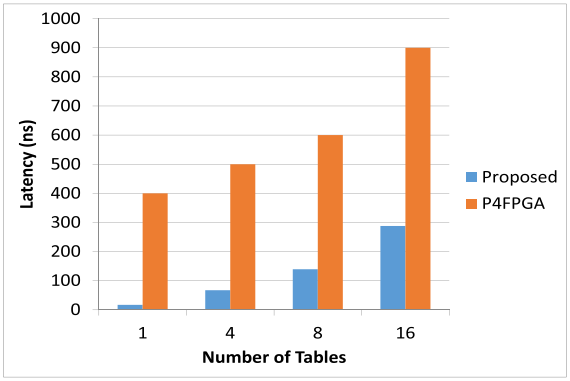


FIGURE 14. Processing latency v.s. number of tables.

search time between two cascaded sub-tables. As a result, the cascaded strategy is chosen for the compiler to do the CAM optimization.

B. FUNCTION MODULES COMPARISON

This experiment evaluates the performance of generated parser and match-action engine of different pipeline stages by comparing the results of P4FPGA; the comparison data are those reported by [25]. The generated header slices are set to 1024 bits, while about 50% of the bits of each protocol are extracted and loaded into the header vector during the parsing. In our design, the match-action engine consists of CAM, RAM, and actions, and each of these elements consumes one cycle to process the incoming packets. In addition, we set the size of the CAMs to 48-bit width * 128 depth, the size of RAMs to 64-bit width * 128 depth, and the width of the header vector to 512 bits. The same as that in P4FPGA, the ''Latency'' metric is used to evaluate the performance of our proposed solution; Fig.13 and Fig.14 present the comparison results by using this metric.

As can be seen from Fig.13, the proposed parser takes less than 40 ns to parse a header of 16 protocols, whereas P4FPGA takes about 450 ns. Moreover, Fig.14 shows that the match-action engine of 16 pipeline stages takes less than 300 ns to process a packet, while P4FPGA takes about 900 ns.

TABLE 6. Evaluation of the static estimation.

Test Case	Resource Error[a]				Timing Error[b]			
	L1[c]	L2	L3	L4	L1	L2	L3	L4
Adder	20	0	0	0	18	0	1	1
FIFO	20	0	0	0	20	0	0	0
Shifter	20	0	0	0	19	0	0	1
Parser Pipeline Stage	18	2	0	0	12	7	1	0

[a] The estimation errors are calculated separately based on different resources and the highest error is used.
[b] The highest delay is used to calculate the estimation error.
[c] The error levels are: L1 ($0 \leq Error \leq 5\%$), L2 ($5\% < Error \leq 10\%$), L3 ($10\% < Error \leq 15\%$, and L4 ($Error > 15\%$)).

As a result, it is evident that our proposed solution takes far less time to process the packets in the parser and match-action engine.

C. ESTIMATION METHOD EVALUATION

To evaluate the accuracy of the estimation methods, static estimation and dynamic estimation experiments are performed. We use the ''estimation error'' metric to evaluate the estimation methods, including estimating the resource usage and clock rate. This error is calculated by (4) when the estimation values and synthesized values are generated.

$$\left|\frac{stimationValue - realValue}{realValue}\right| * 100\% \tag{4}$$

1) EVALUATION OF THE STATIC ESTIMATION

In this experiment, the components ''Adder'', ''Fifo'', ''Shifter'' and ''Parser Pipeline Stage'' are chosen for illustrative purposes; here, the ''Parser Pipeline Stage'' is a kind of combination of the other components and has complex functionality. In addition, 20 configurations that are not included in the evaluation library are chosen randomly for each component. The estimation data and synthesized data of these configurations are generated separately, based on our method and a related EDA tool, and their corresponding errors are calculated. We divide these errors into different levels and record the numbers of the errors at each of these levels separately. Table 6 presents the experimental results.

In Table 6, there are only two instances of ''Adder'' and ''Shifter'' that whose timing estimation errors are outside of the range of 0 to 15%, and their bus widths are both 2 bits. By investigating these instances, we can determine that the timings have a significant error when the bus width increases from 1 to 2, and that the result calculated by the fitting formula is unable to cover this difference. For the first three components, all resource estimation errors are under 5%. However, there are 12 instances in error level 2 and level 3 for the ''Parser Pipeline Stage'' component. By investigating these instances, we can determine that there are 10 instances with bus widths that are higher than 768 bits. From this result, we can conclude that the static estimation method achieves higher accuracy when estimating the resource usage than when estimating timings; moreover, it has higher accuracy when estimating simple circuits than when estimating

TABLE 7. Evaluation of the dynamic estimation.

Domain	Resource Error[a]				Frequency Error			
	L1[b]	L2	L3	L4	L1	L2	L3	L4
Network (7)	6	1	0	0	5	1	1	0
Image (3)	0	3	0	0	0	0	2	1

[a] The estimation errors are calculated separately based on different resources and the highest error is used.
[b] The error levels are: L1 ($0 \leq Error \leq 15\%$), L2 ($15\% < Error \leq 20\%$), L3 ($20\% < Error \leq 30\%$, and L4 ($Error > 30\%$)).

complex circuits. In fact, we can obtain similar conclusions by applying our method to other components in the evaluation library. In general, the estimation accuracy meets our expectations.

2) EVALUATION OF THE DYNAMIC ESTIMATION

In this experiment, DAGs of the test applications are inputted into the estimation module. Their frequency and resource usage are then estimated based on the steps described in IV-C4. Based on the estimation report, the constraints for the applications are calculated by the estimated clock rate, which is used to synthesize its corresponding applications. We also divide these errors into different levels, albeit of different ranges. A total of 10 applications/ functions are chosen for evaluation, such as a ''parser'', ''deparser'', ''IPv4ToIPv6'', and so on. We also select three applications in the image processing domain, such as ''YCrCb to RGB'' and ''1-D/ 2-D Discrete Cosine Transform (DCT)'', as they consist of the same basic components and are not included in the evaluation library; for example, the components of ''float multiply''. The results are shown in Table 7.

From the table, it can be seen that this module has high estimation accuracy for applications in the network domain, and only one resource estimation error exceeds our expectation of 15%. However, two timing estimation errors exceed 15%. For the additional three applications in the image processing domain, the estimation accuracy is not high, and all errors exceed our expectations. Through investigation, for the high error instances in the network domain, we determine that the error sources are from unknown component combinations, while the error sources in the image processing applications are from both unknown component combinations and components that are not in the evaluation library. In general, the method of estimating the applications of complex functions and high resource consumption should be improved; moreover, more real applications should be used to train our estimation model.

D. EXAMPLE APPLICATION COMPARISON

Three use cases [38] are chosen to implement their cores using the proposed framework, which include the parser, match-action engine, and deparser. The ''Basic'' application is a basic layer-3 forwarding application with two protocols in the packet header. Many key fields are extracted by the parser, and these fields decide how the processor will process the incoming packets: forwarding it to a specific destination, dropping the packets, or taking no action. The ''Mri'' application is a program for extending the layer-3 forwarding that allows users to track both the path and the length of queues that every packet travels through. Extra information, such as identification (ID) and queue lengths in the related queue, needs to be added to the output packet. The ''Calc'' application implements a typical state networking application that uses a custom protocol header to implement some basic operations with the input operands by the network processor, then returns the result to the sender.

1) RESOURCE USAGE

As shown in Table 8, each of these three applications uses less than 1.5% of LUTs, 2.0% of flip flops, and 0.2% of block RAMs. This result indicates that there are a lot of resources in the FPGA that can be used to implement other functions in the processor, including more buffers to store intermediate results and fixed-function runtime for different boards, such as PCIe interface, MACs, and so on.

2) THROUGHPUT

These three applications run at a clock rate higher than 160 MHz. The throughput can be calculated by multiplying the clock rate by the input bus width, which shows the processing capability of the combination of the parser, the match-action engine, and the deparser; the final throughputs are listed in the last column of Table 8. Considering the smallest size packet limitation and the real network environment, the real throughput of ''Basic'' and ''Calc'' should be higher

TABLE 8. Performance of example applications.

Use cases[a]	Input Bus Width (bits)	LUTs	FFs	BRAMs	Constraints (ns)[b]	Estimation Error		Clock Rate (MHz)	Latency (cycles)	Throughput (G bps)[c]
						Timing	Resource			
Basic	272	5636 (1.30%)	1943 (0.20%)	2 (0.13%)	6	5.8%	3.3%	176.9	7	48.12
Mri	880	8003 (0.96%)	8354 (1.85%)	2 (0.13%)	6	0.1%	4.5%	168.3	10	148.1
Calc	240	693 (0.16%)	1445 (0.17%)	0	3	10.0%	2.6%	307.5	4	73.8

[a] Tables depth in the processor is set to 256.
[b] Constraints are set based on estimation values.
[c] Throughput for processing the header vector, payload is not included.

than 90.6 Gbps and 157.4 Gbps. In addition, the performance of our design is related to the packet rate: the larger the size of the packet, the higher the data rate that can be achieved. Moreover, the clock rate also shows that the packet rates achieved are close to 170 Mpps.

3) ESTIMATION

For these three application, we also use the ''estimation error'' metric to evaluate the estimation accuracy of the compiler during the converting process. The ''Estimation Error'' column presents the clock rate error and resource error, and both of them are less than 15%, which is in line with our expectations.

VI. FUTURE WORK

Based on the proposed framework, P4 programs are efficiently and rapidly converted to VHDL, then synthesized and implemented using standard EDA tools. Effectively decomposing the processor into small components and creating VHDL templates for these components are critical aspects of the compilation flow. Future work will include adding more templates for components not shown in Fig.1, as well as upgrading the existing library templates to further reduce resource usage and increase the clock rate. In addition, programmable parsers and deparsers can be added to the library as alternatives.

Other future tasks include improving the accuracy of estimation for resource usage and timings, as well as applying the estimation model in a wide range of applications. In addition, optimization levels should be added to balance the frequency and resource usage.

Finally, the hardware architecture can be improved at both the system and micro-architecture levels. For example, a solution involving multi-chip architecture is currently being studied. By working with off-chip DRAMs and CAMs, FPGAs can use all of the available resources to improve the logical processing units and avoid the limitations of RAMs and CAMs in FPGA. High-performance buses such as AXI, which greatly improve flexibility, can also be used in the architecture.

VII. CONCLUSION

We proposed a template-based framework, which converts P4 programs to VHDL and automatically implements them in FPGA, to generate a network processor based on a match-action architecture. To ensure that the generated processors use few resources while still providing high performance, the framework implements the following approaches:

- All components and their bus widths in the processors are generated on demand. Resources are therefore not wasted in the generation of extraneous circuits.
- A pre-built library of well-designed and thoroughly tested templates corresponding to functions in the P4 programs are used in the compilation.
- The optimization of the processors is based on managing the basic components that construct the processor. Process latency is reduced by increasing the parallelism of the components, while the clock rate is increased by adding pipeline stages based on the evaluation library. Even though conflicts may exist between reducing latency and increasing the clock rate, the compiler attempts to find a good trade-off.

The framework rapidly converts P4 programs to VHDL which greatly reduces the effort required to design the network processor for an FPGA platform. Our experiments demonstrate that the generated processor uses FPGA resources efficiently and can achieve a line rate of nearly 100 Gbps for a basic layer-3 forwarding application.

REFERENCES

[1] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, ''The click modular router,'' *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 217–231, Dec. 1999, doi: 10.1145/319344.319166.

[2] J. Lovato. *Data Plane Development Kit (DPDK) Further Accelerates Packet Processing Workloads, Issues Most Robust Platform Release to Date*. Accessed: Jun. 21, 2018. [Online]. Available: https://www.dpdk.org/news/press/

[3] USA. *Switches for every network*. Accessed: Dec. 15, 2018. [Online]. Available: https://www.cisco.com/c/en/us/products/switches/index.html

[4] China. *Network Switches*. Accessed: Dec. 15, 2018. [Online]. Available: https://e.huawei.com/en/products/enterprise-networking/switches

[5] J. Morgan. (Sep. 2004). *Intel IXP2XXX Network Processor Architecture Overview*. [Online]. Available: https://www.slideserve.com/bikita/intel-ixp2xxx-network-processor-architecture-overview

[6] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui, ''Software-defined networking (SDN): A survey,'' *Security Commun. Netw.*, vol. 9, no. 18, pp. 5803–5833, Dec. 2016.

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, ''Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN,'' in *Proc. ACM SIGCOMM*, Hong Kong, 2013, pp. 99–110.

[8] S. Chole, ''dRMT: Disaggregated programmable switching,'' in *Proc. ACM SIGCOMM*, Los Angeles, CA, USA, 2017, pp. 1–14.

[9] USA. *Tofino*. Accessed: Nov. 2, 2018. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[10] *Intel Ethernet Switch FM6000 Series*, Intel, Santa Clara, CA, USA, 2013.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. M. Vahdat, G. Varghese, and D. Walker, ''P4: Programming protocol-independent packet processors,'' *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, doi: 10.1145/2656877.2656890.

[12] B. Li, K. Tan, L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, ''ClickNP: Highly flexible and high performance network processing with reconfigurable hardware,'' in *Proc. ACM SIGCOMM*, Florianópolis, Brazil, vol. 2016, pp. 1–14.

[13] USA. *P4$_1$6 Language Specification*. Accessed: Oct. 5, 2018. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[14] S. Singh and D. Greaves, ''Kiwi: Synthesis of FPGA circuits from parallel programs,'' in *Proc. 16th IEEE Int. Symp. Field-Program. Custom Comput. Mach.*, Apr. 2008, pp. 3–12.

[15] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, ''OpenFlow: Enabling innovation in campus networks,'' *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, doi: 10.1145/1355734.1355746.

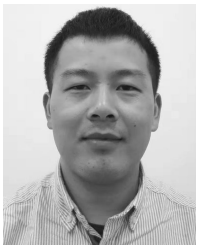
[16] D. Kreutz, F. M. Ramos, and P. Verissimo, ''Towards secure and dependable software-defined networks,'' in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, Hong Kong, 2013, pp. 55–60.

[17] A. Abdou, P. C. van Oorschot, and T. Wan, ''Comparative analysis of control plane security of SDN and conventional networks,'' *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 3542–3559, 4th Quart., 2018.

[18] EE Times. *Xilinx Introduces SDNet & 'Softly' Defined Networks*. Accessed: Mar. 31, 2013. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1321700#
[19] G. Brebner and W. Jiang, "High-speed packet processing using reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 8–18, Jan. 2014, doi: 10.1109/mm.2014.19.
[20] USA. *Software Development*. Accessed: Jun. 2019. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone.html
[21] J. Zhong. (Nov. 2018). P4 based packet processing with Xilinx. Xilinx, Beijing, China. [Online]. Available: https://www.sdnlab.com/22712.html
[22] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep./Oct. 2014.
[23] N. Zilberman, "NetFPGA: Rapid prototyping of networking devices in open source," *Proc. ACM SIGCOMM*, London, U.K., 2015, pp. 363–364.
[24] USA. *NetFPGA GitHub Organization*. Accessed: Oct. 7, 2018. [Online]. Available: https://github.com/NetFPGA
[25] H. Wang, "P4FPGA: A rapid prototyping framework for P4," in *Proc. SOSR*, Santa Clara, CA, USA, 2017, pp. 122–135.
[26] P. Benácek, V. Puš, H. Kubátová, and T. Cejka, "P4-To-VHDL: Automatic generation of high-speed input and output network blocks," *Microprocessors Microsyst.*, vol. 56, pp. 22–33, Feb. 2018.
[27] P. Benek, V. Pu, H. Kubtov, and T. Ejka, "P4-TO-VHDL," *SIGCOMM Comput. Commun. Rev.*, vol. 20, pp. 87–95, Aug. 2018.
[28] N. Sultana, S. Galea, D. Greaves, M. Wójcik, J. Shipton, R. Clegg, and P. Costa, "Emu: Rapid prototyping of networking services," in *Proc. USENIX ATC*, Santa Clara, CA, USA, 2017, pp. 459–471.
[29] G. Gibb, G. Varghese, M. Horowitz, and N. Mckeown, "Design principles for packet parsers," in *Proc. Archit. Netw. Commun. Syst.*, San Jose, CA, USA, Oct. 2013, pp. 21–22.
[30] V. Puš, L. Kekely, and J. Kořenek, "Low-latency modular packet header parser for FPGA," in *Proc. Proc. ACM ANCS*, Austin, TX, USA, Oct. 2012, pp. 77–78.
[31] J. S. da Silva, F.-R. Boyer, and J. M. Langlois, "P4-compatible high-level synthesis of low latency 100 Gb/s streaming packet parsers in FPGAs," in *Proc. FPGA*, Monterey, CA, USA, 2018, pp. 147–152.
[32] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Washington, DC, USA, Oct. 2011, pp. 12–23.
[33] Z. Cao, H. Zhang, J. Li, M. Wen, and C. Zhang, "A fast approach for generating efficient parsers on FPGAs," *Symmetry*, vol. 11, no. 10, p. 1265, Oct. 2019, doi: 10.3390/sym11101265.
[34] X.-T. Nguyen, T.-T. Hoang, H.-T. Nguyen, K. Inoue, and C.-K. Pham, "An efficient I/O architecture for ram-based content-addressable memory on FPGA," *IEEE Trans. Circuits Syst., II, Exp. Briefs*, vol. 66, no. 3, pp. 472–476, Mar. 2019, doi: 10.1109/tcsii.2018.2849925.
[35] I. Ullah, Z. Ullah, and J.-A. Lee, "Efficient TCAM design based on multipumping-enabled multiported SRAM on FPGA," *IEEE Access*, vol. 6, pp. 19940–19947, 2018, doi: 10.1109/access.2018.2822311.
[36] A. Ahmed, K. Park, and S. Baeg, "Resource-efficient SRAM-based ternary content addressable memory," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 4, pp. 1583–1587, Apr. 2017, doi: 10.1109/tvlsi.2016.2636294.
[37] USA. *P4 Compiler*. Accessed: Oct. 5, 2018. [Online]. Available: https://github.com/p4lang/p4c
[38] USA. *Tutorials*. Accessed: Mar. 2019. [Online]. Available: https://github.com/p4lang/tutorials/tree/master/exercises
[39] G. Brebner, "Softly defined networking," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Austin, TX, USA, Oct. 2012, pp. 1–2.
[40] R. Nikhil, "Bluespec system verilog: Efficient, correct RTL from high level specifications," in *Proc. ACM/IEEE MEMOCODE*, Jun. 2004, pp. 69–70.
[41] L. Jose, L. Yang, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *Proc. NSDI*, Oakland, CA, USA, May 2015, pp. 103–115.
[42] *Cavium and XPliant Introduce a Fully Programmable Switch Silicon Family Scaling to 3.2 Terabits Per Second*, Cavium, San Jose, CA, USA, Sep. 2014.
[43] A. Freier, P. Karlton, and P. Kocher, "SSL protocol," in *The Secure Sockets Layer (SSL) Protocol Version 3.0*. New York, NY, USA: Columbia Univ. Press, vol. 2011, pp. 12–36.

ZHUANG CAO received the B.S. degree in automation from the National University of Defense Technology, Changsha, China, and the M.S. degree in computer science and technology from Xiangtan University. He is currently pursuing the Ph.D. degree with the Department of Computer Science in National University of Defense Technology.

His main research interests include high-performance computing, parallel computing, and reconfigurable computing.



HUAYOU SU (Member, IEEE) was born in 1985. He received the B.S., M.S., and Ph.D. degrees in computer science and technology from the National University of Defense Technology, Changsha, China.

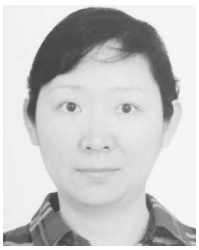
He is currently an Assistant Professor with the Computer College, National University of Defense Technology. His main research interests include high-performance computing, GPU programming, and parallel computing.

QIANMING YANG was born in 1984. He received the B.S., M.S., and Ph.D. degrees in computer science and technology from the National University of Defense Technology, Changsha, China.

He is currently an Assistant Professor with the Computer College, National University of Defense Technology. His main research interests include computer architecture, high-performance computing, and reconfigurable computing.

JUNZHONG SHEN received the M.S. degree in computer science and technology from the National University of Defense Technology, Changsha, China, in 2015, where he is currently pursuing the Ph.D. degree with the Department of Computer Science.

His current research interests include deep learning, computer architecture, and reconfigurable computing.



MEI WEN received the B.S., M.S., and Ph.D. degrees in computer science and technology from the National University of Defense Technology, Changsha, China, in 1995, 1999, and 2006, respectively.

She is currently a Professor with the Computer College, National University of Defense Technology. Her research interests include computer architecture, parallel programming, and scientific computing.

CHUNYUAN ZHANG (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and technology from the National University of Defense Technology, Changsha, China, in 1985, 1990, and 1996, respectively.

He is currently a Professor with the Computer College, National University of Defense Technology. He is also the Director of a series of research projects, including National Natural Science Foundation projects of China. His research interests include computer architecture, parallel programming, embedded systems, and scientific computing.